Bend 2 · HVM · Interaction nets — the compendium

*Claude edition, 2026-11-06. One of two documents: this **compendium** collects every source verbatim; the **coursebook** (`../coursebook/bend-hvm-inet-coursebook-claude.md`) teaches from them, with exercises and worked solutions, and cites this document by section id.*

How to use this document

- **Ids.** Every source file has a stable id (`A.4`, `H.46`, `L.7`) in its heading and in machine markers `<!-- ===== BEGIN FILE · <id> ===== -->` / `END FILE`. `grep -n "BEGIN FILE" …` lists every source; the ids of the parts kept from the first edition (A–I and K) are unchanged, so older references still resolve.
- **Fidelity.** Every part is verbatim: the only additions are banners and markers. Everything authored for this edition is outside the parts: this introduction, Appendix R (audit), the link index and the provenance.
- **Snapshots.** Some sources exist in several copies (the tutorial in Part A and inside the skills; HVM4 docs in Parts H, I and L). They are different snapshots; the newer one wins, and Appendix R says which is newer.

The parts

Part	What	Read it for
A	the site's Bend 2 tutorial (17 chapters, bend 2.0.25 era)	the language, BendRT, proofs, what released Bend 2 does and does not do
B	site pages: learn + notes (interaction nets, four interactions, HVM runtime, search by superposition)	short explanations with diagrams
C	`.opencode` Bend 2 skills (9)	project-distilled facts about Bend 2 targets, ownership, superposition, performance
D	`.opencode` hvm5-ic skill	the IC as used by the HVM5 project
E	research-to-release skill	HVM5 v1 maps and method
F	Bend2/HVM5/INET knowledge base	broad background, rollups, lineage
G	HVM5 specifications v1 and v2	homoiconic, forkable runtime design; **v2 README G.51, REVIEW G.52, code-as-data G.57**; v1 README G.2; model↔machine bridge G.5
H	primary sources: Victor Taelin, HigherOrderCO, bend2.dev, bendlang/bend	the authors' own words (**H.33 IC README, H.41–H.55 Taelin posts**, H.1–H.4 Bend README and guides)
I	HVM4 **March 2026** snapshot (`2d7f28e`): docs + test suite	history: the multi-file `clang/` runtime with AOT, FFI, threads
K	workspace context: experiments, docs-site design notes	how this documentation project came about
L	**HVM4 at HEAD `6defdfc` (2026-05-29)**: README, AGENTS, primer, theory, core, memory, syntax, collapser, all 61 interaction sheets, test listing	**the current machine** (single sequential `src/hvm.c`)
R	audit ledger: what was wrong or outdated in the first edition, with evidence and fixes	trust calibration

Reading paths

Goal	Path
Understand interaction nets from zero	coursebook Parts 1–5 → B (interaction-nets, four-interactions) → H.33 → L.4
Understand the HVM4 machine and why it is built that way	coursebook Parts 8 and 8B → the HVM2 paper §13 (its limitations, quoted in the coursebook) → H.46 → L.2 (AGENTS: code map) → L.6 (memory) → L.8 (collapser) → L.9–L.69 (rules)
Understand Taelin's ideas in his words	H.42 → H.43 → H.46 → H.41 → H.53 → H.50/H.51 → coursebook Part 11
Understand Bend 2 as released	A (chapters 01–05, 14) → H.1–H.4 → C
Understand HVM5	G.51 → G.52 → G.57 → coursebook Part 10
Check what changed	Appendix R → L vs I

Victor Taelin: a map of his public work

The person behind the Interaction Calculus, HVM1–HVM4 and Bend. The sources below are the ones this workspace holds; read them in this order.

Topic	Source (id)	Online
The calculus itself: affine global variables, DUP/SUP, the rules	Interaction-Calculus README (H.33)	https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus)
Manifesto: why substitution is not physical and interaction combinators are	"Towards an optimal computer" (H.42)	—
Optimal evaluation explained from scratch ("copy incrementally")	"Optimal evaluation in 1 or 10 or 10 years" (H.43)	—
How HVM works: linear objects, lazy clone, sharing inside lambdas, memory format	"HVM: how it works" (H.46)	https://github.com/HigherOrderCO/HVM (https://github.com/HigherOrderCO/HVM)
History: Optlam, Lamping's abstract algorithm and oracle, why optimal reduction stalled	HOC history (H.41)	https://github.com/VictorTaelin/optlam (https://github.com/VictorTaelin/optlam)
The limits of Lévy optimality; unordered superpositions	(H.53)	—
Superposition as program search; the collapse monad; SAT; the sieve	H.50, H.51, H.44, H.49, H.52	—
Deforestation at run time; the linker; type checkers	H.45, H.47, H.54, H.55	—
The machines	HVM1–HVM4 READMEs (H.27–H.30), Part L	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3), https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)
Symmetric Interaction Calculus (HVM1's memory format)	cited in H.46	https://github.com/VictorTaelin/Symmetric-Interaction-Calculus (https://github.com/VictorTaelin/Symmetric-Interaction-Calculus)
The language	Bend (H.1–H.4), bend2.dev (H.8–H.23)	https://github.com/bendlang/bend (https://github.com/bendlang/bend), https://bend2.dev/ (https://bend2.dev/)
His profile and repositories	H.24	https://github.com/VictorTaelin (https://github.com/VictorTaelin)

Editions and drift

- **HVM4.** Parts H (`hvm4-docs` , September 2026 retrieval of older docs) and I (March 2026 clone) describe the runtime **before** commit `4bc2336` (2026-05-07, "Refactor HVM runtime into pure single source"). Part L is HEAD. At HEAD: build `clang -O2 -o src/hvm src/hvm.c` ; flags `-s -S -C[=N] -D -d -v` ; no AOT, FFI or threads.
- **Bend.** Part A was checked against bend 2.0.25; the local compiler is 2.0.39; H.1–H.4 were fetched fresh for the first edition.
- **HVM5.** v1 is superseded by v2 (G.2); v2's status table is dated 2026-10-23 (G.51).

Table of contents

Numbered by part (1–11) and by file id. Every id is a heading in this document.

1 · Part A — Bend 2 tutorial (complete)

- A.1 · `tutorial/index`
- A.2 · `tutorial/01-first-programs`
- A.3 · `tutorial/02-data-and-recursion`
- A.4 · `tutorial/03-ownership-and-copying`
- A.5 · `tutorial/04-interaction-nets`
- A.6 · `tutorial/05-parallelism-and-runtime`
- A.7 · `tutorial/06-duplication-superposition`
- A.8 · `tutorial/07-laws-and-proofs`
- A.9 · `tutorial/08-dependent-types-and-totality`
- A.10 · `tutorial/09-io-files-and-targets`
- A.11 · `tutorial/10-branching-and-self-change`
- A.12 · `tutorial/11-rules-and-errors`
- A.13 · `tutorial/12-syntax-reference`
- A.14 · `tutorial/13-glossary`
- A.15 · `tutorial/14-superposition-and-supgen`
- A.16 · `tutorial/15-capstone`
- A.17 · `tutorial/16-references`

2 · Part B — Bend 2 site pages: learn + notes

- B.1 · `learn/data-matching`
- B.2 · `learn/effects-targets`
- B.3 · `learn/first-programs`
- B.4 · `learn/gpu`
- B.5 · `learn/hello`
- B.6 · `learn/hvm-hands-on`
- B.7 · `learn/parallelism`
- B.8 · `learn/proofs`
- B.9 · `learn/sharing`
- B.10 · `learn/synthesis`
- B.11 · `learn/types-proofs`
- B.12 · `notes/four-interactions`
- B.13 · `notes/hvm-runtime`
- B.14 · `notes/interaction-nets`
- B.15 · `notes/optimal-computer`
- B.16 · `notes/proof-lineage`
- B.17 · `notes/release-status`
- B.18 · `notes/search-by-superposition`
- B.19 · `notes/vs-lean`
- B.20 · `notes/vs-mojo`
- B.21 · `notes/what-is-bend2`

3 · Part C — Bend 2 skills (.opencode → bend2*, verbatim)

- C.1 · bend-gme/version-1/.opencode/skills/bend2/SKILL.md
- C.2 · bend-gme/version-1/.opencode/skills/bend2/references/01-tutorial-index.md
- C.3 · bend-gme/version-1/.opencode/skills/bend2/references/02-what-is-bend2.md
- C.4 · bend-gme/version-1/.opencode/skills/bend2/references/03-hello.md
- C.5 · bend-gme/version-1/.opencode/skills/bend2/references/04-release-status.md
- C.6 · bend-gme/version-1/.opencode/skills/bend2/references/05-glossary.md
- C.7 · bend-gme/version-1/.opencode/skills/bend2-language/SKILL.md
- C.8 · bend-gme/version-1/.opencode/skills/bend2-language/references/01-first-programs.md
- C.9 · bend-gme/version-1/.opencode/skills/bend2-language/references/02-data-and-recursion.md
- C.10 · bend-gme/version-1/.opencode/skills/bend2-language/references/03-syntax-reference.md
- C.11 · bend-gme/version-1/.opencode/skills/bend2-language/references/04-rules-and-errors.md
- C.12 · bend-gme/version-1/.opencode/skills/bend2-language/references/05-first-programs-concepts.md
- C.13 · bend-gme/version-1/.opencode/skills/bend2-language/references/06-data-matching-concepts.md
- C.14 · bend-gme/version-1/.opencode/skills/bend2-proofs/SKILL.md
- C.15 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/01-laws-and-proofs.md
- C.16 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/02-dependent-types-and-totality.md
- C.17 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/03-proofs-concepts.md
- C.18 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/04-types-and-proofs-concepts.md
- C.19 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/05-proof-lineage.md
- C.20 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/06-bend-vs-lean.md
- C.21 · bend-gme/version-1/.opencode/skills/bend2-ownership/SKILL.md
- C.22 · bend-gme/version-1/.opencode/skills/bend2-ownership/references/01-ownership-and-copying.md
- C.23 · bend-gme/version-1/.opencode/skills/bend2-ownership/references/02-sharing-concepts.md
- C.24 · bend-gme/version-1/.opencode/skills/bend2-superposition/SKILL.md
- C.25 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/01-duplication-superposition.md
- C.26 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/02-superposition-and-supgen.md
- C.27 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/03-branching-and-self-change.md
- C.28 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/04-synthesis-concepts.md
- C.29 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/05-search-by-superposition.md
- C.30 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/SKILL.md
- C.31 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/01-io-files-and-targets.md
- C.32 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/02-effects-targets-concepts.md
- C.33 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/03-js-target-facts.md
- C.34 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/04-minebender-notes.md
- C.35 · bend-gme/version-1/.opencode/skills/bend2-performance/SKILL.md
- C.36 · bend-gme/version-1/.opencode/skills/bend2-app/SKILL.md
- C.37 · bend-gme/version-1/.opencode/skills/bend2-app/references/01-capstone.md
- C.38 · bend-gme/version-1/.opencode/skills/bend2-app/references/02-reading-real-repos.md
- C.39 · bend-gme/version-1/.opencode/skills/bend2-runtime/SKILL.md
- C.40 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/01-interaction-nets.md
- C.41 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/02-parallelism-and-runtime.md
- C.42 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/03-parallelism-concepts.md
- C.43 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/04-gpu-concepts.md
- C.44 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/05-hvm-hands-on.md
- C.45 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/06-interaction-nets-notes.md
- C.46 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/07-four-interactions.md
- C.47 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/08-hvm-runtime-notes.md
- C.48 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/09-optimal-computer.md
- C.49 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/10-bend-vs-mojo.md

4 · Part D — HVM5 Interaction Calculus skill (.opencode → hvm5-ic, verbatim)

- D.1 · bend-gme/version-1/.opencode/skills/hvm5-ic/SKILL.md
- D.2 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/01-lineage.md
- D.3 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/02-v2-readme.md
- D.4 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/03-v2-review.md
- D.5 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/04-v2-overview.md
- D.6 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/05-v2-calculus.md
- D.7 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/06-v2-runtime.md
- D.8 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/07-v2-compiler.md
- D.9 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/08-v2-code-as-data.md
- D.10 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/09-v2-islands.md
- D.11 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/10-v2-formats.md
- D.12 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/11-v2-trust.md
- D.13 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/12-v2-plan.md
- D.14 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/13-v2-map.md
- D.15 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/14-v1-calculus.md
- D.16 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/15-v1-hvm4-execution.md

5 · Part E — Runtime-project skills (.opencode → research-to-release, verbatim)

- E.1 · bend-gme/version-1/.opencode/skills/research-to-release/SKILL.md
- E.2 · bend-gme/version-1/.opencode/skills/research-to-release/references/01-skill-style.md
- E.3 · bend-gme/version-1/.opencode/skills/research-to-release/references/02-hvm5-v1-maps-readme.md
- E.4 · bend-gme/version-1/.opencode/skills/research-to-release/references/03-hvm5-v1-goal-map.md
- E.5 · bend-gme/version-1/.opencode/skills/research-to-release/references/04-hvm5-v1-system-map.md
- E.6 · bend-gme/version-1/.opencode/skills/research-to-release/references/05-hvm5-v1-types-map.md
- E.7 · bend-gme/version-1/.opencode/skills/research-to-release/references/06-hvm5-v1-spaces-map.md
- E.8 · bend-gme/version-1/.opencode/skills/research-to-release/references/07-env-and-keys.md
- E.9 · bend-gme/version-1/.opencode/skills/research-to-release/references/07-hvm5-v1-lifecycle-map.md
- E.10 · bend-gme/version-1/.opencode/skills/research-to-release/references/08-hvm5-v1-code-map.md
- E.11 · bend-gme/version-1/.opencode/skills/research-to-release/references/09-hvm5-v1-risks-map.md
- E.12 · bend-gme/version-1/.opencode/skills/research-to-release/references/10-hvm5-v1-slices.md
- E.13 · bend-gme/version-1/.opencode/skills/research-to-release/references/11-hvm5-v1-gates.md
- E.14 · bend-gme/version-1/.opencode/skills/research-to-release/references/12-hvm5-v1-acceptance.md
- E.15 · bend-gme/version-1/.opencode/skills/research-to-release/references/13-hvm5-v1-traceability.md
- E.16 · bend-gme/version-1/.opencode/skills/research-to-release/templates/01-artifact-templates.md

6 · Part F — Bend2 / HVM5 / INET knowledge base (bend-

runtime/knowledge, verbatim)

- F.1 · bend-runtime/knowledge/README.md
- F.2 · bend-runtime/knowledge/00-start-here/glossary.md
- F.3 · bend-runtime/knowledge/00-start-here/how-to-use-this-knowledge-base.md
- F.4 · bend-runtime/knowledge/00-start-here/overview.md
- F.5 · bend-runtime/knowledge/01-bend-language/io-effects-apps-modules.md
- F.6 · bend-runtime/knowledge/01-bend-language/laws-proofs.md
- F.7 · bend-runtime/knowledge/01-bend-language/match-recursion-arrays-parallelism.md
- F.8 · bend-runtime/knowledge/01-bend-language/quantities-kinds-templates.md
- F.9 · bend-runtime/knowledge/01-bend-language/syntax-reference.md
- F.10 · bend-runtime/knowledge/02-guides-cli/cli-tooling-limits.md
- F.11 · bend-runtime/knowledge/02-guides-cli/guides-index.md
- F.12 · bend-runtime/knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md
- F.13 · bend-runtime/knowledge/02-guides-cli/raw/BASE-TYPES-DUMP.md
- F.14 · bend-runtime/knowledge/02-guides-cli/raw/CLI-HELP-DUMP.txt
- F.15 · bend-runtime/knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md
- F.16 · bend-runtime/knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md
- F.17 · bend-runtime/knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md
- F.18 · bend-runtime/knowledge/03-hvm-inet-theory/01-interaction-nets-foundations.md
- F.19 · bend-runtime/knowledge/03-hvm-inet-theory/02-symmetric-combinators-rules.md
- F.20 · bend-runtime/knowledge/03-hvm-inet-theory/03-confluence-optimal-reduction.md
- F.21 · bend-runtime/knowledge/03-hvm-inet-theory/04-hvm-runtime-tags-memory.md
- F.22 · bend-runtime/knowledge/03-hvm-inet-theory/05-bend-compilation-fuel.md
- F.23 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/00-overview-architecture.md
- F.24 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/01-memory-model-no-gc.md
- F.25 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/02-schedulers-cpu-gpu.md
- F.26 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/03-ffi-determinism-io.md
- F.27 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/04-hvm5-mini-reference-port.md
- F.28 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/05-laws-proof-gate-cli.md
- F.29 · bend-runtime/knowledge/05-ecosystem/bendlang-bend-repo.md
- F.30 · bend-runtime/knowledge/05-ecosystem/higherorderco-org.md
- F.31 · bend-runtime/knowledge/05-ecosystem/papers-formalization-community.md
- F.32 · bend-runtime/knowledge/05-ecosystem/victor-taelin.md
- F.33 · bend-runtime/knowledge/06-examples/01-hello-area-adder.bend
- F.34 · bend-runtime/knowledge/06-examples/02-recursion-parallel-arrays.bend
- F.35 · bend-runtime/knowledge/06-examples/03-io-channels-tcp-files.bend
- F.36 · bend-runtime/knowledge/06-examples/04-laws-proofs-app-loop.bend
- F.37 · bend-runtime/knowledge/06-examples/05-bench-and-proof-templates.md
- F.38 · bend-runtime/knowledge/07-smart-contracts-dynamic/01-kindelia-model.md
- F.39 · bend-runtime/knowledge/07-smart-contracts-dynamic/02-hub-fork-upgrade.md
- F.40 · bend-runtime/knowledge/07-smart-contracts-dynamic/03-domain-types-hvm5-checklist.bend
- F.41 · bend-runtime/knowledge/08-roadmap-hvm5-spec/references-index.md
- F.42 · bend-runtime/knowledge/08-roadmap-hvm5-spec/spec-todo.md
- F.43 · bend-runtime/knowledge/08-roadmap-hvm5-spec/what-we-know-vs-gaps.md
- F.44 · bend-runtime/knowledge/09-hvm5-cluster-runtime/01-concepts.md
- F.45 · bend-runtime/knowledge/09-hvm5-cluster-runtime/02-architecture-layers.md
- F.46 · bend-runtime/knowledge/09-hvm5-cluster-runtime/03-domain-types.md
- F.47 · bend-runtime/knowledge/09-hvm5-cluster-runtime/04-api.md
- F.48 · bend-runtime/knowledge/09-hvm5-cluster-runtime/05-registry-naming.md
- F.49 · bend-runtime/knowledge/09-hvm5-cluster-runtime/06-update-fork-merge.md
- F.50 · bend-runtime/knowledge/09-hvm5-cluster-runtime/07-ocap-wires.md
- F.51 · bend-runtime/knowledge/09-hvm5-cluster-runtime/08-similar-projects.md
- F.52 · bend-runtime/knowledge/09-hvm5-cluster-runtime/09-implementation-plan.md
- F.53 · bend-runtime/knowledge/09-hvm5-cluster-runtime/10-meta-prompt-for-builder-llm.md
- F.54 · bend-runtime/knowledge/09-hvm5-cluster-runtime/README.md
- F.55 · bend-runtime/knowledge/ALL-COMBINED.md
- F.56 · bend-runtime/knowledge/ALL-COMBINED-MINI.md
- F.57 · bend-runtime/knowledge/ALL-COMBINED-MINI-KINDELIA.md
- F.58 · bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.md
- F.59 · bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.META-PROMPT.md
- F.60 · bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.SOURCES.md

7 · Part G — HVM5 runtime specifications (bend-runtime/hvm5, verbatim)

- G.1 · bend-runtime/hvm5/README.md
- G.2 · bend-runtime/hvm5/version1/README.md
- G.3 · bend-runtime/hvm5/version1/spec/00-system.md
- G.4 · bend-runtime/hvm5/version1/spec/01-calculus.md
- G.5 · bend-runtime/hvm5/version1/spec/01a-hvm4-execution.md
- G.6 · bend-runtime/hvm5/version1/spec/01b-contract-compiler.md
- G.7 · bend-runtime/hvm5/version1/spec/01c-bend2-to-inet-support.md
- G.8 · bend-runtime/hvm5/version1/spec/01d-self-hosted-compiler.md
- G.9 · bend-runtime/hvm5/version1/spec/01e-core-ir.md
- G.10 · bend-runtime/hvm5/version1/spec/02-homoiconic-code.md
- G.11 · bend-runtime/hvm5/version1/spec/03-types-isolation.md
- G.12 · bend-runtime/hvm5/version1/spec/04-subspaces.md
- G.13 · bend-runtime/hvm5/version1/spec/05-registry-system-contract.md
- G.14 · bend-runtime/hvm5/version1/spec/06-lifecycle.md
- G.15 · bend-runtime/hvm5/version1/spec/07-cluster.md
- G.16 · bend-runtime/hvm5/version1/spec/08-api-cli.md
- G.17 · bend-runtime/hvm5/version1/spec/09-security.md
- G.18 · bend-runtime/hvm5/version1/spec/10-benchmarks.md
- G.19 · bend-runtime/hvm5/version1/spec/11-acceptance.md
- G.20 · bend-runtime/hvm5/version1/spec/12-traceability.md
- G.21 · bend-runtime/hvm5/version1/spec/legacy/hvm5-inet-contract-runtime-spec-v0.md
- G.22 · bend-runtime/hvm5/version1/spec/README.md
- G.23 · bend-runtime/hvm5/version1/maps/00-goal.md
- G.24 · bend-runtime/hvm5/version1/maps/10-system.md
- G.25 · bend-runtime/hvm5/version1/maps/20-types.md
- G.26 · bend-runtime/hvm5/version1/maps/30-spaces.md
- G.27 · bend-runtime/hvm5/version1/maps/40-lifecycle.md
- G.28 · bend-runtime/hvm5/version1/maps/50-code.md
- G.29 · bend-runtime/hvm5/version1/maps/60-risks.md
- G.30 · bend-runtime/hvm5/version1/maps/README.md
- G.31 · bend-runtime/hvm5/version1/plan/00-plan.md
- G.32 · bend-runtime/hvm5/version1/plan/01-slices.md
- G.33 · bend-runtime/hvm5/version1/plan/02-gates.md
- G.34 · bend-runtime/hvm5/version1/plan/03-risks.md
- G.35 · bend-runtime/hvm5/version1/plan/legacy/hvm5-bendrt-contract-runtime-plan-v0.md
- G.36 · bend-runtime/hvm5/version1/plan/README.md
- G.37 · bend-runtime/hvm5/version1/bench/results/w3-2026-10-20.md
- G.38 · bend-runtime/hvm5/version1/compiler/README.md
- G.39 · bend-runtime/hvm5/version1/contracts/README.md
- G.40 · bend-runtime/hvm5/version1/src/README.md
- G.41 · bend-runtime/hvm5/version1/legacy/net-v0/readme.md
- G.42 · bend-runtime/hvm5/version1/legacy/lattice-v0/fitness.md
- G.43 · bend-runtime/hvm5/version1/legacy/lattice-v0/gaps.md
- G.44 · bend-runtime/hvm5/version1/legacy/lattice-v0/goal.md
- G.45 · bend-runtime/hvm5/version1/legacy/lattice-v0/graph.md
- G.46 · bend-runtime/hvm5/version1/legacy/lattice-v0/invariants.md
- G.47 · bend-runtime/hvm5/version1/legacy/lattice-v0/owners.md
- G.48 · bend-runtime/hvm5/version1/legacy/lattice-v0/placement.md
- G.49 · bend-runtime/hvm5/version1/legacy/lattice-v0/rejects.md
- G.50 · bend-runtime/hvm5/version1/legacy/lattice-v0/seams.md
- G.51 · bend-runtime/hvm5/version2/README.md
- G.52 · bend-runtime/hvm5/version2/REVIEW.md
- G.53 · bend-runtime/hvm5/version2/spec/00-overview.md
- G.54 · bend-runtime/hvm5/version2/spec/01-calculus.md
- G.55 · bend-runtime/hvm5/version2/spec/02-runtime.md
- G.56 · bend-runtime/hvm5/version2/spec/03-compiler.md
- G.57 · bend-runtime/hvm5/version2/spec/04-code-as-data.md
- G.58 · bend-runtime/hvm5/version2/spec/05-islands.md
- G.59 · bend-runtime/hvm5/version2/spec/06-formats.md
- G.60 · bend-runtime/hvm5/version2/spec/07-trust.md
- G.61 · bend-runtime/hvm5/version2/spec/attic/05-islands-leader.md
- G.62 · bend-runtime/hvm5/version2/maps/00-map.md
- G.63 · bend-runtime/hvm5/version2/plan/00-plan.md
- G.64 · bend-runtime/hvm5/reference/hvm5-mini/main.bend

- G.65 · `bend-runtime/hvm5/reference/hvm5-mini/laws.bend`
- G.66 · `bend-runtime/hvm5/reference/hvm5-mini/proof.bend`

8 · Part H — Primary sources: Victor Taelin, HigherOrderCO, bend2.dev,

upstream bendlang/bend (verbatim)

- H.1 · bendlang/bend · README.md
- H.2 · bendlang/bend · guide/GUIDE.md
- H.3 · bendlang/bend · guide/EFFECTS.md
- H.4 · bendlang/bend · guide/SHADERS.md
- H.5 · fresh fetch · retrieval record
- H.6 · openspec/changes/bend-docs-site/research/README
- H.7 · openspec/changes/bend-docs-site/research/concept-map
- H.8 · openspec/changes/bend-docs-site/research/RETRIEVED.txt
- H.9 · openspec/changes/bend-docs-site/research/sources/bend2-release-date
- H.10 · openspec/changes/bend-docs-site/research/sources/bend2-vs-bend1
- H.11 · openspec/changes/bend-docs-site/research/sources/bend2-vs-lean
- H.12 · openspec/changes/bend-docs-site/research/sources/bend2-vs-mojo
- H.13 · openspec/changes/bend-docs-site/research/sources/contents
- H.14 · openspec/changes/bend-docs-site/research/sources/gpu
- H.15 · openspec/changes/bend-docs-site/research/sources/hello
- H.16 · openspec/changes/bend-docs-site/research/sources/home
- H.17 · openspec/changes/bend-docs-site/research/sources/hvm4-explained
- H.18 · openspec/changes/bend-docs-site/research/sources/interaction-nets-explained
- H.19 · openspec/changes/bend-docs-site/research/sources/learn-index
- H.20 · openspec/changes/bend-docs-site/research/sources/parallelism
- H.21 · openspec/changes/bend-docs-site/research/sources/proofs
- H.22 · openspec/changes/bend-docs-site/research/sources/what-is-bend2
- H.23 · openspec/changes/bend-docs-site/research/repos/_meta.txt
- H.24 · openspec/changes/bend-docs-site/research/repos/victortaelin-profile
- H.25 · openspec/changes/bend-docs-site/research/repos/taelin-theorem-proving-vs-testing
- H.26 · openspec/changes/bend-docs-site/research/repos/HigherOrderCO_Bend-README
- H.27 · openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM1-README
- H.28 · openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM2-README
- H.29 · openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM3-README
- H.30 · openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM4-README
- H.31 · openspec/changes/bend-docs-site/research/repos/HigherOrderCO_Kind-README
- H.32 · openspec/changes/bend-docs-site/research/repos/VictorTaelin_Bend2-README
- H.33 · openspec/changes/bend-docs-site/research/repos/VictorTaelin_Interaction-Calculus-README
- H.34 · openspec/changes/bend-docs-site/research/repos/VictorTaelin_hvm4-formal-README
- H.35 · openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-collapser
- H.36 · openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-core
- H.37 · openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-memory
- H.38 · openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-syntax
- H.39 · openspec/changes/bend-docs-site/research/repos/hvm4-docs/ic-theory
- H.40 · openspec/changes/bend-docs-site/research/repos/hvm4-docs/primer
- H.41 · openspec/changes/bend-docs-site/research/victor/hoc-history
- H.42 · openspec/changes/bend-docs-site/research/victor/hoc-optimal-computer
- H.43 · openspec/changes/bend-docs-site/research/victor/hvm-10min
- H.44 · openspec/changes/bend-docs-site/research/victor/hvm-collapse-monad
- H.45 · openspec/changes/bend-docs-site/research/victor/hvm-deforestation
- H.46 · openspec/changes/bend-docs-site/research/victor/hvm-how
- H.47 · openspec/changes/bend-docs-site/research/victor/hvm-linker
- H.48 · openspec/changes/bend-docs-site/research/victor/hvm-mystery
- H.49 · openspec/changes/bend-docs-site/research/victor/hvm-sat
- H.50 · openspec/changes/bend-docs-site/research/victor/hvm-search1
- H.51 · openspec/changes/bend-docs-site/research/victor/hvm-search3
- H.52 · openspec/changes/bend-docs-site/research/victor/hvm-sieve
- H.53 · openspec/changes/bend-docs-site/research/victor/hvm-unordered-sup
- H.54 · openspec/changes/bend-docs-site/research/victor/kind-conv-checker
- H.55 · openspec/changes/bend-docs-site/research/victor/term-checker

9 · Part I — HVM4 runtime documentation (HigherOrderCO/HVM4 clone, verbatim)

- I.1 · forgecode_clone/hvm/README.md
- I.2 · forgecode_clone/hvm/CLAUDE.md
- I.3 · forgecode_clone/hvm/docs/primer.md
- I.4 · forgecode_clone/hvm/docs/theory/interaction_calculus.md
- I.5 · forgecode_clone/hvm/docs/hvm/core.md
- I.6 · forgecode_clone/hvm/docs/hvm/memory.md
- I.7 · forgecode_clone/hvm/docs/hvm/syntax.md
- I.8 · forgecode_clone/hvm/docs/hvm/collapser.md
- I.9 · forgecode_clone/hvm/docs/hvm/ffi.md
- I.10 · forgecode_clone/hvm/docs/hvm/interactions/alo_cop.md
- I.11 · forgecode_clone/hvm/docs/hvm/interactions/alo_dry.md
- I.12 · forgecode_clone/hvm/docs/hvm/interactions/alo_dup.md
- I.13 · forgecode_clone/hvm/docs/hvm/interactions/alo_lam.md
- I.14 · forgecode_clone/hvm/docs/hvm/interactions/alo_nam.md
- I.15 · forgecode_clone/hvm/docs/hvm/interactions/alo_nod.md
- I.16 · forgecode_clone/hvm/docs/hvm/interactions/alo_var.md
- I.17 · forgecode_clone/hvm/docs/hvm/interactions/and_era.md
- I.18 · forgecode_clone/hvm/docs/hvm/interactions/and_inc.md
- I.19 · forgecode_clone/hvm/docs/hvm/interactions/and_num.md
- I.20 · forgecode_clone/hvm/docs/hvm/interactions/and_sup.md
- I.21 · forgecode_clone/hvm/docs/hvm/interactions/app_ctr.md
- I.22 · forgecode_clone/hvm/docs/hvm/interactions/app_dry.md
- I.23 · forgecode_clone/hvm/docs/hvm/interactions/app_era.md
- I.24 · forgecode_clone/hvm/docs/hvm/interactions/app_inc.md
- I.25 · forgecode_clone/hvm/docs/hvm/interactions/app_lam.md
- I.26 · forgecode_clone/hvm/docs/hvm/interactions/app_mat_ctr.md
- I.27 · forgecode_clone/hvm/docs/hvm/interactions/app_mat_num.md
- I.28 · forgecode_clone/hvm/docs/hvm/interactions/app_mat_sup.md
- I.29 · forgecode_clone/hvm/docs/hvm/interactions/app_nam.md
- I.30 · forgecode_clone/hvm/docs/hvm/interactions/app_sup.md
- I.31 · forgecode_clone/hvm/docs/hvm/interactions/ddu_era.md
- I.32 · forgecode_clone/hvm/docs/hvm/interactions/ddu_inc.md
- I.33 · forgecode_clone/hvm/docs/hvm/interactions/ddu_num.md
- I.34 · forgecode_clone/hvm/docs/hvm/interactions/ddu_sup.md
- I.35 · forgecode_clone/hvm/docs/hvm/interactions/dsu_era.md
- I.36 · forgecode_clone/hvm/docs/hvm/interactions/dsu_inc.md
- I.37 · forgecode_clone/hvm/docs/hvm/interactions/dsu_num.md
- I.38 · forgecode_clone/hvm/docs/hvm/interactions/dsu_sup.md
- I.39 · forgecode_clone/hvm/docs/hvm/interactions/dup_dry.md
- I.40 · forgecode_clone/hvm/docs/hvm/interactions/dup_lam.md
- I.41 · forgecode_clone/hvm/docs/hvm/interactions/dup_nam.md
- I.42 · forgecode_clone/hvm/docs/hvm/interactions/dup_nod.md
- I.43 · forgecode_clone/hvm/docs/hvm/interactions/dup_sup.md
- I.44 · forgecode_clone/hvm/docs/hvm/interactions/eql_any.md
- I.45 · forgecode_clone/hvm/docs/hvm/interactions/eql_ctr.md
- I.46 · forgecode_clone/hvm/docs/hvm/interactions/eql_dry.md
- I.47 · forgecode_clone/hvm/docs/hvm/interactions/eql_era.md
- I.48 · forgecode_clone/hvm/docs/hvm/interactions/eql_inc.md
- I.49 · forgecode_clone/hvm/docs/hvm/interactions/eql_lam.md
- I.50 · forgecode_clone/hvm/docs/hvm/interactions/eql_mat.md
- I.51 · forgecode_clone/hvm/docs/hvm/interactions/eql_nam.md
- I.52 · forgecode_clone/hvm/docs/hvm/interactions/eql_num.md
- I.53 · forgecode_clone/hvm/docs/hvm/interactions/eql_sup.md
- I.54 · forgecode_clone/hvm/docs/hvm/interactions/eql_use.md
- I.55 · forgecode_clone/hvm/docs/hvm/interactions/mat_inc.md
- I.56 · forgecode_clone/hvm/docs/hvm/interactions/op2_era.md
- I.57 · forgecode_clone/hvm/docs/hvm/interactions/op2_inc.md
- I.58 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_era.md
- I.59 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_num.md
- I.60 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_sup.md
- I.61 · forgecode_clone/hvm/docs/hvm/interactions/op2_sup.md
- I.62 · forgecode_clone/hvm/docs/hvm/interactions/or_era.md
- I.63 · forgecode_clone/hvm/docs/hvm/interactions/or_inc.md
- I.64 · forgecode_clone/hvm/docs/hvm/interactions/or_num.md
- I.65 · forgecode_clone/hvm/docs/hvm/interactions/or_sup.md
- I.66 · forgecode_clone/hvm/docs/hvm/interactions/uns.md
- I.67 · forgecode_clone/hvm/docs/hvm/interactions/use_era.md
- I.68 · forgecode_clone/hvm/docs/hvm/interactions/use_inc.md
- I.69 · forgecode_clone/hvm/docs/hvm/interactions/use_sup.md
- I.70 · forgecode_clone/hvm/docs/hvm/interactions/use_val.md
- I.71 · forgecode_clone/hvm/test/*.hvm (aggregated)

10 · Part K — Workspace context: Bend experiments and docs-site design notes

- K.1 · bend-runtime/README.md
- K.2 · bend-branch-graph/README.md
- K.3 · bend-branch-lab/README.md
- K.4 · openspec/changes/bend-docs-site/proposal
- K.5 · openspec/changes/bend-docs-site/design

11 · Part L — HVM4 at HEAD `6defdfc` (2026-05-29), verbatim

- L.1 · HVM4/README.md
- L.2 · HVM4/AGENTS.md
- L.3 · HVM4/docs/primer.md
- L.4 · HVM4/docs/theory/interaction_calculus.md
- L.5 · HVM4/docs/hvm/core.md
- L.6 · HVM4/docs/hvm/memory.md
- L.7 · HVM4/docs/hvm/syntax.md
- L.8 · HVM4/docs/hvm/collapser.md
- L.9 · HVM4/docs/hvm/interactions/alo_cop.md
- L.10 · HVM4/docs/hvm/interactions/alo_dry.md
- L.11 · HVM4/docs/hvm/interactions/alo_dup.md
- L.12 · HVM4/docs/hvm/interactions/alo_lam.md
- L.13 · HVM4/docs/hvm/interactions/alo_nam.md
- L.14 · HVM4/docs/hvm/interactions/alo_nod.md
- L.15 · HVM4/docs/hvm/interactions/alo_var.md
- L.16 · HVM4/docs/hvm/interactions/and_era.md
- L.17 · HVM4/docs/hvm/interactions/and_inc.md
- L.18 · HVM4/docs/hvm/interactions/and_num.md
- L.19 · HVM4/docs/hvm/interactions/and_sup.md
- L.20 · HVM4/docs/hvm/interactions/app_ctr.md
- L.21 · HVM4/docs/hvm/interactions/app_dry.md
- L.22 · HVM4/docs/hvm/interactions/app_era.md
- L.23 · HVM4/docs/hvm/interactions/app_inc.md
- L.24 · HVM4/docs/hvm/interactions/app_lam.md
- L.25 · HVM4/docs/hvm/interactions/app_mat_ctr.md
- L.26 · HVM4/docs/hvm/interactions/app_mat_num.md
- L.27 · HVM4/docs/hvm/interactions/app_mat_sup.md
- L.28 · HVM4/docs/hvm/interactions/app_nam.md
- L.29 · HVM4/docs/hvm/interactions/app_sup.md
- L.30 · HVM4/docs/hvm/interactions/ddu_era.md
- L.31 · HVM4/docs/hvm/interactions/ddu_inc.md
- L.32 · HVM4/docs/hvm/interactions/ddu_num.md
- L.33 · HVM4/docs/hvm/interactions/ddu_sup.md
- L.34 · HVM4/docs/hvm/interactions/dsu_era.md
- L.35 · HVM4/docs/hvm/interactions/dsu_inc.md
- L.36 · HVM4/docs/hvm/interactions/dsu_num.md
- L.37 · HVM4/docs/hvm/interactions/dsu_sup.md
- L.38 · HVM4/docs/hvm/interactions/dup_dry.md
- L.39 · HVM4/docs/hvm/interactions/dup_lam.md
- L.40 · HVM4/docs/hvm/interactions/dup_nam.md
- L.41 · HVM4/docs/hvm/interactions/dup_nod.md
- L.42 · HVM4/docs/hvm/interactions/dup_sup.md
- L.43 · HVM4/docs/hvm/interactions/eql_any.md
- L.44 · HVM4/docs/hvm/interactions/eql_ctr.md
- L.45 · HVM4/docs/hvm/interactions/eql_dry.md
- L.46 · HVM4/docs/hvm/interactions/eql_era.md
- L.47 · HVM4/docs/hvm/interactions/eql_inc.md
- L.48 · HVM4/docs/hvm/interactions/eql_lam.md
- L.49 · HVM4/docs/hvm/interactions/eql_mat.md
- L.50 · HVM4/docs/hvm/interactions/eql_nam.md
- L.51 · HVM4/docs/hvm/interactions/eql_num.md
- L.52 · HVM4/docs/hvm/interactions/eql_sup.md
- L.53 · HVM4/docs/hvm/interactions/eql_use.md
- L.54 · HVM4/docs/hvm/interactions/mat_inc.md
- L.55 · HVM4/docs/hvm/interactions/op2_era.md
- L.56 · HVM4/docs/hvm/interactions/op2_inc.md
- L.57 · HVM4/docs/hvm/interactions/op2_num_era.md
- L.58 · HVM4/docs/hvm/interactions/op2_num_num.md
- L.59 · HVM4/docs/hvm/interactions/op2_num_sup.md
- L.60 · HVM4/docs/hvm/interactions/op2_sup.md
- L.61 · HVM4/docs/hvm/interactions/or_era.md
- L.62 · HVM4/docs/hvm/interactions/or_inc.md
- L.63 · HVM4/docs/hvm/interactions/or_num.md
- L.64 · HVM4/docs/hvm/interactions/or_sup.md
- L.65 · HVM4/docs/hvm/interactions/uns.md
- L.66 · HVM4/docs/hvm/interactions/use_era.md
- L.67 · HVM4/docs/hvm/interactions/use_inc.md
- L.68 · HVM4/docs/hvm/interactions/use_sup.md
- L.69 · HVM4/docs/hvm/interactions/use_val.md
- L.70 · HVM4/devs/test/ (generated listing, 218 programs)

12 · Appendix R — audit ledger · 13 · Link index · 14 · Provenance

PART A · Bend 2 tutorial (complete)

The site's own Bend 2 course: 16 chapters + index, written for readers who know Haskell or OCaml. Chapters 01-03 teach the language, 04 is interaction nets, 05 the runtime, 06/14 duplication-superposition (and what the released compiler does not have), 07/08 laws/proofs/dependent types, 09 effects, 10 branching/self-change, 11 checker rules and every diagnostic, 12 the full syntax sheet, 13 a glossary, 15 a capstone, 16 a map of community repositories. Checked against bend 2.0.25 (last updated 2026-10-20).

Source root: `bend-docs/src/content/tutorial` · **17 files · 201.6 KB**

▼ In this part (file list)

- **A.1** · `bend-docs/src/content/tutorial/index.md`
- **A.2** · `bend-docs/src/content/tutorial/01-first-programs.md`
- **A.3** · `bend-docs/src/content/tutorial/02-data-and-recursion.md`
- **A.4** · `bend-docs/src/content/tutorial/03-ownership-and-copying.md`
- **A.5** · `bend-docs/src/content/tutorial/04-interaction-nets.md`
- **A.6** · `bend-docs/src/content/tutorial/05-parallelism-and-runtime.md`
- **A.7** · `bend-docs/src/content/tutorial/06-duplication-superposition.md`
- **A.8** · `bend-docs/src/content/tutorial/07-laws-and-proofs.md`
- **A.9** · `bend-docs/src/content/tutorial/08-dependent-types-and-totality.md`
- **A.10** · `bend-docs/src/content/tutorial/09-io-files-and-targets.md`
- **A.11** · `bend-docs/src/content/tutorial/10-branching-and-self-change.md`
- **A.12** · `bend-docs/src/content/tutorial/11-rules-and-errors.md`
- **A.13** · `bend-docs/src/content/tutorial/12-syntax-reference.md`
- **A.14** · `bend-docs/src/content/tutorial/13-glossary.md`
- **A.15** · `bend-docs/src/content/tutorial/14-superposition-and-supgen.md`
- **A.16** · `bend-docs/src/content/tutorial/15-capstone.md`
- **A.17** · `bend-docs/src/content/tutorial/16-references.md`

A.1 · `bend-docs/src/content/tutorial/index.md`

Bend 2 tutorial: table of contents · `runnable` · updated 2026-10-20 · canonical `/tutorial/`

Bend 2 tutorial

This tutorial teaches Bend 2 to someone who already knows Haskell or OCaml. It covers the syntax and the checker's rules, how values are owned and copied, the interaction-net model behind the language's history, parallel execution, and how to state and prove properties as *laws*.

It is unofficial and not affiliated with Higher Order Company. The reference is bend2.dev `(https://bend2.dev/)` and the official guide, which `bend guide` prints from your installed compiler.

WHAT YOU CAN DO AT THE END

- Read a Bend 2 program, including its annotations, and predict whether the checker accepts it.
- State a law, prove it with a same-named `def`, and read the checker's messages when a proof fails.
- Say when a parallel call really runs in parallel, and how to measure it.
- Say what Bend 2 does **not** have (SupGen, `sup` syntax, copy-on-write forks of running code, homoiconic self-modification), and what to use instead.
- Open a real Bend repository, find its laws, entry point, tests, and effect boundary, and say what it proves and what it only measures.

TABLE OF CONTENTS

#	Chapter	You learn	Status
01	First programs	installing, the four run modes, functions, `let`, `match` on `Bool`, numbers, strings, tuples, lists	runnable
02	Data and recursion	algebraic data types, records, nested and multi-scrutinee matching, termination, imports, definition order	runnable
03	Ownership and copying	affine binders, `+`, `-`, `Data` versus `Type`, kinds, arrays, maps, templates, closures, folds	runnable
04	Interaction nets	cells, ports, active pairs, rules, confluence, the symmetric interaction combinators, labels and the text of the Interaction Calculus, β as one step	conceptual, pen and paper
05	Parallelism and the runtime	parallel calls, the fork shape, leaves, `!` GPU calls, `--threads`, measuring, auditing the emitted C	runnable
06	Duplication, superposition, SupGen	`dup` and `sup`, labels, search by superposition, and what Bend 2 does not have	conceptual, with checked parse errors
07	Laws and proofs	laws, `{==}`, match refinement, induction, rewrites, holes, the LAWS and PROOF layout, the gate	runnable
08	Dependent types and totality	types computed from values, `Unit` and `Empty`, length-indexed lists, termination, `@unsafe`	runnable
09	IO, files, and targets	`do` blocks, effects, fork and join, channels, files, arguments, C, JavaScript, and native builds	runnable
10	Branching and self-change	what SupGen, copy-on-write forks, and homoiconic code would need, and what Bend 2 offers instead	runnable, design
11	Rules and errors	the checker's rules, every official diagnostic with its real message, version differences	reference
12	Syntax reference	the model in three layers (kinds, types, values), every syntax form on one sheet, laws and proofs, evaluation, and the checker's messages for the common mistakes	runnable
13	Glossary for Haskell and OCaml programmers	term-by-term mapping and definitions	reference
14	Superposition, collapse, and SupGen	the Interaction Calculus behind `sup` and `dup`, labels, collapse, search with erasure, what is public about SupGen, and whether a cheap fork exists	runnable, needs the Interaction Calculus binary
15	Capstone: a checked command parser	a job-queue parser built from constructors, `Maybe`, two proved laws, and command-line arguments	runnable
16	Reading real Bend repositories	a curated map of community and official repositories: what each one demonstrates, which files to open first, and which composition patterns to copy	reference

READING PATH

Read 01, 02, and 03 in order. They teach the language itself. Then read 07 and 08 (laws, proofs, dependent types), 05 (parallelism), and 09 (effects). Chapters 04 and 06 are the machine model and history: read them before 10, which applies everything to forking and testing branches. Keep 11 to 13 open as references, with 12 for every syntax form. Finish with 15, which combines the parts into one checked program. Then use 16 as a map of real repositories to read next.

HOW TO READ THE BOXES

Some sections set their key points apart in three kinds of box:

> **Core insight.** The one idea that the next part depends on.

> **Definition.** What a term means in this tutorial.

Composed of: a `text` block lists what a thing is made of. A `-` starts each part, a `|` between names means one of them, and indentation shows nesting. So `CON | DUP | ERA` reads as one of three cells.

VERSIONS

- Checked with **bend 2.0.25** on 2026-10-17. Run `bend version` to confirm yours.
- The examples on bend2.dev are pinned to **2.0.36** (commit `ae1101c`). 39 of its 42 examples print the documented output on 2.0.25. The other three print `All terms check.` where the site prints `ALL PROOFS CHECK`. 8 of 9 official diagnostics produce an error on 2.0.25, with the wording differences listed in chapter 11. The ninth uses `f?` names that 2.0.25 cannot parse.
- Version differences that change what you type are listed in chapter 11. The most important one: `bend PROOF.bend --verdict` does not exist in 2.0.25.

HONEST LIMITS

1. **Bend 2 does not run on interaction nets.** Chapter 04 explains the model behind the language's history. The released compiler emits a flat C evaluator called BendRT, which has no active pairs at runtime.
2. **The released compiler has no SupGen, no `sup` syntax, and no homoiconic self-modification.** Chapter 06 shows the evidence. `&{1, 2}` is a parse error, and nothing in `bend base` or `bend guide` describes SupGen.
3. **"Cheap" and "fast" are measurements, not theorems.** A law proves a property of the values a program computes. Cost, memory, and speed come from running the program and counting what the emitted C does.

HOW THE EXAMPLES ARE CHECKED

Each ```` ```bend ```` block is a complete program. `scripts/check-tutorial.py` compiles and runs every one of them with the installed `bend`:

- a block followed by an ```` ```output ```` block must print exactly that text;
- a block followed by an ```` ```error ```` block must fail with those lines;
- a block followed by a ```` ```contains ```` block must print those lines;
- a block with no expectation must pass `bend FILE --check-only`.

`ic` and `ic-collapse` blocks run Victor Taelin's Interaction Calculus reference binary when `IC_BIN` points to it, and are reported as skipped otherwise. Fences marked `text`, `sh`, or `bend-syntax` are documentation and are not run.

```
python3 bend-docs/scripts/check-tutorial.py bend-docs/src/content/tutorial
```

A.2 · bend-docs/src/content/tutorial/01-first-programs.md

> **Bend 2 tutorial 01: first programs** · `runnable` · updated 2026-10-18 · canonical `/tutorial/01-first-programs/`

01: first programs

Every program in this tutorial was checked with `bend 2.0.25`. Blocks marked `output` show what the compiler printed. Blocks marked `error` show the checker rejecting a program, and the lines are the real messages.

INSTALL AND CHECK THE TOOLCHAIN

```
curl -fsSL https://bend-lang.com/install.sh | sh
bend version
bend guide
```

`bend version` prints the compiler version. `bend guide` prints the language reference that ships with the compiler, and `bend base` prints the standard library. Read the guide before you write Bend code: it is short and it is the authority.

`bend update` installs the latest compiler. The bend2.dev examples are checked against 2.0.36, which has the `--verdict` option. Version 2.0.25 does not have it, and chapter 07 explains what that option needs.

THE FOUR WAYS TO RUN A FILE

Command	What it does
`bend file.bend`	Checks the file, then runs `main`. An `IO main` runs compiled; a pure `main` is evaluated by the checker, which is slow for big work.
`bend file.bend --check-only`	Checks the file and its imports, and runs nothing. Use it after every definition.
`bend file.bend -o file`	Builds a native binary with clang 14 or later. Then run `./file --threads 8`.
`bend file.bend -o file.c`	Emits one C file. `-o file.js` emits JavaScript instead.

Two options matter for the binaries: `--threads N` sets the number of CPU threads, and `--gpu off` runs `!` calls on the CPU.

HELLO, WORLD

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

```
Hello, world!
```

- `import Base` loads the prelude. Every program in this tutorial starts with it.
- `def main() -> IO(Unit):` declares the entry point. `IO(Unit)` is an effect that returns `Unit`, the type with one value `Unit{}`. In Haskell, this is `main :: IO ()`.
- `do IO<Unit>:` opens a block whose lines run in order, like Haskell's `do`.
- `IO.print` is an effect defined in Base.

FUNCTIONS

```
import Base

def square(+x: U32) -> U32:
  (x * x : U32)

def main() -> U32:
  square(7)
```

```
49
```

- Parameter types follow `:`, and the result type follows `->`.
- The last expression is the result. There is no `return`.

- Every operator needs an annotation, even on `Nat`: `(n + 1n : Nat)`. One annotation can cover a whole expression of one type, as in `(x * x * x : U32)`. A bare `x * x` is rejected, as the next example shows. The bundled guide (`bend guide`) still says a bare operator means `Nat`; the 2.0.25 checker rejects that. An annotation covers only the operators inside its own parentheses: in `(U32.add(a + b, 1) : U32)`, the `a + b` needs its own annotation. Chapter 12 lists every rule of this kind.
- `+x` makes the parameter reusable, because `x` appears twice. Chapter 03 explains why plain parameters can be used only once.

```
import Base

def add(x: U32, y: U32) -> U32:
  x + y

def main() -> U32:
  add(40, 2)
```

```
- message  : a type for this operator (write (a + b : Nat))
```

In Haskell, `square :: Word32 -> Word32` and `square x = x * x` would compile without any annotation. Bend asks for the annotation because it does almost no type inference, and that keeps its error messages precise.

READING `:`, `->`, AND `=`

Three marks do most of the work:

- `name: Type` says *has type*. It appears in parameters, constructor fields, and `for` binders.
- `(expr : Type)` *annotates* an expression, and `{term : Type}` annotates one term, as in `{12 : U32}`. Bend does almost no inference, so you write the type where the checker needs it.
- A `:` at the end of a line *opens a block*: a body, a `case`, a `do`, or a `law`.
- `->` separates the parameters from the result type. `=` binds a value. A `:` never means "equals".

A signature reads from left to right:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name
```

- `name: String` gives the parameter `name` its type.
- `-> IO(String)` gives the result: an effect that returns a `String`.
- The final `:` opens the body. The `do IO<String>:` header is a separate form, with angle brackets. Chapter 12 lists the rules for both.

LET BINDINGS

```
import Base

def main() -> U32:
  +price = {12 : U32}
  tax = {3 : U32}
  (price + price + tax : U32)
```

```
27
```

- `tax = {3 : U32}` binds a value used once.
- `+price = {12 : U32}` binds a value that the body uses twice.
- Inside a `do` block, a let needs its type, as in `n : U32 = (6 * 7 : U32)`. Chapter 09 covers `do`.

WHY A LET NEEDS A TYPE

A `let` has no type to take from, and Bend does no inference, so a bare literal is rejected:

```
import Base

def main() -> U32:
  +price = 12
  (price + price : U32)
```

```
- expected : an annotated term (cannot infer)
```

The fix is an annotation, `{12 : U32}`. That is not a constructor: a constructor has a name in front of its braces, as in `Tok{3n}` (chapter 02). Braces with no name hold one term and its type.

BRACES OR PARENTHESES

Braces annotate one term, such as `{12 : U32}` or `{x : U32}`. An operator needs parentheses, so that the annotation covers it:

```
import Base

def main() -> U32:
  {6 * 7 : U32}
```

```
- message  : a type for this operator (write (a * b : Nat))
```

Write `(6 * 7 : U32)` instead. The parentheses only group and annotate. They do not delay the work: Bend is strict, so the term runs when its definition runs (chapter 12, section 11).

CONDITIONALS: MATCH, NOT IF

Bend has no `if`. You branch with `match` on `True{}` and `False{}`, and the scrutinee must be a parameter, not a computed call:

```
import Base

def choose(test: Bool, yes: U32, no: U32) -> U32:
  match test:
    case True{}:
      yes
    case False{}:
      no

def main() -> U32:
  choose(U32.is_lt(3, 5), 10, 20)
```

```
10
```

The computed comparison `U32.is_lt(3, 5)` is passed to `choose`, and `choose` matches its own parameter. Matching the call directly is rejected:

```
import Base

def main() -> U32:
  match U32.is_eq(1, 1):
    case True{}:
      1
    case False{}:
      0
```

```
- message  : a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)
```

NUMBERS

Bend has three numeric types. `Nat` is an unbounded natural, written with an `n` suffix (`3n`). `U32` is a 32-bit unsigned integer, and its arithmetic wraps. `F32` is a 32-bit float. This chapter uses `Nat` and `U32`.

```
import Base

def main() -> String:
  n = Nat.add(20n, 22n)
  w = (6 * 7 : U32)
  Nat.show(n) ++ " / " ++ U32.show(w)
```

```
"42 / 42"
```

`Nat` patterns use `0n` and `1n+p`, where `p` is the predecessor (chapter 02). `U32` wraps at 2^32:

```
import Base

def main() -> U32:
  (4294967295 + 1 : U32)
```

```
0
```

`F32` values print as written in a checked program (`1.5`). `F32` arithmetic, however, is declared in Base as a law, so the checker shows `F32.mul(1.5, 2.0)` as an unevaluated term. A native build computes it, and `IO.print(F32.show(F32.mul(1.5, 2.0)))` prints `3`.

STRINGS, TUPLES, AND LISTS

Strings join with `++`. `String.length` returns a `Nat`:

```
import Base

def main() -> IO(Unit):
  text = String.to_upper("bend")
  IO.print(text ++ " " ++ Nat.show(String.length("proof")))
```

```
BEND 5
```

A pair has type `A & B` and value `(a, b)`:

```
import Base

def sum(pair: U32 & U32) -> U32:
  match pair:
    case (a, b):
      (a + b : U32)

def main() -> U32:
  sum((19, 23))
```

```
42
```

A list is `Nil{}` or `Con{head, tail}`, and `[5, 10, 15]` builds one:

```
import Base

def sum(xs: List<U32>) -> U32:
  match xs:
    case Nil{}:
      0
    case Con{head, tail}:
      (head + sum(tail) : U32)

def main() -> U32:
  sum([5, 10, 15])
```

```
30
```

The list in that example is consumed as the function walks it. Chapter 03 explains which lists can be reused and how `List<&2, U32>` changes that.

EXERCISES

1. Double and add. Write `double_plus` so that it returns `2a + b`. The parameter `a` is used twice, so it needs `+`. Then `main` should print `25`.

```
import Base

def double_plus(+a: U32, b: U32) -> U32:
  (a + a + b : U32)

def main() -> U32:
  double_plus(10, 5)
```

```
25
```

2. A label from a comparison. Write `label(n)` that returns `"small"` when `n < 10` and `"big"` otherwise. Remember that `match` cannot inspect `U32.is_lt(n, 10)` directly, so pass it to a helper.

```
import Base

def pick(test: Bool) -> String:
  match test:
    case True{}:
      "small"
    case False{}:
      "big"

def label(n: U32) -> String:
  pick(U32.is_lt(n, 10))

def main() -> String:
  label(42)
```

```
"big"
```

NEXT

Chapter 02 covers data types, records, recursion and its termination rule, nested patterns, imports, and definition order.

A.3 · bend-docs/src/content/tutorial/02-data-and-recursion.md

Bend 2 tutorial 02: data and recursion · `runnable` · updated 2026-10-18 · canonical `/tutorial/02-data-and-recursion/`

02: data and recursion

This chapter covers the data types you define, how a `match` reads them, and the rule that makes recursion safe to check. Haskell and OCaml readers will recognize almost everything. The differences are the explicit annotations and the fixed order of definitions.

ALGEBRAIC DATA TYPES

```
import Base

type Shape is Data:
  Point{}
  Segment{length: Nat}

def measure(shape: Shape) -> Nat:
  match shape:
    case Point{}:
      0n
    case Segment{length}:
      length

def main() -> Nat:
  measure(Segment{3n})
```

```
3n
```

- `type Shape is Data:` declares a type and says its values may be reused (chapter 03). Each indented line is a constructor.
- Constructors always use braces, even without fields: `Point{}`.
- Fields are named inside the braces: `Segment{length: Nat}` builds `Segment{3n}`.
- Haskell's `data Shape = Point | Segment { length :: Nat }` is the same declaration.

TYPES, CONSTRUCTORS, AND VALUES

A type is a set of values: `Shape`. A constructor (often shortened to *ctor*) is a named way to build one of those values: `Point{}` and `Segment{length: Nat}` are the two constructors declared in `type Shape`. In Haskell, `data Shape = Point | Segment Nat` declares the same two constructors. A value is what a constructor builds: `Point{}` and `Segment{3n}` are both values of type `Shape`.

A name followed by braces, `K{...}`, is always a constructor. The braces do different jobs depending on where they appear:

- In a `type` declaration, `Segment{length: Nat}` lists the fields of the constructor.
- In an expression, `Tok{3n}` applies the constructor to `3n`. The compiler reads it as `Ctor(Tok, [3n])`.
- In a `match` pattern, `case Segment{length}:` takes the value apart and binds the field.

Braces without a name are a different form. `{12 : U32}` is an annotation, `Ann(12, U32)`, and it is not a constructor. Chapter 12 lists every use of braces.

```
import Base

type Shape is Data:
  Point{}
  Segment{length: Nat}

def main() -> Shape & Shape:
  (Point{}, Segment{3n})
```

```
(Point{}, Segment{3n})
```

Both values are built directly, and no `match` is needed to make them. `Point{}` takes no fields, and `Segment{3n}` takes one.

Base declares its own types the same way. `bend base --types` prints them:

Type	Constructors	Example value
`Nat`	`Zero{}` , `Succ{pred}`	`2n` , which is `Succ{Succ{Zero{}}}`
`Bool`	`False{}` , `True{}`	`True{}`
`Unit`	`Unit{}`	`Unit{}`
`Maybe<a, A>`	`None{}` , `Some{value}`	`Some{42}`
`Result<a, b, E, A>`	`Fail{error}` , `Done{value}`	`Done{5}`
`List<a, A>`	`Nil{}` , `Con{head, tail}`	`[1, 2]` , which is `Con{1, Con{2, Nil{}}}`
`String`	`SNil{}` , `SCon{head, tail}`	`"hi"` , which is `SCon{'h', SCon{'i', SNil{}}}`

Angle brackets name types. `Maybe<&2, U32>` is a type, and `Some{42}` is a value of that type. In Haskell, `Maybe a` is a type constructor and `Just x` is a data constructor. Bend keeps the two apart the same way.

A constructor needs its type from context. When a `let` has no context, write the type:

```
import Base

type Tok is Data:
  Tok{n: Nat}

def show(t: Tok) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  t : Tok = Tok{3n}
  show(t)
```

```
3n
```

RECORDS

A record is a type with one constructor:

```
import Base

type Point is Data:
  Point{x: U32, y: U32}

def translate(p: Point) -> Point:
  match p:
    case Point{x, y}:
      Point{(x + 1 : U32), (y + 2 : U32)}

def main() -> Point:
  translate(Point{3, 4})
```

```
Point{4, 6}
```

`match` consumes `p` and binds its fields. The result is a new `Point` . Nothing is mutated.

RECURSION AND THE TERMINATION RULE

Bend repeats work by recursion, and every recursive call must be checked to terminate. The rule: a recursive call reads its arguments from left to right, and each argument must be passed unchanged until one of them is a smaller part of its parameter. Put the argument that shrinks first.

```
import Base

def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}:
      acc
    case Con{head, tail}:
      sum(tail, (acc + head : U32))

def main() -> U32:
  sum([1, 2, 3, 4], 0)
```

```
10
```

`tail` is a smaller part of `xs`, which the `match` exposed, and `xs` is the first argument. The accumulator changes freely because it comes after the shrinking argument.

Counting with a machine integer does not work, because subtraction gives the checker no structural evidence. Count with a `Nat` and match its predecessor. The parameter often called *fuel* bounds a loop that the outside world decides when to stop:

```
import Base

def count_down(fuel: Nat, acc: U32) -> U32:
  match fuel:
    case 0n:
      acc
    case 1n+remaining:
      count_down(remaining, (acc + 1 : U32))

def main() -> U32:
  count_down(5n, 0)
```

```
5
```

A recursive call that passes its input unchanged is rejected:

```
import Base

def loop(n: Nat) -> Nat:
  loop(n)

def main() -> Nat:
  loop(1n)
```

```
- expected : a decreasing self-call
```

Mutual recursion is not allowed in a safe definition. Merge the two functions into one, with an extra argument that selects which branch to run. Chapter 11 covers the `@unsafe` escape hatch and why a law must never depend on it.

MATCHING IN DEPTH

Patterns nest. This one inspects the second element of a list:

```
import Base

def second(xs: List<U32>) -> U32:
  match xs:
    case Nil{}:
      0
    case Con{h, Con{k, rest}}:
      k
    case Con{h, Nil{}}:
      h

def main() -> U32:
  second([4, 9, 7])
```

```
9
```

`match` can take several scrutinees, and `_` matches anything:

```
import Base

def both(a: Nat, b: Nat) -> Nat:
  match a b:
    case 0n 0n:
      0n
    case 0n 1n+q:
      1n+q
    case 1n+p 0n:
      1n+p
    case 1n+p 1n+q:
      Nat.add(p, q)

def first_or_zero(a: Nat, b: Nat) -> Nat:
  match a b:
    case 0n _:
      0n
    case 1n+p _:
      1n+p

def main() -> Nat & Nat:
  (both(3n, 4n), first_or_zero(5n, 7n))
```

```
(5n, 5n)
```

Three rules govern every `match`:

1. The scrutinee is a parameter or a variable bound by an earlier pattern. `match U32.is_eq(1, 1):` is rejected (chapter 01).
2. Matches are exhaustive. This one omits `False{}`:

```
import Base

def pick(b: Bool) -> U32:
  match b:
    case True{}:
      1
```

```
- expected : cases for False
```

3. Scrutinees follow binder order. A `let` cannot come before a `match` on a parameter.

ABSENCE WITHOUT NULL: MAYBE

Bend has no null. A value that may be missing has type `Maybe<&2, A>`: `None{}` means no value, and `Some{x}` wraps one.

```
import Base

def add_strings(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a)
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> Maybe<&2, U32>:
  add_strings("40", "2")
```

```
Some{42}
```

Inside `do Maybe<&2, U32>:`, the first `None` ends the block with `None`, as the guide describes. `Maybe.default` turns the result back into a plain value:

```
import Base

def add_strings(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a)
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> U32:
  Maybe.default(&2, U32, add_strings("40", "oops"), 0)
```

```
0
```

A division that refuses to divide by zero uses the helper pattern from chapter 01: the helper matches its own parameter. `b` is marked `+` because the function uses it twice:

```
import Base

def pick_div(zero: Bool, a: Nat, b: Nat) -> Maybe<&2, Nat>:
  match zero:
    case True{}:
      None{}
    case False{}:
      Some{Nat.div(a, b)}

def safe_div(a: Nat, +b: Nat) -> Maybe<&2, Nat>:
  pick_div(Nat.is_eq(b, 0n), a, b)

def main() -> Maybe<&2, Nat>:
  safe_div(10n, 3n)
```

```
Some{3n}
```

IMPORTS AND MODULES

An import gives a file a name. The alias is local to the importing file:

numbers.bend

```
import Base

def three() -> Nat:
  3n
```

main.bend

```
import Base
import ./numbers.bend as Numbers

def main() -> Nat:
  Numbers.three()
```

```
3n
```

`import Base` needs no alias. Imports come before definitions in a file.

DEFINITION ORDER

Bend checks each definition against the definitions above it. Calling a function written below it is an error:

```
import Base

def main() -> U32:
  answer()

def answer() -> U32:
  42
```

```
- expected : a defined name
```

Move `answer` above `main` and the program checks. Datatypes follow a related rule. In `bend 2.0.25`, a constructor field may only name a type that is already defined above it:

```
import Base

type Outer is Data:
  Wrap{inner: Inner}

type Inner is Data:
  Leaf{n: Nat}

def main() -> Nat:
  0n
```

```
- expected : a defined name
```

The bend2.dev guide says datatypes may appear in any order. The compiler we checked does not allow it, so define the inner type first.

EXERCISES

1. Length of a list. Write `length(xs)` that returns the number of elements as a `Nat`. The head is not needed, and an unused binder is dropped without any annotation.

```
import Base

def length(xs: List<U32>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{head, tail}:
      1n+length(tail)

def main() -> Nat:
  length([1, 2, 3, 4])
```

```
4n
```

2. Largest element. Write `max_of(xs, best)` that folds a list and keeps the larger of each element and the running best. Use `U32.max`, which is in Base.

```
import Base

def max_of(xs: List<U32>, best: U32) -> U32:
  match xs:
    case Nil{}:
      best
    case Con{head, tail}:
      max_of(tail, U32.max(head, best))

def main() -> U32:
  max_of([3, 9, 4], 0)
```

```
9
```

NEXT

Chapter 03 explains why a plain binder can be used only once, and what `+` costs.

A.4 · bend-docs/src/content/tutorial/03-ownership-and-copying.md

Bend 2 tutorial 03: ownership and copying · `runnable` · updated 2026-10-18 · canonical `/tutorial/03-ownership-and-copying/`

03: ownership and copying

Haskell shares values by default, and a garbage collector frees them later. Bend does the opposite: every binder says how many times it may be used, and the checker enforces it. This chapter explains those quantities, the cost of each way to copy, and the containers that rely on them.

AFFINE BY DEFAULT

A plain binder may be used at most once. Dropping it is free:

```
import Base

def twice(n: Nat) -> Nat:
  Nat.add(n, n)

def main() -> Nat:
  twice(3n)
```

```
- expected : n
- observed : n (consumed more than once)
```

The fix is `+`, which marks the binder as reusable:

```
import Base

def twice(+n: Nat) -> Nat:
  Nat.add(n, n)

def main() -> Nat:
  twice(3n)
```

```
6n
```

`+` is allowed only when the binder's type is `Data`. Nat, U32, Bool, and records declared `is Data` qualify. Functions, arrays, and file handles never do.

DATA VERSUS TYPE

A type declared `is Data` may be copied with `+`. A type declared `is Type` may not:

```
import Base

type Token is Type:
  Tok{n: Nat}

def show(t: Token) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  t : Token = Tok{3n}
  +u = t
  show(u)
```

```
- expected : Data
- observed : Type
```

The same program works when `Token` is declared `is Data`, and it uses an annotation to build the value:

```
import Base

type Tok is Data:
  Tok{n: Nat}

def show(t: Tok) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  +t = {Tok{3n} : Tok}
  Nat.add(show(t), show(t))
```

```
6n
```

Think of a kind as a reuse budget. `Type` allows one use, and `Data` allows as many copies as the code asks for, marked with `+`. A type with arguments takes its kind from them, so `Maybe<&2, Nat>` is `Data` and `Maybe<&1, Nat>` is `Type`:

```
import Base

def dup(+m: Maybe<&2, Nat>) -> Maybe<&2, Nat> & Maybe<&2, Nat>:
  (m, m)

def main() -> Maybe<&2, Nat> & Maybe<&2, Nat>:
  dup(Some{4n})
```

```
(Some{4n}, Some{4n})
```

With `&1`, the same `+m` is rejected:

```
import Base

def dup(+m: Maybe<&1, Nat>) -> Maybe<&1, Nat> & Maybe<&1, Nat>:
  (m, m)

def main() -> Maybe<&1, Nat> & Maybe<&1, Nat>:
  dup(Some{4n})
```

```
- expected : Data
- observed : Type
```

THE MINUS SIGN: ERASED PARAMETERS

`-A` is a parameter that exists only at compile time. It appears in types and is deleted before the program runs. `keep_first` below ignores its second argument, which is dropped:

```
import Base

def twice(+n: Nat) -> Nat:
  Nat.add(n, n)

def keep_first(-A: Type, x: A, y: A) -> A:
  x

def main() -> Nat:
  keep_first(Nat, twice(3n), 9n)
```

```
6n
```

WHY MARK A PARAMETER ERASED

Without the minus sign, `A` is an ordinary parameter. The checker accepts both versions, but the compiled programs differ: the plain version receives the type as a value and drops it at the end. Write the program twice, once with `-A` and once with `A`, and compare the C each one emits (chapter 05 shows `-o file.c`):

```
bend erased.bend -o erased.c
bend plain.bend -o plain.c
grep -c _A_0 erased.c plain.c
```

```
erased.c:0
plain.c:2
```

The erased version has no variable for `A`. The plain version has one, `_A_0`, which it receives and then drops. The minus sign removes that cost.

WHAT A RUN-TIME USE IS

A binder is used at run time when the compiled body needs its value: it returns the value or computes with it. An erased binder cannot be used that way, because its only uses are in types and proofs:

```
import Base

def bad(-A: Type, x: A) -> A:
  A

def main() -> Nat:
  bad(Nat, 6n)
```

```
- expected : A
- observed : Type
```

TYPE ARGUMENTS ARE EXPLICIT

Bend infers almost nothing, so a type argument is passed like any other argument. Leaving it out is rejected:

```
import Base

def identity(-A: Type, x: A) -> A:
  x

def main() -> U32:
  identity(42)
```

```
- expected : Type
- observed : U32
```

With the type passed, the call checks. The type is erased, so it costs nothing at run time:

```
import Base

def identity(-A: Type, x: A) -> A:
  x

def main() -> U32:
  identity(U32, 42)
```

```
42
```

KINDS: HOW OFTEN A VALUE MAY BE USED

Every type has a kind that caps its uses. `Type` is `Kind(&1)` (affine) and `Data` is `Kind(&2)` (reusable). Base declares `List<a, A>` so that a list is exactly as reusable as its elements. A function can take the kind as an argument:

```
import Base

def count(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{head, tail}:
      1n+count(a, A, tail)

def main() -> Nat:
  count(&2, U32, [10, 20, 30])
```

```
3n
```

A parameter of type `+List<U32>` is a list whose elements are reusable. The type is `List<&2, U32>`, so the `+` on the parameter lets the function use the whole list twice:

```
import Base

def twice_len(+xs: +List<U32>) -> Nat:
  Nat.add(List.length(&2, U32, xs), List.length(&2, U32, xs))

def main() -> Nat:
  twice_len([1, 2, 3])
```

```
6n
```

ARRAYS: ONE OWNER, UPDATED IN PLACE

An array has exactly one owner. Writing to it changes the storage in place, and the array comes back to you. `[0 : U32*8n]` creates eight slots (the count must be a power of two). A read returns the array together with the element:

```
import Base

def value(result: Array<U32> & U32) -> U32:
  match result:
    case (array, item):
      item

def main() -> U32:
  a = [0 : U32*8n]
  a[5] <- 42
  value(a[5])
```

```
42
```

The bracket form is an array literal, and its type part says what the slots hold. `[fill : Type*slots]` asks for `slots` slots, which must be a power of two. `[fill : Type^depth]` names a depth, so `U32^3n` is 2^3 = 8 slots. The function form is `Array.new(-T: Data, +d: Nat, +v: T) -> Array<T>` in Base. `Array.new(U32, 3n, 0)` builds the same array as `[0 : U32^3n]`, because its second argument is the depth. Indexes wrap, so on an eight-slot array `a[9]` is slot 1. A slot count that is not a power of two is rejected:

```
import Base

def main() -> U32:
  a = [0 : U32*6n]
  a[5] <- 42
  0
```

```
- message  : a power of two count (^d takes a depth)
```

An array passed to a function is updated there and returned. Put the write in a def that takes the array as a parameter:

```
import Base

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item

def set_first(a: Array<U32>) -> Array<U32>:
  a[0] <- 7
  a

def main() -> U32:
  a = [1 : U32*4n]
  b = set_first(a)
  first_of(b[0])
```

```
7
```

A second independent array needs `Array.clone`, which copies every slot. Its result is a pair, and the pair cannot be destructured in `main`: a `match` may not inspect a computed value. Pass it to a helper that matches its parameter:

```
import Base

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item

def set_first(a: Array<U32>) -> Array<U32>:
  a[0] <- 7
  a

def use_clone(p: Array<U32> & Array<U32>) -> U32:
  match p:
    case (a1, a2):
      b = set_first(a1)
      x = first_of(b[0])
      y = first_of(a2[0])
      (x + y : U32)

def main() -> U32:
  a = [1 : U32*4n]
  use_clone(Array.clone(U32, a))
```

```
8
```

Here `x` is 7 and `y` is 1, so the copies stayed independent. Destructuring the result in `main` is rejected:

```
import Base

def main() -> U32:
  a = [1 : U32*4n]
  (a1, a2) = Array.clone(U32, a)
  first_of(a1[0])

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item
```

```
- message  : a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)
```

MAPS

A `Map` associates string keys with values. `Map.new` takes the kind and the value type. `Map.set` and `Map.get` return the map with their result, so the map keeps its owner. `Map.get` takes its default before the map:

```
import Base

def value(result: Map<&2, U32> & U32) -> U32:
  match result:
    case (map, item):
      item

def main() -> U32:
  m = Map.new(&2, U32)
  m = Map.set(&2, U32, m, "answer", 42)
  value(Map.get(U32, 0, m, "answer"))
```

```
42
```

A `Set` is a map with no values. `Set.add` returns the set, and `Set.size` counts its keys:

```
import Base

def main() -> Nat:
  s = Set.new()
  s = Set.add(s, "x")
  s = Set.add(s, "y")
  Set.size(s)
```

```
2n
```

TEMPLATES: COMPILE-TIME ARGUMENTS

A template parameter starts with `~`. Its argument is substituted at compile time and is not passed at run time. Each distinct template argument produces its own compiled copy, so a template may be called as often as you like:

```
import Base

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

```
42
```

A template may call itself and pass its template argument along. Two different steps give two compiled copies of the same loop:

```
import Base

def run(~step: U32 -> U32, n: Nat, x: U32) -> U32:
  match n:
    case 0n:
      x
    case 1n+p:
      run(~step, p, step(x))

def main() -> U32 & U32:
  (run(~(x => (x + 1 : U32)), 3n, 0), run(~(x => (x + 2 : U32)), 3n, 0))
```

```
(3, 6)
```

A template argument must be closed: it may mention top-level definitions, but not a local variable of the caller. Here `k` is a local variable, and the checker rejects the call:

```
import Base

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def add_k(k: U32, x: U32) -> U32:
  twice(~(y => (y + k : U32)), x)

def main() -> U32:
  add_k(1, 40)
```

```
- expected : a template applied to closed ~ arguments (k is a variable here, not comptime: pass it at run time)
- observed : twice
```

WHY A TEMPLATE, NOT A CLOSURE

A parameter of type `U32 -> U32` is a closure, and a closure is affine. `f(f(x))` uses `f` twice, so the checker rejects this program:

```
import Base

def apply_twice(f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  apply_twice(x => (x + 1 : U32), 40)
```

```
- expected : f
- observed : f (consumed more than once)
```

A `~f` argument is substituted at compile time, so `f` is code rather than a value that can be used up. The `twice` above is that fix.

CLOSURES

A function may return a closure that captures a value. A closure is affine: it may be called once, even when everything it captures is `Data`:

```
import Base

def offset(k: U32) -> U32 -> U32:
  x => (x + k : U32)

def main() -> U32:
  add2 = offset(2)
  add5 = offset(5)
  add5(add2(35))
```

```
42
```

A top-level definition may be called many times. Use a template when you need a function called repeatedly.

FOLDS

`List.foldl` reduces a list. Its signature takes the quantity of the elements, their kind, the accumulator type, and the combining function as template arguments, then the list and the starting value. Here `add` is a top-level definition, so it costs nothing to pass:

```
import Base

def add(a: U32, b: U32) -> U32:
  (a + b : U32)

def main() -> U32:
  List.foldl(~&1, ~U32, ~U32, ~add, [10, 20, 12], 0)
```

```
42
```

`List.filter` and `List.sort` take the element type and a function as template arguments, and they return `List<&2, A>`, the reusable form. This `filter` keeps the elements greater than 5:

```
import Base

def is_big(x: U32) -> Bool:
  U32.is_gt(x, 5)

def main() -> List<&2, U32>:
  List.filter(~U32, ~is_big, [1, 6, 3, 8])
```

```
[6, 8]
```

WHAT A COPY COSTS

Operation	What happens	Cost
`+x` on a `Data` value	both uses share one value; a reference count tracks the copies	a count update per shared use
`Array.clone(T, a)`	every slot is copied into a second array	proportional to the array's size
`Array.fork`	two handles share one array; reads and writes race through atomics	cheap, but marked `@unsafe`, so no law may depend on it
a template argument	a separate compiled copy for each distinct argument	code size, no run-time cost
a closure	affine: called once	no sharing to pay for

The guide says `+` pays a reference count. The emitted C keeps that count in 24 bits (`RFC_CNT`). Chapter 05 shows how to find the counts. Chapter 10 explains why none of this gives you copy-on-write forks of running code.

EXERCISES

1. Cube. The parameter appears three times in `x * x * x`. Add the qualifier that makes it reusable, then check that `cube(3)` prints `27`.

```
import Base

def cube(+x: U32) -> U32:
  (x * x * x : U32)

def main() -> U32:
  cube(3)
```

```
27
```

2. Keep the original. Clone the array, change the first slot of one copy to 7, and return the first slot of the other copy. The answer is `1`.

```
import Base

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item

def set_first(a: Array<U32>) -> Array<U32>:
  a[0] <- 7
  a

def other_copy(p: Array<U32> & Array<U32>) -> U32:
  match p:
    case (a1, a2):
      b = set_first(a1)
      first_of(a2[0])

def main() -> U32:
  a = [1 : U32*4n]
  other_copy(Array.clone(U32, a))
```

```
1
```

NEXT

Chapter 04 is the machine model: interaction nets, the rules that rewrite them, and why the model is the reason copying is explicit in Bend.

A.5 · bend-docs/src/content/tutorial/04-interaction-nets.md

Bend 2 tutorial 04: interaction nets · `docs-only` · updated 2026-10-18 · canonical `/tutorial/04-interaction-nets/`

04: interaction nets

This chapter is about a computation model, not a Bend program. You will not run any code here. Its purpose is to explain three words you will see in Bend's documentation, *active pair*, *confluence*, and *duplication*, and why the designers started from this model.

Status: **docs-only**. The examples are pen-and-paper exercises. Chapter 05 shows what the released compiler does.

THE PROBLEM: COPYING IS THE WHOLE GAME

Lambda calculus has one operation that does the work: β-reduction, which substitutes an argument into a function body. A variable used twice in the body forces you to copy the argument, and if the argument is itself an expensive computation, the copy repeats the work. Tree-shaped rewriting has no way to share that work.

Interaction nets solve this by representing a term as a graph. A shared piece of the graph is one node with several wires leaving it, so it is computed once. The price is that copying becomes an explicit operation with its own rules.

CELLS, PORTS, AND NETS

A **cell** has a symbol (its name), exactly one **principal port**, and a fixed number of **auxiliary ports**. A **net** is a collection of cells whose ports are joined pairwise by wires. A port that has no wire is free, and the free ports form the net's *interface*.



Core insight. A net is a graph, and its rules rewrite the graph. Released Bend 2 does not build that graph. Its runtime runs definitions, as the BendRT block below shows.

Composed of: a net.

```
net
- cells: CON | DUP | ERA, each with one principal port
- auxiliary ports: CON 2, DUP 2, ERA 0
- wires: each joins two ports; a port with no wire is part of the interface
```

ACTIVE PAIRS AND RULES

Two cells joined by a wire between their principal ports form an **active pair**. Only an active pair can be reduced. A **rule** replaces an active pair with another net that has the same free ports. The rule is local: it reads two cells and rewrites them, and nothing else.

Each unordered pair of symbols has at most one rule. A net with k symbols therefore has at most $k(k+1)/2$ rules. A net is in **normal form** when no active pair remains.

Two active pairs never share a cell. A cell has one principal port, so it can be part of at most one active pair. This is the property that makes local, parallel reduction safe.

CONFLUENCE: ORDER DOES NOT MATTER

Take two distinct active pairs in one net. Reducing either one leaves the other still reducible, and reducing both in either order produces the same net. This is **one-step confluence**, the diamond property:

```
        N
       /  \
  reduce a  reduce b
       \  /
        M
```

The bend2.dev page on interaction nets states a stronger property for the same systems: if a net has a normal form, reduction order does not change that result, or the number of interaction steps needed to reach it. The same page is clear about what this does not give you. It does not guarantee that every program terminates, that every implementation of the rules is correct, or that the implementation is free of synchronization, allocation, and scheduling costs.

Read those two claims as the model's claims. Chapter 05 shows what a real runtime must still pay for.

THE SYMMETRIC INTERACTION COMBINATORS

Lafont (1997) showed that three symbols are enough. The system is *symmetric* because its two binary symbols have the same shape, one principal and two auxiliary ports, so no special primitive is needed for lambda or copy. The bend2.dev page uses the Greek names γ, δ, ε; this chapter uses the names common in later literature:

Symbol	Name	Auxiliary ports
CON	constructor (γ)	2
DUP	duplicator (δ)	2
ERA	eraser (ε)	0

There are six rules, one for each unordered pair of symbols:

1. **CON–CON annihilates.** The two cells disappear, and their auxiliary ports are reconnected pairwise: first with first, second with second.
2. **DUP–DUP annihilates.** The same crosswise reconnection as rule 1.
3. **ERA–ERA annihilates.** Both cells disappear and nothing replaces them.
4. **CON–DUP commutes.** Four new cells appear: two DUP copies attached to the CON's auxiliary wires, and two CON copies attached to the DUP's auxiliary wires. Each new CON is wired to each new DUP, so the four new cells form a 2×2 crossing. The number of cells goes from two to four.
5. **CON–ERA erases.** The CON cell disappears, and two new ERA cells are attached to its two auxiliary wires.
6. **DUP–ERA erases.** The same as rule 5, for a DUP cell.

Rule 4 is the one that copies. A DUP that meets a CON is pushed through it, so each of the CON's two results gets its own DUP. Repeated application copies a structure one layer at a time, and each step is one local interaction.

Lafont proved that these six rules are universal: a translation maps any other interaction system into them and preserves its computation and its parallelism. The bend2.dev page describes this as "universal", and it does not claim that the six rules alone make programs fast.

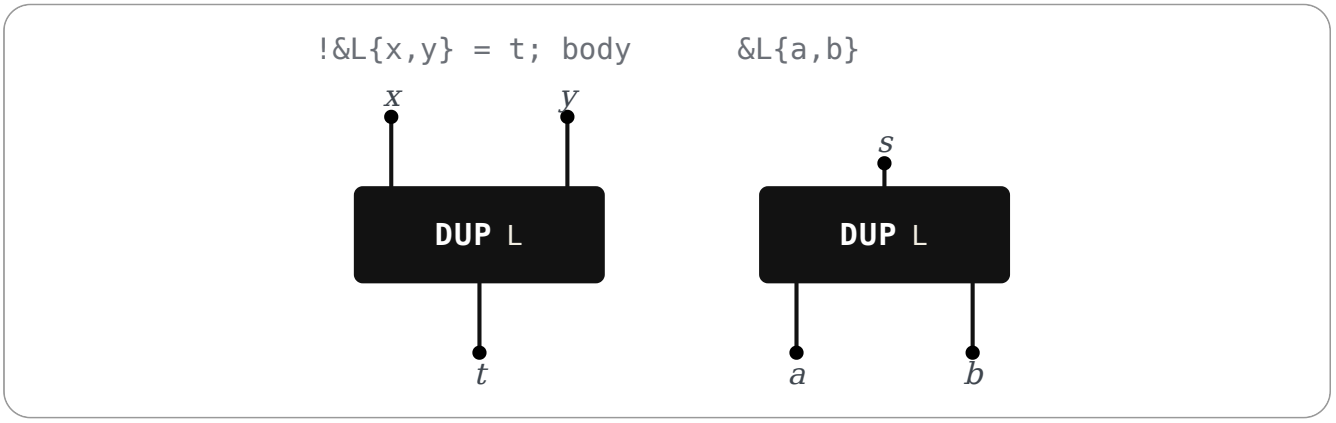
LABELS: THE INTERACTION CALCULUS

The three combinators above have one kind of DUP. Victor Taelin's Interaction Calculus (IC) keeps the same three kinds of cell, gives every DUP a **label** (a number), and adds a text notation for nets. The text is not a drawing: `!&L{x,y} = t; body` names cells and wires, the way `λx.t` names a CON. Read the text as cells:

Text	Cell	Ports
`λx.t` and `(f a)`	CON	for `λx.t`, the output is principal and the aux ports hold the variable and the body; for `(f a)`, the function is principal and the aux ports hold the argument and the result
`!&L{x,y} = t; body`	DUP, label `L`	principal: `t`; aux: `x` and `y`; `body` is the rest of the net
`&L{a,b}`	DUP, label `L`, read from the other side	principal: the value; aux: the alternatives `a` and `b`
`*`	ERA	principal only

Give every DUP one label, and the rule that two DUPs annihilate (rule 2 above) is the case where their labels match. Different labels commute instead. So the symmetric combinators are the special case with a single label.

So a superposition is not a new kind of cell. `&L{a,b}` is a DUP labelled `L`, read from the other side: its principal port faces the value, and its aux ports hold the alternatives. Both readings are drawn side by side:



Chapter 14 runs these rules on the reference implementation, prints the interaction counts, and shows where a superposition is born.

B-REDUCTION AS ONE INTERACTION

Interaction nets encode a lambda abstraction and an application as two CON cells:

- **λx. M** is a CON whose principal port is the lambda's output, whose first auxiliary port is the variable x (a wire to every use of x), and whose second auxiliary port is the body M.
- **(F A)** is a CON whose principal port connects to the function F, whose first auxiliary port carries the argument A, and whose second auxiliary port carries the result.

When a lambda meets an application through their principal ports, that is an active pair of two CONs, and rule 1 applies. The first auxiliary wires join: the variable's wire now connects to the argument A. The second auxiliary wires join: the body M now connects to the result. β-reduction is one interaction.

If the variable is used twice, its wire must be duplicated, and each copy is a separate CON–DUP commutation. That is the duplication cost. The model does not hide it, and Chapter 04's exercises count it.

WHY THIS MATTERS FOR BEND, AND WHAT IT DOES NOT

The bend2.dev notes describe interaction nets as the model behind Bend's earlier HVM runtimes. Released Bend 2 does not run them. Its compiler emits a flat C evaluator, called BendRT, that keeps pending work explicitly and does not reduce graphs. The notes list what BendRT does instead:

- **Affine ownership.** Each ordinary value has at most one live use, so the compiler knows when memory can be freed. No tracing garbage collector is needed.
- **Counted sharing.** A `+` value keeps a reference count. A value used through `+` is shared by count, not by graph.
- **Balanced forks.** `a b = f(x) g(y)` hands each call to a core once. The programmer promises the two calls take about the same time.

> **Core insight.** BendRT is a flat evaluator of definitions. Its primitives are words, definitions, jumps and tasks, not cells and rewrites.

Composed of: BendRT, the released runtime.

```
BendRT
- term: one 64-bit word; a small value is stored inline, anything else is a pointer into one heap shared by every core and the GPU
- definition: a segment of a flat state machine; a call is a jump
- parallel call: one join task, plus one task per call, dealt to CPU or GPU lanes
- memory: no garbage collector; a match frees the node it opens; only `+` values carry a reference count, which the emitted C keeps in 24 bits
```

The guide's Under the Hood section gives the first three parts and the memory rule. The 24-bit width comes from the emitted C.

So the model explains *why* Bend is affine and why copying is spelled out. It does not predict how fast a program runs. The notes quote the Asperti and Mairson result that the implementation cost of optimal reduction cannot be bounded by an elementary function of the number of reduction steps. Counting interactions is not the same as measuring time.

SUMMARY OF PROPERTIES

Property	What it says	Where the page states it
Locality	a rule reads and rewrites only its two cells	interaction-nets page
One-step confluence	two distinct active pairs commute	interaction-nets page, citing Lafont 1990
Step-count invariance	any order to a normal form takes the same number of steps	interaction-nets page
No termination guarantee	a net can reduce forever	interaction-nets page
Universality	the six combinator rules can encode other systems	interaction-nets page, citing Lafont 1997
Cost is not steps	allocation, scheduling, and memory still cost time	bendrt-explained and interaction-nets pages

EXERCISES (PEN AND PAPER)

1. Erase a cell. A CON has its principal port wired to an ERA. Its two auxiliary ports are wired to two free ports, `p` and `q`. How many interactions happen, and what is left?

Solution. One interaction: rule 5. The CON disappears and two new ERA cells attach to `p` and `q`. Before the step there were two cells (CON, ERA). After it there are two cells (ERA, ERA). No active pair remains, so the net is in normal form.

2. Count the cells of a commutation. A CON and a DUP are joined through their principal ports. Their four auxiliary ports are wired to four free ports. How many cells and how many internal wires exist after rule 4?

Solution. Four cells: two DUP copies and two CON copies. The copies' principal ports take the four free wires, and their auxiliary ports form four internal wires, one for each (CON copy, DUP copy) pair. The number of cells rises from two to four.

3. Two active pairs. Net N has two active pairs, A and B, that share no cell. Show that reducing A then B gives the same net as reducing B then A.

Solution. Rule A rewrites only the cells of A and creates wires only to A's free ports, which are cells or interface ports that B does not touch. The same holds for B. Both orders therefore produce the same set of cells and wires, and the diamond closes.

4. Beta in one step. Encode `(λx. x) y` with two CONs, where `y` is a free port. Reduce it.

Solution. The lambda CON's first auxiliary wire (the variable x) meets the application's first auxiliary wire (the argument y), and the lambda's body wire meets the result wire. The body of the lambda is `x`, which is the variable's wire, so the result is directly connected to `y`. One interaction; no DUP was needed because `x` is used once.

NEXT

Chapter 05 shows what the released compiler does for a parallel call, and what a balanced fork looks like in code.

A.6 · bend-docs/src/content/tutorial/05-parallelism-and-runtime.md

Bend 2 tutorial 05: parallelism and the runtime · `runnable` · updated 2026-10-18 · canonical `/tutorial/05-parallelism-and-runtime/`

05: parallelism and the runtime

Bend runs independent recursive calls in parallel without threads, locks, or annotations. You write a *parallel call*, and the compiler makes it a fork. Two promises come with it. The calls must be independent, which purity and affine ownership guarantee. They must also take about the same time, which is your job. This chapter shows the shape that meets the second promise, and how to check the result honestly.

THE PARALLEL CALL

```
import Base

def pow2(+d: Nat) -> U32:
  match d:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)

def main() -> U32:
  pow2(12n)
```

```
4096
```

`a b = pow2(p) pow2(p)` is one line with two calls. Both start together, and the line continues when both have finished. `+d` lets both calls use the depth. The checker runs this program and prints 4096. A native binary runs the same calls on several cores. Compile it with `bend pow2.bend -o pow2` and run `./pow2 --threads 8`.

Bang runs the calls on a GPU. `pow2!(12n)` sends that call and every parallel call it makes to the GPU. The lab machine for this tutorial has no GPU device (there is no `/dev/dri` and no NVIDIA device), so the bang form was not timed here. On a machine with no usable GPU, `--gpu off` runs bang calls on the CPU threads.

HOW A CALL RUNS

A top-level definition is code, not a value, so calling it does not use it up. The checker counts uses of values: a closure may be called once, and a `+` value costs a reference count when it is shared (chapter 03). A definition may call itself as often as its argument shrinks (chapter 02).

The guide describes the run-time shape of those calls:

- A tail call compiles to a loop. `leaf` below calls itself in tail position, so each leaf becomes one loop.
- Bend has no C stack. Each definition compiles to a segment of a flat state machine, and a call is a jump.

THE FORK SHAPE: A TREE OF LEAVES

A parallel call that splits one call into two equal halves is a *fork*. Forking at every level down to single numbers costs more in scheduling than the work itself. The shape that works has two parts:

- **Tree.** Above a fixed depth, split the range in two equal halves with a parallel call.
- **Leaf.** Below the tree, a tail loop walks 8 to 64 consecutive items with scalar accumulators. A loop of this kind compiles to one native loop.

This program sums the numbers from 0 to 511 with eight leaves of 64 items each:

```
import Base

def leaf(+n: Nat, +i: U32, acc: U32) -> U32:
  match n:
    case 0n:
      acc
    case 1n+p:
      leaf(p, (i + 1 : U32), (acc + i : U32))

def tree(+d: Nat, +i: U32) -> U32:
  match d:
    case 0n:
      leaf(64n, i, 0)
    case 1n+p:
      a b = tree(p, i) tree(p, (i + (64 << p) : U32))
      (a + b : U32)

def main() -> U32:
  tree(3n, 0)
```

```
130816
```

The check is arithmetic: the numbers 0 to 511 sum to 511 × 512 / 2 = 130816. Two details carry the design:

- The second half starts at `i + (64 << p)`. Each leaf covers 64 indices, so a tree of depth `p` covers `64 << p` indices, and the second half begins right after the first. The general formula is `i + (L << p)`, where `L` is the leaf size. With leaves of one item it is `i + (1 << p)`, which is why that form appears in other examples. With leaves of 64 it must use 64.
- The tree's halves are equal in work, because every leaf does the same 64 steps. A fork over unequal halves would leave one core idle.

Use a `tree` for data a fork splits, and a `List` for data a leaf walks once. Head-and-tail recursion over a list runs on one core, so use it only inside a leaf. Use an `Array` for data you index or update in place (chapter 03).

EXERCISES

1. Four levels. Change the depth to `4n`. How many numbers are summed, and what is the output? Check your answer by running the program.

Solution. The tree covers 64 × 2^4 = 1024 numbers, so the sum is 1023 × 1024 / 2 = 523776.

```
import Base

def leaf(+n: Nat, +i: U32, acc: U32) -> U32:
  match n:
    case 0n:
      acc
    case 1n+p:
      leaf(p, (i + 1 : U32), (acc + i : U32))

def tree(+d: Nat, +i: U32) -> U32:
  match d:
    case 0n:
      leaf(64n, i, 0)
    case 1n+p:
      a b = tree(p, i) tree(p, (i + (64 << p) : U32))
      (a + b : U32)

def main() -> U32:
  tree(4n, 0)
```

```
523776
```

2. Break the midpoint. Replace `64 << p` with `1 << p` in the program above and run it. Explain the output before you run it.

Solution. The halves overlap. The second half starts only one index after the first, so most indices are summed several times and the total is wrong. Expect the program to print 17920, and that the number is not 130816.

```
import Base

def leaf(+n: Nat, +i: U32, acc: U32) -> U32:
  match n:
    case 0n:
      acc
    case 1n+p:
      leaf(p, (i + 1 : U32), (acc + i : U32))

def tree(+d: Nat, +i: U32) -> U32:
  match d:
    case 0n:
      leaf(64n, i, 0)
    case 1n+p:
      a b = tree(p, i) tree(p, (i + (1 << p) : U32))
      (a + b : U32)

def main() -> U32:
  tree(3n, 0)
```

```
17920
```

Each leaf still sums 64 consecutive values from `i`. With the midpoint `i + (1 << p)`, the second half of a level-one tree starts one index after the first, so every leaf overlaps its neighbour by 63 values. The total counts many values several times, and 17920 is not the sum 0 to 511. The check is simple: whenever a parallel sum disagrees with the sequential sum, recompute the midpoints first.

BUILD A NATIVE BINARY AND RUN IT

The checker runs a pure `main` slowly, so measure with a native build:

```
bend main.bend -o main
./main --threads 1
./main --threads 8
```

Both runs must print the same output. If they do not, the program has a bug, not a scheduling problem. The speedup is the one-thread time divided by the multi-thread time, measured on the same input, with checks that the outputs match.

MEASURE HONESTLY

A speedup claim needs five things:

1. The same input for every run, and the same output checked against the one-thread result.
2. Several runs per mode, reported as a median, with the order rotated between modes.
3. The machine, thread count, GPU setting, and compiler version written down with the numbers.
4. A named baseline. "Twelve times faster" means nothing without the two times.
5. Build time separate from run time. Compile first, then time only the binary.

Small inputs mislead: for work this small, process startup dominates the time. Measure with inputs big enough that the run takes long enough to time reliably, and state how long that is.

CHOOSING THE THREAD COUNT

Bend has no automatic setting. `--threads` takes a number from 1 to 128, and without the flag a binary uses the CPU count, which `./main --help` prints. `--threads auto` is rejected:

```
$ ./main --threads auto
bend: expected a thread count of 1 or more after --threads
```

Definition. The thread count is the `--threads N` setting of a native binary: the number of CPU threads it runs on, from 1 to 128. Without the flag it is the CPU count.

Core insight. The fastest count belongs to one program, one input, and one machine. The number of leaves does not give it. `tree(d)` has 2^d leaves, far more than any thread count, and the scheduler hands each task to a core once and never moves it (the guide). Measure.

A script can do the measuring. It runs each count after a warm-up, requires every run to print the output of the one-thread run, and reports the median and the range:

```
#!/usr/bin/env bash
# sweep.sh PROGRAM [THREADS...]
# Runs PROGRAM at each thread count after one warm-up run. Every timed run must
# print the same output as the one-thread run, or the sweep stops. Prints the
# median and range of RUNS runs per count, then the fastest count.
set -euo pipefail
prog=$1
shift
[ "$#" -gt 0 ] || set -- 1 2 4 8 12
runs=5
reference=$("$prog" --threads 1)
best_t=""
best_ms=""
for t in "$@"; do
  "$prog" --threads "$t" >/dev/null
  times=""
  for _ in $(seq "$runs"); do
    start=$(date +%s%N)
    out=$("$prog" --threads "$t")
    end=$(date +%s%N)
    if [ "$out" != "$reference" ]; then
      echo "threads=$t printed a different result; stopping" >&2
      exit 1
    fi
    times="$times $(( (end - start) / 1000000 ))"
  done
  sorted=$(printf '%s\n' $times | sort -n)
  median=$(printf '%s\n' "$sorted" | sed -n "$(( (runs + 1) / 2 ))p")
  low=$(printf '%s\n' "$sorted" | head -1)
  high=$(printf '%s\n' "$sorted" | tail -1)
  echo "threads=$t median=${median} ms (range ${low}-${high} ms)"
  if [ -z "$best_ms" ] || [ "$median" -lt "$best_ms" ]; then
    best_t=$t
    best_ms=$median
  fi
done
echo "fastest: --threads $best_t (${best_ms} ms)"
```

Save it as `sweep.sh` and run `bash sweep.sh ./main 1 2 4 8 12 16 24`, with `main` set to `tree(26n, 0)`. On the lab machine (12 CPUs) it printed:

```
threads=1 median=366 ms (range 343-372 ms)
threads=2 median=214 ms (range 208-237 ms)
threads=4 median=150 ms (range 143-151 ms)
threads=8 median=108 ms (range 102-118 ms)
threads=12 median=112 ms (range 107-115 ms)
threads=16 median=108 ms (range 100-113 ms)
threads=24 median=101 ms (range 98-112 ms)
fastest: --threads 24 (101 ms)
```

Three things stand out:

- Most of the gain comes by 8 threads: 366 ms falls to 108 ms.
- The default of 12 was not the fastest. Sixteen and 24 were a few milliseconds faster, but their ranges overlap with the range of 12.
- On `tree(22n, 0)`, one thread took 22 ms, and each of 4, 8, 12, 16, and 24 threads took 14 to 17 ms. Those differences are about the size of the run-to-run spread, so the sweep cannot choose a count there. Use an input large enough to time, as the section Measure honestly says.

The script has limits. It times wall-clock time on the machine it runs on, so other load changes the numbers. It needs bash and GNU `date`. Its answer holds for this program, this input, and this machine. The guide says the GPU shines on uniform numeric work, such as mandelbrot or nbody, and that divergent work, such as n-queens, stays faster on the CPU. Measure those too, with `!` and without it.

COUNT WHAT THE COMPILER EMITS

`bend main.bend -o main.c` writes the C source. Three names mark runtime work that copies or releases a shared value: `term_keep`, `ctr_take`, and `rfc_seal`. A reference count is a `term_keep`. A program that uses `+` on values pays for them, and a program that does not should have none in its own functions.

For the `pow2` program above, the compiler emitted `term_keep` 2 times, `ctr_take` 3 times, `rfc_seal` 2 times, and `term_peek` 6 times. A program with no sharing at all, `def main() -> U32: 1`, emits exactly the same counts. So those occurrences come from the runtime, and `pow2` adds none of its own. `term_peek` is a borrow, which costs no count. When

you audit a program, compare its counts with that baseline, and treat any new keep in your own code as a regression to remove or explain.

NEXT

Chapter 06 covers duplication and superposition, the terms behind labels and search. It also says plainly what the released compiler does not have.

A.7 · `bend-docs/src/content/tutorial/06-duplication-superposition.md`

Bend 2 tutorial 06: duplication, superposition, and SupGen · `docs-only` · updated 2026-10-18 · canonical `/tutorial/06-duplication-superposition/`

06: duplication, superposition, and SupGen

Your question mentions three things: copy-on-write with `sup` and `dup`, SupGen, and forks of running code. This chapter explains what the words mean in the lineage behind Bend, then shows what the released compiler accepts. Short version: the released Bend 2 has no `sup` syntax, no SupGen in its compiler or Base library, and no copy-on-write forks of running code. The rest of this chapter gives the evidence.

DUPLICATION AND SUPERPOSITION, IN THE CALCULUS

The Interaction Calculus, which HVM4 implements, has two kinds of term for sharing:

- A **duplication** `dup` takes one value and gives two copies, linked by a label. It is the term-level form of the DUP cell in chapter 04.
- A **superposition** `sup` holds two alternatives at once, also linked by a label. It is the term-level form of a choice that exists in several places.

The bend2.dev HVM4 page gives a worked example in its own syntax, where `&{1, 2}` is a superposition of two numbers. Adding 10 to it gives the two results 11 and 12. The page says the evaluator performs an addition for each alternative, and that duplication "still consumes work and memory". Sharing saves work only when the alternatives share their computation.

Labels decide how two terms meet. In the HVM literature, a `dup` and a `sup` with the same label route their parts to each other, and with different labels they commute, creating new `dup` and `sup` terms. Chapter 14 runs these rules on Victor Taelin's Interaction-Calculus reference implementation, which is a separate implementation from HVM4. HVM4 itself was not run.

Core insight. `dup` and `sup` are two terms of one calculus. Their syntax differs, and they meet only through a label. Equal labels route, and different labels commute.

Composed of: a sharing term with label `L`.

```
sharing term, label L
- dup: !&L{x, y} = t; body     one value t, two uses x and y
- sup: &L{a, b}                two alternatives a and b, held in one place
```

Chapter 14 runs both rules. In its examples, equal labels take 2 interactions and different labels take 9.

SEARCH BY SUPERPOSITION

The bend2.dev note *Search by superposition* describes a technique built on this calculus. Apply a function to a superposition of its whole input domain, and the evaluator shares work across all inputs at once. The note reports results for a SAT instance and for program search. Those experiments used earlier HVM syntax. The page marks them **documentation only**: the numbers were reported by their author and were not re-run. Do not cite them as Bend 2 measurements.

WHAT THE RELEASED COMPILER ACCEPTS

These two programs are rejected by `bend 2.0.25`, and the messages are the real ones:

```
import Base

def main() -> U32:
  x = &{1, 2}
  (x + 10 : U32)
```

```
- expected : a name
- observed : '{'
```

`if` is not a keyword either. Bend branches with `match`, as chapter 01 shows:

```
import Base

def main() -> U32:
  if True{}:
    1
  else:
    2
```

```
- expected : '='
- observed : ':'
```

Adding a label does not help. The labelled form is rejected as well:

```
import Base

def main() -> U32:
  x = &0{1, 2}
  (x + 10 : U32)
```

```
- expected : ':'
- observed : ','
```

The released guide, Base, and the compiler say nothing about SupGen, and the bend2.dev pages document none. The site's own release-status note lists "Bend2–SupGen integration" among the launch blockers, and its HVM runtime note calls SupGen integration "still private". Treat SupGen as **not available** in the released compiler. Chapter 10 explains what you can do in its place.

WHAT BEND 2 HAS INSTEAD OF COPY-ON-WRITE

The released language has copying operations, but none of them is a copy-on-write fork of a running program:

- **`+` on a `Data` value** shares the value and counts the sharing. Nothing is copied, and the count is kept in 24 bits.
- **`Array.clone`** copies every slot into a second array. The cost grows with the array's size.
- **`Array.fork`** gives two handles to one array in constant time. Both handles read and write the same cells, so racing writes go through atomic operations. It is marked `@unsafe` in Base, which means the checker does not verify it, and no law may depend on it.
- **Templates** make one compiled copy for each distinct template argument. The copy happens at compile time, not while the program runs.

None of these forks a live program. A running Bend program is a value that the compiler has already compiled, and nothing in the language rewrites it.

EXERCISES

1. Read the error. Explain, in one sentence each, why `&{1, 2}` and `if` are rejected.

Solution. `&{` begins a superposition in HVM syntax, which the released parser does not accept, so it expects a name where it sees the brace. `if` is not a keyword, so the parser reads `if True{}` as the start of a binding and expects `=` where it finds `:`.

2. Two alternatives without superposition. Replace the idea of a superposed price with an ordinary choice. Write a `Choice` type with two constructors and a function that prices each one. Print both prices as a pair.

Solution. The alternatives are values, so they can be passed around, stored, and checked by laws. The output below is the pair of both prices.

```
import Base

type Choice is Data:
  Low{}
  High{}

def price(c: Choice) -> Nat:
  match c:
    case Low{}:
      10n
    case High{}:
      20n

def main() -> Nat & Nat:
  (price(Low{}), price(High{}))
```

```
(10n, 20n)
```

3. Why no law can use `Array.fork`. Explain what the `@unsafe` marker in Base changes for a law that depends on `Array.fork`.

Solution. The checker treats an `@unsafe` definition as outside its proof guarantees. A law whose proof depends on it is rejected in the proof check, so the law would be unverified. Chapter 11 shows the exact message.

NEXT

Chapter 07 covers laws and proofs, the part of Bend that checks properties of the values a program computes.

A.8 · bend-docs/src/content/tutorial/07-laws-and-proofs.md

Bend 2 tutorial 07: laws and proofs · `runnable` · updated 2026-10-18 · canonical `/tutorial/07-laws-and-proofs/`

07: laws and proofs

A **law** is a claim about a program, written as a type. Bend has no tactics and no proof search, so a proof is an ordinary definition with the same name as the law. The compiler checks that the definition has the claimed type. If you know Haskell, think of a QuickCheck property that the compiler must check for every input, not just sampled ones. If you know Lean, the table below maps the ideas, with one big difference: Bend has no tactics and no proof search, so every step is a term you write.

Core insight. A law is a type, and a proof is a program of that type. This is the Curry-Howard idea: a claim is a type, and a term that inhabits it is its proof. `bend PROOF.bend` accepts a proof only if its def has the law's type, its recursive calls terminate, and each `{==}` holds because both sides compute to the same term.

Composed of: a law and its proof.

```
law add_zero                    the type: a claim
- for x: Nat                    a binder
- {Nat.add(x, 0n) == x : Nat}   the claim

def add_zero(x)                 the program, with the law's name
- one case per constructor of x
- each case: {==}, or a rewrite %e : P followed by {==}
```

Lean idea	Bend 2	Note
a theorem statement and its proof	`law name:` and a same-named `def name(...)`	the law is the claim; the def is the proof
`rfl`, closing a goal by computation	`{==}`	both sides must compute to the same term
`rw [h]`	`%h : P`	`P` is the goal, with `_` where the right-hand side goes
`induction n with`	`match n:` plus a recursive call	the recursive call is the induction hypothesis
`∃ w, P w`	`exs w: T` in the law	the proof returns `(w, proof)`
`sorry`	`?TODO`	the gate rejects any open hole
`simp`, `omega`, other tactics	nothing	write the rewrite or the case split yourself

YOUR FIRST LAW: INDUCTION

This law says that adding zero to any natural number does nothing:

```
import Base

law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> Nat:
  Nat.add(42n, 0n)
```

```
42n
```

Read the file in three parts.

- `law add_zero:` names the claim. `for x: Nat` quantifies over every natural. The equation `{Nat.add(x, 0n) == x : Nat}` is the claim itself: the two sides are equal at type `Nat`.
- `def add_zero(x):` is the proof. It has the same name as the law and one parameter for each `for`.
- `match x:` splits the proof into cases, and each branch refines the goal.

Look at the goal in each branch. In the case `0n`, the left side `Nat.add(0n, 0n)` computes to `0n`, which is the right side, so `{==}` closes the goal. In the case `1n+p`, the left side is `Nat.add(1n+p, 0n)`, and it computes to `1n+Nat.add(p, 0n)`, which is not yet `1n+p`. The recursive call `add_zero(p)` is the **induction hypothesis**: it proves the claim for the smaller number `p`. The line `%add_zero(p) : {...}` rewrites the goal with that equation. The `_` marks where the right-hand side of the equation goes. After the rewrite, both sides are the same, and `{==}` closes the goal.

The induction hypothesis is allowed only because `p` is smaller than `x`: it is a pattern-bound variable from `1n+p`, so the termination rule of chapter 02 holds.

EQUALITY BY COMPUTATION

When both sides compute to the same term, `{==}` alone is enough. The law below holds because `Nat.add` matches on its first argument, so `0n` on the left reduces without any induction:

```
import Base

law zero_left:
  for x: Nat
  {Nat.add(0n, x) == x : Nat}

def zero_left(x):
  {==}

def main() -> Nat:
  Nat.add(0n, 5n)
```

```
5n
```

The same reasoning fails when the computed argument is a variable. This law puts the variable where the computation needs it, and `{==}` cannot prove it:

```
import Base

law twice_is_double:
  for x: Nat
  {Nat.add(x, x) == Nat.double(x) : Nat}

def twice_is_double(x):
  {==}
```

```
- expected : Nat.add(x, x)
- observed : Nat.double(x)
```

Both sides are correct, but `{==}` needs them to reduce to the same term, and the variable `x` blocks the reduction. Split the variable with `match` and prove each case, or use induction as `add_zero` does.

BOOLEAN LAWS BY CASES

A law over a finite type needs one `{==}` per constructor. This law says negating twice gives back the original value:

```
import Base

law not_not:
  for b: Bool
  {Bool.not(Bool.not(b)) == b : Bool}

def not_not(b):
  match b:
    case True{}:
      {==}
    case False{}:
      {==}

def main() -> Bool:
  Bool.not(Bool.not(True{}))
```

```
True{}
```

EQUALITY AS A RELATION

Equality laws state that two computations give the same result. The bend2.dev example proves that cancelling a job twice has the same effect as cancelling it once, for every state. The job type and its cancel function live in another file:

job.bend

```
import Base

type State is Data:
  Queued{}
  Running{}
  Cancelled{}
  Finished{}

def cancel(state: State) -> State:
  match state:
    case Queued{}: Cancelled{}
    case Running{}: Cancelled{}
    case Cancelled{}: Cancelled{}
    case Finished{}: Finished{}
```

main.bend

```
import Base
import ./job.bend as Job

law cancel_idempotent:
  for state: Job.State
  {Job.cancel(Job.cancel(state)) == Job.cancel(state) : Job.State}

def cancel_idempotent(state):
  match state:
    case Job.Queued{}: {==}
    case Job.Running{}: {==}
    case Job.Cancelled{}: {==}
    case Job.Finished{}: {==}

def main() -> Job.State:
  Job.cancel(Job.cancel(Job.Running{}))
```

```
job.Cancelled{}
```

The law is checked as a type, so `main` does not need to call it. The output prints the constructor with its module prefix, `job.Cancelled{}`.

Two equality lemmas in Base help you rewrite equations: `Equal.sym` turns `e : {a == b : T}` into `{b == a : T}`, and `Equal.trans` chains two equations. The next example uses `Equal.sym`:

```
import Base

law flip:
  for +a: Nat
  for +b: Nat
  for e: {a == b : Nat}
  {b == a : Nat}

def flip(a, b, e):
  Equal.sym(Nat, a, b, e)
```

```
All terms check.
```

EXISTENTIAL LAWS

`exs y: T` in a law asks for a witness. The proof returns the witness together with the proof of the equation:

```
import Base

law has_zero:
  exs y: Nat
  {Nat.add(y, 0n) == 0n : Nat}

def has_zero():
  (0n, {==})
```

```
All terms check.
```

HOLES: UNFINISHED PROOFS

A `?TODO` in a proof leaves that branch open. The checker counts open holes, and a file with any open hole fails the proof check:

```
import Base

law cancel_all:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def cancel_all(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      ?TODO
```

```
Error: 1 TODO found.
The code is incomplete, and not a valid proof yet.
```

A named hole, `?name`, stops the check and prints the goal that the branch must prove. Here the goal is printed for the branch `1n+p`:

```
import Base

law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      ?inspect
```

```
- expected : {1n+Nat.add(p, 0n) == 1n+p : Nat}
- observed : ?inspect
```

Use `?name` to see the goal you must prove, then replace it with a proof. Leave `?TODO` only while you write the file. The proof gate rejects both.

LAWS AND PROOF: THE PROJECT LAYOUT

A project keeps its claims and its proofs apart. `LAWS.bend` imports the code and states the laws. `PROOF.bend` imports both and proves each law with a def of the same name, as `def Laws.name` for a law named `name`. The example below stores three files in one directory. `job.bend` is the code under test, and it is the same file as in the equality example. `LAWS.bend` states four laws about it:

job.bend

```
import Base

type State is Data:
  Queued{}
  Running{}
  Cancelled{}
  Finished{}

def cancel(state: State) -> State:
  match state:
    case Queued{}: Cancelled{}
    case Running{}: Cancelled{}
    case Cancelled{}: Cancelled{}
    case Finished{}: Finished{}
```

LAWS.bend

```
import Base
import ./job.bend as Job

law cancel_idempotent:
  for state: Job.State
  {Job.cancel(Job.cancel(state)) == Job.cancel(state) : Job.State}

law cancels_queued:
  {Job.cancel(Job.Queued{}) == Job.Cancelled{} : Job.State}

law cancels_running:
  {Job.cancel(Job.Running{}) == Job.Cancelled{} : Job.State}

law preserves_finished:
  {Job.cancel(Job.Finished{}) == Job.Finished{} : Job.State}
```

PROOF.bend

```
import Base
import ./job.bend as Job
import ./LAWS.bend as Laws

def Laws.cancel_idempotent(state):
  match state:
    case Job.Queued{}: {==}
    case Job.Running{}: {==}
    case Job.Cancelled{}: {==}
    case Job.Finished{}: {==}

def Laws.cancels_queued():
  {==}

def Laws.cancels_running():
  {==}

def Laws.preserves_finished():
  {==}
```

```
All terms check.
```

`bend PROOF.bend` is the gate. It checks every law and every proof, and it prints `All terms check.` only when none is open. Run `bend LAWS.bend` alone and every law is an open claim. The message counts them as TODOs:

```
Error: 4 TODOs found.
```

Two rules follow from the layout:

- **An accepted law is never weakened to pass.** Changing a law so that its proof becomes easy defeats its purpose. If a proof is hard, the law may be correct and the proof is what needs work.
- **A law with no def is an open claim.** It stays open until a same-named def proves it. A file that states a law and leaves it unproved fails the gate.

Chapter 11 covers the gate's other failure mode: a proof that depends on an `@unsafe` definition. Version 2.0.25 prints a warning and still exits with status 0, so do not treat exit status alone as the gate. Bend 2.0.36 adds `bend PROOF.bend --verdict`, which rechecks the proofs with a separate kernel. The bend2.dev notes say that option needs Lean v4.34.0 exactly, or a prebuilt kernel named by the `BENDTT` variable. Version 2.0.25 has no such option, and no Lean toolchain was installed on the lab machine, so the gate in this tutorial is `bend PROOF.bend` plus a reading of its output.

EXERCISES

1. Prove a right identity. Prove `Nat.add(x, 0n) == x` without copying `add_zero`, by naming the induction hypothesis and rewriting with it.

Solution. This is the proof above. Put the hypothesis `add_zero(p)` in a rewrite whose motive is `1n+_`, then close with `{==}`.

2. Find the failing proof. The law `cancels_running` states that cancelling a running job yields `Cancelled{}`. Change the `Running{}` branch of `cancel` to return `Queued{}` and predict what the gate prints.

Solution. The gate fails, because `cancels_running` no longer holds for the new `cancel`. The law is the specification, and the implementation has to meet it.

NEXT

Chapter 08 covers types that depend on values, and why they make preconditions checkable.

A.9 · bend-docs/src/content/tutorial/08-dependent-types-and-totality.md

Bend 2 tutorial 08: dependent types and totality · `runnable` · updated 2026-10-18 · canonical `/tutorial/08-dependent-types-and-totality/`

08: dependent types and totality

A **dependent type** is a type that a function computes from a value. In Haskell you reach for GADTs or type families. In Bend every type is an ordinary function call, so the same machinery that checks values checks types. This chapter shows three uses: a type that is `Unit` for one input and `Empty` for another, a list whose length is part of its type, and the rule that keeps every such definition terminating.

Three kinds of type appear in this chapter:

Kind	Example	What it depends on
normal	`U32`	nothing
function	`U32 -> U32`	nothing: it is the type of functions from values to values
dependent	`IsZero(n)`, `Vec(A, n)`	a value `n`, and for `Vec` also a type `A`

A dependent type is a function that returns a type. `IsZero` takes a `Nat` and returns `Unit` or `Empty`, and that result limits what a caller may pass: for `0n` there is one value to pass, and for `1n+p` there is none. The two sections below build those dependent types.

A TYPE THAT COMPUTES

`IsZero(n)` is `Unit` when `n` is zero and `Empty` otherwise. `Empty` has no values, so a function that demands an `IsZero` argument can only be called with zero:

```
import Base

def IsZero(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      Empty

def require_zero(n: Nat, evidence: IsZero(n)) -> Nat:
  n

def main() -> Nat:
  require_zero(0n, Unit{})
```

```
0n
```

`Unit{}` is the only value of `Unit`, so the caller must supply it. Calling `require_zero(1n, Unit{})` is rejected, because `IsZero(1n)` is `Empty` and `Unit{}` is not an `Empty`:

```
import Base

def IsZero(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      Empty

def require_zero(n: Nat, evidence: IsZero(n)) -> Nat:
  n

def main() -> Nat:
  require_zero(1n, Unit{})
```

```
- expected : Empty
```

A function whose argument has type `Empty` can be written with an empty `match`. It needs no case, because no value can exist:

```
import Base

def from_empty(-A: Type, e: Empty) -> A:
  match e:

def main() -> Nat:
  0n
```

```
0n
```

LENGTH IN THE TYPE

A `Vec` is a nested pair, and its type depends on a `Nat` length. `first` takes a list of length `1n+n`, which is one or more, so it has no empty case:

```
import Base

def Vec(-A: Type, n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      A & Vec(A, p)

def first(-A: Type, -n: Nat, xs: Vec(A, 1n+n)) -> A:
  (x, rest) = xs
  x

def total(n: Nat, xs: Vec(U32, n)) -> U32:
  match n:
    case 0n:
      0
    case 1n+p:
      (x, rest) = xs
      (x + total(p, rest) : U32)

def main() -> U32 & U32:
  x = first(U32, 2n, (10, (20, (30, Unit{}))))
  s = total(3n, (10, (20, (30, Unit{}))))
  (x, s)
```

```
(10, 60)
```

Two things make this work. The length `n` determines how many pairs `total` consumes, and its recursive call gets the smaller length `p`. The parameter `n` of `first` is erased: it appears only in the type, so it does not exist at run time (the `-` marks it).

Haskell would enforce this with a length-indexed GADT and a `Nat`-kinded phantom type. Bend uses the same idea, but the index is an ordinary function that the checker evaluates.

A PROOF THAT USES A TYPE

A law can take evidence as an argument. `IsEven(x)` below is `Unit` for zero, `Empty` for one, and `IsEven(p)` for `2n+p`. The law says halving an even number and doubling the result gives the number back. The evidence `e` is what makes the law true for only even inputs:

```
import Base

def IsEven(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+0n:
      Empty
    case 2n+p:
      IsEven(p)

def half(n: Nat) -> Nat:
  match n:
    case 0n:
      0n
    case 1n+0n:
      0n
    case 2n+p:
      1n+half(p)

law half_ok:
  for x: Nat
  for e: IsEven(x)
  {Nat.double(half(x)) == x : Nat}

def half_ok(x, e):
  match x:
    case 0n:
      {==}
    case 1n+0n:
      match e:
    case 2n+p:
      %half_ok(p, e) : {2n+Nat.double(half(p)) == 2n+_ : Nat}
      {==}

def main() -> Nat:
  half(8n)
```

```
4n
```

The branch `1n+0n` has no proof body of its own. Its evidence `e` has type `Empty`, so `match e:` with no cases closes the branch. The indentation of `case 2n+p:` is the one the checker accepts after an empty `match`, and it is the same as the bend2.dev example.

TOTALITY: EVERY RECURSION SHRINKS

Bend requires every safe recursive call to pass a structurally smaller input, as chapter 02 explains. The rule holds for proofs as well as for programs, because a proof is also a recursive definition. Two consequences matter in practice.

1. A recursion that does not shrink is rejected, so a loop bounded by the outside world takes a `Nat` fuel argument and matches its predecessor.
2. Mutual recursion is not allowed in a safe definition. Merge the functions into one with a selector argument.

WHEN @UNSAFE CHANGES THE RULES

`@unsafe` disables the termination check for one definition. The compiler still runs the function, but any proof that depends on it is outside the guarantee. This program runs and prints 3, with a warning that the proof check now depends on unsafe code:

```
import Base

@unsafe def idu(n: Nat) -> Nat:
  match n:
    case 0n:
      0n
    case 1n+p:
      1n+idu(p)

def main() -> Nat:
  idu(3n)
```

```
3n
All terms check, but 2 defs rely on unsafe or foreign code:
```

The same definition used by a law is the case that matters. Version 2.0.25 reports it and exits with status 0, so the gate must read the message, not just the status:

```
import Base

@unsafe def idu(n: Nat) -> Nat:
  match n:
    case 0n:
      0n
    case 1n+p:
      1n+idu(p)

law idu_two:
  {idu(2n) == 2n : Nat}

def idu_two():
  {==}
```

```
All terms check, but 2 defs rely on unsafe or foreign code:
```

Be careful with unsafe loops too. A law that requires the checker to evaluate a non-terminating `@unsafe` function does not finish. In one run on this machine, a law of that shape ran for more than 20 seconds before the run was stopped. Keep `@unsafe` out of every definition that a law depends on, and keep laws about terminating functions.

EXERCISES

1. Odd numbers. Write `IsOdd(n)` that is `Empty` for zero, `Unit` for one, and `IsOdd(p)` for `2n+p`. It is the complement of `IsEven`. Check that a function `pred_odd(n, e)` needs `IsOdd(n)`: prove `IsOdd(1n)` by building `Unit{}`, and check that `IsOdd(0n)` is empty.

Solution. `IsOdd` mirrors `IsEven`. The function `pred_odd` takes evidence that its argument is odd. `IsOdd(0n)` is `Empty`, so the zero case is closed by an empty `match`. Odd numbers such as `5n` pass the evidence `Unit{}`:

```
import Base

def IsOdd(n: Nat) -> Type:
  match n:
    case 0n:
      Empty
    case 1n+0n:
      Unit
    case 2n+p:
      IsOdd(p)

def pred_odd(n: Nat, e: IsOdd(n)) -> Nat:
  match n:
    case 0n:
      match e:
    case 1n+p:
      p

def main() -> Nat:
  pred_odd(5n, Unit{})
```

```
4n
```

To check that `IsOdd(0n)` is empty, match on evidence of that type. A `match` with no cases closes the branch:

```
import Base

def IsOdd(n: Nat) -> Type:
  match n:
    case 0n:
      Empty
    case 1n+0n:
      Unit
    case 2n+p:
      IsOdd(p)

def odd_zero_empty(e: IsOdd(0n)) -> Nat:
  match e:

def main() -> Nat:
  1n
```

```
1n
```

2. Fuel. Write `repeat(fuel, x)` that returns `x` plus `fuel`, using the fuel as the loop bound: each step adds 1 to `x`. Explain why it terminates.

Solution. It terminates because the first argument `fuel` is matched and the recursive call gets its predecessor. The following program adds 5 to 10:

```
import Base

def repeat(fuel: Nat, x: U32) -> U32:
  match fuel:
    case 0n:
      x
    case 1n+p:
      repeat(p, (x + 1 : U32))

def main() -> U32:
  repeat(5n, 10)
```

```
15
```

NEXT

Chapter 09 covers the IO monad, files, command-line arguments, and the compilation targets.

A.10 · bend-docs/src/content/tutorial/09-io-files-and-targets.md

Bend 2 tutorial 09: IO, files, and targets · `runnable` · updated 2026-10-18 · canonical `/tutorial/09-io-files-and-targets/`

09: IO, files, and targets

Bend is pure. Effects live in the type `IO`, and a `do` block sequences them, much as Haskell's `IO` monad does. The differences are that every bind names its type, and that a file handle is affine: an operation on a handle gives the handle back with its result. This chapter covers the effect forms, files, arguments, and the three compilation targets.

SEQUENCING EFFECTS

A `do` block runs its lines in order. A bind `x : T <- action` runs an action and names its result with a type. A plain line runs an action whose result is `Unit`, and `return v` ends the block with a pure value:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("first")
    IO.print("second")
```

```
first
second
```

The last line of a `do` block must be an effect or a `return`. A pure term there is rejected:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    (6 * 7 : U32)
```

```
- observed : U32
```

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1)
    return "Hello, " ++ name

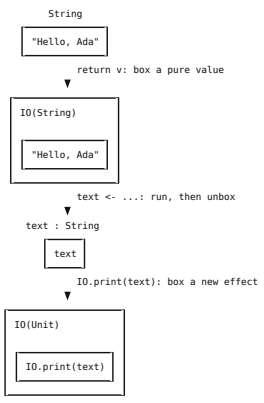
def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    IO.print(text)
```

```
Hello, Ada
```

In that example `IO.sleep(1)` waits one millisecond, and `return` wraps the pure string as an `IO` result. Every bind carries a type, as in `text : String <- greet("Ada")`. Without it, the checker cannot choose the monad's type arguments.

BOXES: PURE, BIND, AND EFFECTS

The picture is a model, not the runtime. An `IO(A)` is an effect: when it runs, it produces an `A`. Read the outer box as the effect and the inner box as the value, which exists only after the effect runs. A `do` block moves values between the boxes:



The same program, checked:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    IO.print(text)
```

```
Hello, Ada
```

- `return v` boxes a pure value. It is `IO.pure`.
- `text : String <- greet("Ada")` runs the effect and takes the value out. This is the only way out of a box.
- `IO.print(text)` makes a new effect: a box of type `IO(Unit)`.

WHAT A DO BLOCK MEANS

The guide defines `do` by two rules. `x : A <- v` becomes `M.bind(xs.., A, R, v, x => ...)`, and `return e` becomes `M.pure(xs.., R, e)`. For IO, Base defines `IO.pure(A, x)` and `IO.bind(A, B, m, f)`. The program above is the same as these calls:

```
import Base

def greet(name: String) -> IO(String):
  IO.pure(String, "Hello, " ++ name)

def main() -> IO(Unit):
  IO.bind(String, Unit, greet("Ada"), text => IO.print(text))
```

```
Hello, Ada
```

A plain line is a bind whose result is `Unit`, so two effects in a row are one bind with a lambda that ignores its argument:

```
import Base

def main() -> IO(Unit):
  IO.bind(Unit, Unit, IO.print("x"), u => IO.print("y"))
```

```
x
y
```

MATCHING ON IO

An `IO(A)` has no constructors, so a constructor pattern on it is rejected:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def show(io: IO(String)) -> IO(Unit):
  match io:
    case Some{x}:
      IO.print("some")
    case None{}:
      IO.print("none")

def main() -> IO(Unit):
  show(greet("Ada"))
```

```
- expected : a datatype
```

A `match` also cannot head a statement inside `do`, because a `match` heads a def body. Binding a value and then matching it in the same block is rejected:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    match text:
      case _:
        IO.print(text)
```

```
- expected : a term (a match heads a def body, not a term)
```

A wildcard `case _:` on an IO parameter is accepted, but its branch cannot see inside the effect. The pattern that works is to bind the value with `<-` and pass it to a def whose `match` inspects it:

```
import Base

def find(n: U32) -> IO(Maybe<&2, U32>):
  do IO<Maybe<&2, U32>>:
    return Some{n}

def show(m: Maybe<&2, U32>) -> IO(Unit):
  match m:
    case Some{x}:
      IO.print(U32.show(x))
    case None{}:
      IO.print("none")

def main() -> IO(Unit):
  do IO<Unit>:
    m : Maybe<&2, U32> <- find(7)
    show(m)
```

```
7
```

CONCURRENCY: FORK AND JOIN

`IO.fork` starts an effect in the background and returns a channel for its result. `IO.join` waits on that channel. Between the two, the main effect keeps running:

```
import Base

def delayed() -> IO(String):
  do IO<String>:
    IO.sleep(1)
    return "done"

def main() -> IO(Unit):
  do IO<Unit>:
    channel : Chan(String) <- IO.fork(String, delayed())
    IO.print("waiting")
    text : String <- IO.join(String, channel)
    IO.print(text)
```

```
waiting
done
```

The event loop interleaves the effects. It is not a parallel call: an effect that waits on a timer, a socket, or a channel steps aside for the others. The guide says each computation runs its pure code, possibly in parallel on every core, up to its next effect. Use `IO.fork` for work that waits, and a parallel call (chapter 05) for pure work that divides.

`Chan.new`, `Chan.send`, `Chan.recv`, and `Chan.close` are the low-level operations under `IO.fork` and `IO.join`. Their signatures are in Base, and `bend base Chan` prints them.

CHANNELS: VALUES BETWEEN COMPUTATIONS

`IO.fork` returns a channel for one result. For a stream of values, make a channel with `Chan.new`, give the sending computation the channel, and receive from the other side. `Chan.new(U32, 4)` makes a channel with room for four values. `Chan.recv` returns `Some{value}` for each value sent, and `None{}` once the channel is closed:

```
import Base

def send_numbers(+chan: Chan(U32)) -> IO(Unit):
  do IO<Unit>:
    sent1 : Bool <- Chan.send(U32, chan, 10)
    sent2 : Bool <- Chan.send(U32, chan, 20)
    Chan.close(U32, chan)

def show_message(m: Maybe<&1, U32>) -> IO(Unit):
  match m:
    case None{}:
      IO.print("closed")
    case Some{n}:
      IO.print(U32.show(n))

def main() -> IO(Unit):
  do IO<Unit>:
    +chan : Chan(U32) <- Chan.new(U32, 4)
    IO.spawn(Unit, send_numbers(chan))
    first : Maybe<&1, U32> <- Chan.recv(U32, chan)
    show_message(first)
    second : Maybe<&1, U32> <- Chan.recv(U32, chan)
    show_message(second)
    third : Maybe<&1, U32> <- Chan.recv(U32, chan)
    show_message(third)
```

```
10
20
closed
```

`IO.spawn` starts a computation whose result nobody waits for. Here the sender runs in the background, and the receiver reads the values in the order they were sent.

FILES: AFFINE HANDLES

A file is a handle, and a handle has one owner. Each operation on it returns the handle beside its result, so you must keep the handle and pass it on. `File.open` returns a `Result`, and `IO.try` unwraps it or ends the program with the error. This program writes a file and closes it:

```
import Base

def finish(result: File & Result<&1, &1, U32 & String, Unit>) -> IO(Unit):
  match result:
    case (file, status):
      do IO<Unit>:
        File.close(file)
        IO.pass(Unit, status)
        IO.print("written")

def main() -> IO(Unit):
  do IO<Unit>:
    file : File <- IO.try(File, File.open("message.txt", "w"))
    result : File & Result<&1, &1, U32 & String, Unit> <- File.write(file, "Hello from Bend")
    finish(result)
```

```
written
```

The program writes `message.txt` in the directory where you run it, and it replaces any earlier contents. The `finish` helper closes the handle on every path before it passes the status along. A handle that is not closed is a leak: the checker does not force you to close it, so you must do it yourself.

The guide's list of effect kinds includes TCP and UDP sockets, windows, and audio. Their handles follow the same affine rule. `bend base` prints their signatures.

COMMAND-LINE ARGUMENTS

`IO.args()` returns the arguments after the program name, as a `List<String>`. The compiler's own options are removed, and `--` ends them. Here the program prints how many arguments it received:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    args : List<String> <- IO.args()
    IO.print(Nat.show(List.length(&1, String, args)))
```

```
0
```

Run it with arguments to see the count change:

```
bend main.bend one two
bend main.bend -- one two
```

Both commands print `2` in this lab. The `List.length` call takes its quantity first, `&1` for a list of affine elements, which is why the version without it is rejected (chapter 11).

ERROR RESULTS AND EXITS

Two Base effects end a program on purpose. `IO.die(-A, code, msg)` ends the program with the exit code and the message. `IO.pass(-A, r)` turns a `Result` into an effect that either returns the value or fails with its error. Prefer `IO.pass` and `IO.try` when the failure can be handled, and reserve `IO.die` for a program that cannot continue.

READING THE ENVIRONMENT

`IO.get_env` returns a `Result`. Matching on it handles the failure case. This variable is unset, so the program prints `missing`:

```
import Base

def describe(r: Result<&1, &1, U32 & String, String>) -> IO(Unit):
  match r:
    case Fail{e}:
      IO.print("missing")
    case Done{v}:
      IO.print(v)

def main() -> IO(Unit):
  do IO<Unit>:
    r : Result<&1, &1, U32 & String, String> <- IO.get_env("BEND_TUTORIAL_UNSET")
    describe(r)
```

```
missing
```

COMPILATION TARGETS

Bend has three targets. The same source compiles to each of them.

```
bend main.bend -o main.c
bend main.bend -o main
./main --threads 8
bend main.bend -o main.js
node main.js
```

- `-o main.c` writes one C file that contains the runtime and your program. Compile it with any C compiler, or inspect it, as chapter 05 does.
- `-o main` builds a native executable with clang 14 or newer. A binary that uses `!` also writes `main.gpu`, which must stay beside it. `--gpu off` runs bang calls on the CPU.
- `-o main.js` writes standalone JavaScript. The JavaScript target runs parallel bindings one after another.

The native and JavaScript targets print the same text, which this lab checked for the program below. It is the example from the compile-to-JavaScript page:

```
import Base

def main() -> IO(Unit):
  IO.print("Bend on JavaScript")
```

```
Bend on JavaScript
```

`bend main.bend` without `-o` runs the program directly. An `IO` main runs compiled, and a pure main is evaluated by the checker, so a pure main is slow for large work.

EXERCISES

1. Two greetings. Write `main` so that it runs `greet("Ada")` and then `greet("Lin")`, printing each greeting on its own line.

Solution. Two binds in one `do` block run in order.

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    first : String <- greet("Ada")
    second : String <- greet("Lin")
    IO.print(first)
    IO.print(second)
```

```
Hello, Ada
Hello, Lin
```

2. Count the handles. Explain why `File.write(file, ...)` returns the handle again, and what would break if it did not.

Solution. A file handle is affine. If the write consumed the handle and returned only the result, the program could not close it afterwards. The handle would be lost, and the file would stay open. Returning the handle with the result keeps the single owner, so the program can close it on every path.

NEXT

Chapter 10 applies everything so far to your questions: forks, branches, and self-changing code.

A.11 · bend-docs/src/content/tutorial/10-branching-and-self-change.md

Bend 2 tutorial 10: branching and self-change · `runnable` · updated 2026-10-18 · canonical `/tutorial/10-branching-and-self-change/`

10: branching and self-change

The request behind this chapter asks for four things: SupGen, cheap copy-on-write forks of code (with `sup` and `dup`), homoiconic code that changes itself, and forking a running program with new code so that the main code keeps working and new branches can be tested cheaply. This chapter says which of these the released Bend 2 compiler offers, which it does not, and what to do instead. The short answer is that the part you can have is the laws-first workflow for pure code, and the rest is either absent from the compiler or cannot be stated as a law.

WHAT EACH REQUIREMENT NEEDS

Requirement	Status in bend 2.0.25	What you can do instead	Can a law pin it?
SupGen (type-directed program synthesis)	not in the compiler, Base, or guide; the site says it is private	write the program and its laws by hand	no
Copy-on-write fork with `sup` and `dup`	no `sup` syntax (chapter 06); no copy-on-write forks	values are immutable; `+` shares by count; `Array.clone` copies	value equality yes; sharing and cost no
Homoiconic, self-changing code	programs are fixed definitions; templates specialize at compile time	represent a program as `Data` and interpret it with `eval`	yes, for the interpreter's behaviour
Cheap fork of code	the calculus forks a term into labelled alternatives that share their input (chapter 14); released Bend 2 has no such construct	a new definition or constructor is a new branch; `main` is unchanged	the result yes; the cost no
Test new branches cheaply and fast	`bend --check-only` and `bend PROOF.bend` check the laws; native builds time the program	the laws-first gate below	the behaviour yes; the time no

Two lines in that table are absolute. A law states what a value computes, and the released compiler has no way to state the cost of sharing. Time and memory are measurements.

THE WORKFLOW THAT WORKS TODAY

For pure code, a branch is a new value or a new constructor, and the checker tests it against the same laws as `main`.

1. **State the requirement as laws.** Put the claims in `LAWS.bend`, and never weaken an accepted law to get a proof through.
2. **Implement the main behaviour as pure functions over `Data`.** Keep the input and output shell thin, so the laws can pin the pure part.
3. **Add a branch as a new constructor or a new definition.** Existing cases do not change, so the main laws keep their proofs.
4. **Write one law per branch,** plus laws that relate the branch to `main`. A *difference law* says where the branch may differ from `main`. A *fork law* says a branch that changes nothing behaves exactly like `main`.
5. **Prove in `PROOF.bend`.** The gate must print `All terms check.` with no line about `unsafe` or `foreign` code.
6. **Measure the branch separately,** with native builds, as chapter 05 describes.

The pattern works because the laws quantify over every constructor. A new constructor that is not covered by a proof makes the gate fail. The compiler enforces that every new branch carries its own proof.

PATTERN 1: A BRANCH AS A TEMPLATE

A template argument is a compile-time choice. Two different steps give two compiled copies of one loop, with no run-time selection:

```
import Base

def run(~step: U32 -> U32, n: Nat, x: U32) -> U32:
  match n:
    case 0n:
      x
    case 1n+p:
      run(~step, p, step(x))

def main_step(x: U32) -> U32:
  (x + 1 : U32)

def branch_step(x: U32) -> U32:
  (x + 2 : U32)

def main() -> U32 & U32:
  (run(~main_step, 3n, 0), run(~branch_step, 3n, 0))
```

```
(3, 6)
```

A law can pin the branch at one input, and the proof is computation. The checker evaluates the concrete U32 arithmetic, so `{==}` is enough:

```
import Base

def branch_step(x: U32) -> U32:
  (x + 2 : U32)

law branch_at_zero:
  {branch_step(0) == 2 : U32}

def branch_at_zero():
  {==}

def main() -> U32:
  branch_step(0)
```

```
2
```

A law over every input needs induction or a proof by cases, as chapter 07 shows. The concrete law is the smallest useful claim, and it runs in the same gate.

PATTERN 2: A PROGRAM AS DATA

Homoiconic code means that a program is also data that other code can build and inspect. Bend does not quote its own definitions, but you can represent an expression as a `Data` type and interpret it. A branch is then a different expression value, and the main expression is untouched. Bend is strict, so the data below is built and not computed: `eval` is the only step that computes a value. To keep work unevaluated, you can use a lambda (`x => ...`), a template (`~f`), or data like this. Chapter 12, section 11, shows all three:

```
import Base

type Expr is Data:
  Lit{n: Nat}
  Add{a: Expr, b: Expr}
  Mul{a: Expr, b: Expr}

def eval(e: Expr) -> Nat:
  match e:
    case Lit{n}:
      n
    case Add{a, b}:
      Nat.add(eval(a), eval(b))
    case Mul{a, b}:
      Nat.mul(eval(a), eval(b))

def main() -> Nat & Nat:
  main_expr : Expr = Add{Lit{2n}, Lit{3n}}
  branch_expr : Expr = Add{Lit{2n}, Mul{Lit{3n}, Lit{2n}}}
  (eval(main_expr), eval(branch_expr))
```

```
(5n, 8n)
```

Both values are pure, so evaluating the branch cannot change the main value. The interpreter `eval` is the place for laws: a law such as "`eval` of `Add{a, b}` is the sum of the evals" can be proved by a case split. This is homoiconic in a limited sense: the program is data you can build and test. It is not self-modification of the running binary, and it does not give you cheap sharing of two programs. Each `Expr` value is built and stored in memory, and the cost is what chapter 05 measures.

PATTERN 3: THE DIFFERENCE BETWEEN BRANCHES

The most useful law for a branch relates it to `main`. Suppose the branch should never give a larger answer than `main`. Write the claim, and let the gate check it:

```
law branch_never_more:
  for x: U32
  {U32.is_le(branch_step(x), main_step(x)) == True{} : Bool}
```

That law is false for the example above, because `branch_step` adds 2 and `main_step` adds 1. The gate would reject it, and that is the point: a difference law states exactly how the branch may differ from `main`, and a branch that breaks it is not accepted. Write the law you want, then change the branch until the gate accepts it, not the other way round.

WHAT IS NOT POSSIBLE, AND WHY

- **SupGen** is not in the released compiler. Nothing in this chapter depends on it, and no law can refer to it.
- **Copy-on-write forks with `sup` and `dup`** need a construct the parser rejects (chapter 06). The nearest things are `Array.fork`, which is `@unsafe` and so cannot support a law, and `Array.clone`, which copies.
- **Bend 2 does not fork a running program.** It compiles each program to a fixed binary, and a second program is a second file that you compile. The calculus forks terms, not processes: a superposition holds two alternatives, and applying it runs both on one shared input (chapter 14). The released compiler does not expose that.
- **Cost is not a law.** The checker proves that a branch computes the right values. It does not prove that the branch is cheap. Measure it, count the keeps in the emitted C (chapter 05), and compare builds.

EXERCISES

1. A third step. Add a template step `triple_step` that adds 3, and make `main` return the three results as a triple. Predict the output before you run it.

Solution. The result is `(3, 6, 9)` for three steps of three applications each. Triples are written `A & B & C`.

```
import Base

def run(~step: U32 -> U32, n: Nat, x: U32) -> U32:
  match n:
    case 0n:
      x
    case 1n+p:
      run(~step, p, step(x))

def main_step(x: U32) -> U32:
  (x + 1 : U32)

def branch_step(x: U32) -> U32:
  (x + 2 : U32)

def triple_step(x: U32) -> U32:
  (x + 3 : U32)

def main() -> U32 & U32 & U32:
  (run(~main_step, 3n, 0), run(~branch_step, 3n, 0), run(~triple_step, 3n, 0))
```

```
(3, 6, 9)
```

2. A law for a concrete input. Write the law that `triple_step(0)` is `3`, and prove it by computation.

Solution. The same shape as `branch_at_zero`. The concrete arithmetic reduces, so `{==}` proves it.

```
import Base

def triple_step(x: U32) -> U32:
  (x + 3 : U32)

law triple_at_zero:
  {triple_step(0) == 3 : U32}

def triple_at_zero():
  {==}

def main() -> U32:
  triple_step(0)
```

```
3
```

NEXT

Chapter 11 lists the checker's rules and every official diagnostic with the message the released compiler prints.

A.12 · bend-docs/src/content/tutorial/11-rules-and-errors.md

Bend 2 tutorial 11: checker rules and errors · `runnable` · updated 2026-10-17 · canonical `/tutorial/11-rules-and-errors/`

11: checker rules and errors

This chapter is a reference. The checker runs before any program executes, and it stops with a message when a rule fails. Each rule below is taught in an earlier chapter, and each official diagnostic appears here with the text `bend 2.0.25` prints. The official index on bend2.dev was written against 2.0.36, and its wording differs in places. Both wordings are in the tables.

THE RULES

Rule	Why it exists	Chapter
A definition is defined above its first use	the checker reads top to bottom	02
Recursive calls shrink their first changing argument	termination makes proofs sound	02, 08
Matches are exhaustive	every value must have a branch	02
A match inspects a parameter or a pattern variable	a computed value has no evidence to split	01, 03
Every operator carries a type annotation	the checker does not infer arithmetic types	01
A constructor in a `let` needs its type from context	the checker cannot infer it otherwise	02
A plain binder is used at most once	ownership and freeing	03
`+` marks reuse of a `Data` value only	only `Data` may be copied	03
The result has the declared type	ordinary typing	01
Every name is defined in the file or an import	no silent fallbacks	02
Every law has a same-named proof, with no holes	the gate	07
A proof does not depend on `@unsafe` or foreign code	unsafe code is outside the guarantee	08, 11

THE OFFICIAL DIAGNOSTICS

Each heading names the diagnostic on bend2.dev. The code is the broken program from the official catalogue, and the error lines are what `bend 2.0.25` printed for it. Where a chapter already shows the program and its message, this page links to it.

A VARIABLE WAS CONSUMED MORE THAN ONCE

Chapter 03 shows the program and its fix. The message is `- expected : n` and `- observed : n (consumed more than once)`.

AN OPERATOR HAS NO TYPE ANNOTATION

Chapter 01 shows the program and its fix. The message is `- message : a type for this operator (write (a + b : Nat))`.

A MATCH CANNOT INSPECT A COMPUTED VALUE

Chapter 01 shows the program and its fix. The message is `- message : a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)`.

A FUNCTION CALLS ONE DEFINED BELOW IT

Chapter 02 shows the program and its fix. The message is `- expected : a defined name`.

A RECURSIVE CALL DOES NOT DECREASE

Chapter 02 shows the program and its fix. The message begins `- expected : a decreasing self-call`.

A NAME IS NOT DEFINED

```
import Base

def main() -> Nat:
  missing()
```

```
- expected : a defined name
- observed : missing
```

Define `missing` above `main`, or import the file that defines it.

A LAW IS MISSING ITS PROOF

Chapter 07 shows the program. The message is `Error: 1 TODO found.`, and the same file with `{==}` as its proof prints `All terms check.`

DEFINITIONS RELY ON UNSAFE OR FOREIGN CODE

Chapter 08 shows the case that matters. The official program in this catalogue uses names that end in `?`, which bend 2.0.25 cannot parse:

```
import Base

def is_even?(n: Nat) -> Bool:
  match n:
    case 0n: True{}
    case 1n+p: is_odd(p)
```

```
- expected : '('
```

The fix in 2.0.36 is `@unsafe`, and 2.0.25 accepts that spelling too. The rule itself: a law that depends on an `@unsafe` definition is not verified. Chapter 08 shows the warning this version prints.

THE RESULT HAS THE WRONG TYPE

```
import Base

def main() -> U32:
  True{}
```

```
- expected : U32
- observed : Bool
```

Return a value of the declared type.

ERRORS THIS TUTORIAL FOUND BEYOND THE OFFICIAL LIST

These errors were produced by real programs while writing the chapters. They are not in the official index.

A superposition literal. `&{1, 2}` is rejected (chapter 06).

An `if` statement. Bend has no `if` (chapter 06).

A constructor without a type. A `let` with no context cannot infer the type of a constructor:

```
import Base

type Tok is Data:
  Tok{n: Nat}

def show(t: Tok) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  t = Tok{3n}
  show(t)
```

```
- expected : an annotated term (cannot infer)
```

Write `t : Tok = Tok{3n}`, as chapter 02 does.

A quantity argument left out. `List.length` takes its quantity first, so `List.length(String, args)` fails:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    args : List<String> <- IO.args()
    IO.print(Nat.show(List.length(String, args)))
```

```
- expected : Quant
```

Write `List.length(&1, String, args)`, as chapter 09 does.

A destructured computed value. `(a, b) = f(x)` inside a definition is a match on a computed value (chapter 03).

A hole inside a branch. `?name` stops the proof check and prints the goal (chapter 07).

A constructor field naming a later type. Chapter 02 shows the message.

VERSION DIFFERENCES

The index page is the table of contents. The table below compares the compiler installed for this tutorial, bend 2.0.25, with the 2.0.36 build the bend2.dev examples were checked against.

Topic	bend 2.0.25 (checked here)	bend 2.0.36 (site)
Proof success line	`All terms check.`	`ALL PROOFS CHECK and Use --verdict for mathematical validity.`
`--verdict` option	`unknown option --verdict`	supported
Proof that depends on `@unsafe`	prints `All terms check, but N defs rely on unsafe or foreign code` and exits 0	fails the proof check
Datatype fields naming a later type	rejected (`expected : a defined name`)	the guide says datatypes may appear in any order
`f?` names	parse error	accepted as shorthand for `@unsafe`
Forward reference message	`- expected : a defined name`	`- expected : a filled definition (an unfilled law is a dead claim: live code cannot use it)`

Two consequences for your workflow. First, the gate must read the output, not only the exit status, because 2.0.25 can exit 0 with an unsafe dependency. Second, `bend update` installs the latest compiler. The site's examples use 2.0.36, which has `--verdict`. The bend2.dev notes say that option needs Lean v4.34.0 exactly, or a prebuilt kernel named by `BENDTT`. Chapter 07 explains what the final check adds. This lab has neither the compiler option nor a Lean toolchain, so it has not run that check.

READING AN ERROR

Every message has the same shape: a headline, then `- expected` and `- observed` lines, and sometimes a `Location` with the source around it. Read `expected` first. It names what the checker needed at that point, such as `a defined name`, `Data`, or `cases for False`. Then find the same spot in your file. The `observed` line shows what it found instead.

NEXT

Chapter 12 is the syntax reference, and chapter 13 is the glossary.

A.13 · bend-docs/src/content/tutorial/12-syntax-reference.md

Bend 2 tutorial 12: syntax reference · runnable · updated 2026-10-18 · canonical /tutorial/12-syntax-reference/

12: syntax reference

This page is the language on one sheet. Read sections 1 and 4 once, then use the tables as a lookup. Blocks marked bend are checked by the tutorial gate with bend 2.0.25. The table rows were also run with that compiler when this page was written, except the forms that section 13 lists as not run. Section 12 gives the checker's exact messages for the common mistakes.

1. THE MODEL

Three layers. Each one classifies the layer below it.

Layer	Classifies	Examples
Kind	types	Type (Kind(&1): used at most once), Data (Kind(&2): copied with +)
Type	values	Nat, U32, Shape, U32 -> U32, Maybe<&2, Nat>, Array<U32>, IsZero(n)
Value	the bottom layer	6n, Point{}, (1, 2), [1, 2], x => e, [0 : U32*8n]

Types are terms, so a definition can return one. The guide splits a program in two. *Live* code runs. *Dead* code is types, erased arguments and proofs: the checker reads it, and the compiled program never runs it.

```
import Base

def twice(+n: Nat) -> Nat:
  Nat.add(n, n)

def identity(-A: Type, x: A) -> A:
  x

def main() -> Nat:
  identity(Nat, twice(3n))
```

```
6n
```

In that call, identity is code. Nat is an erased argument, so the compiled program never receives it. twice(3n) is a value computed at run time.

2. DECLARATIONS

Form	Meaning
import Base	the prelude; every file starts with it
import ./m.bend as M	the definitions of m.bend are named M.x
type T is Data:	a datatype whose values may be reused; is Type makes it affine
type D<a, -A: Kind(a)> is Kind(a):	a datatype whose kind comes from a parameter; one constructor per line, as K{x: A}
def f(x: A, -y: B, +z: C) -> T:	a definition: x is used at most once, -y is erased, +z is reusable; the body follows the :
def t(~g: A -> B, x: A) -> B:	a template parameter; template parameters come first
law f: with def f(x):	a claim, and its proof: a def with the same name

Form	Meaning
`@unsafe def f(...)`	skips the termination check; no law may depend on it
`def e(x: A) -> IO(B): with import "./e.c"`	a foreign effect built from a host file (section 13)

```
import Base

type Box<a, -A: Kind(a)> is Kind(a):
  Box{value: A}

def unbox(a, -A: Kind(a), b: Box<a, A>) -> A:
  match b:
    case Box{value}:
      value

def main() -> U32:
  unbox(&1, U32, Box{7})
```

```
7
```

3. MARKS AND PUNCTUATION

Mark	Meaning	Example
`name: T`	has type: a parameter, a field, a `for` binder	`x: U32`
`(e : T)`	annotates an expression; covers the operators inside its own parentheses	`(a + b : U32)`
`{t : T}`	annotates one term: a literal, a name or a constructor	`{12 : U32}` , `{Tok{3n} : Tok}`
`K{...}`	a constructor (ctor); the brace touches the name	`Segment{3n}`
`{a == b : T}`	equality: a proposition	`{Nat.add(x, 0n) == x : Nat}`
`{a != b : T}`	negated equality, the same as `{a == b : T} -> Empty`	`{1n != 0n : Nat}`
`{==}`	reflexivity: proves an equality when both sides compute to one term	
`->`	separates the parameters from the result type	
`=`	binds a value; `a : ` never means equals	`x = v`
`<-`	binds the result of an effect inside `do`	`x : T <- m`
a line ending in `:`	opens a block: a body, a `case` , a `do` , a `law` , a `type`	
`#`	a comment to the end of the line	
`?name` , `?TODO`	a hole: `?name` shows the goal; `?TODO` leaves it open	

- A name directly before braces makes a constructor. Braces with no name make an annotation (`:`), an equality (`==`), or a proof (`{==}`).
- No space goes between a constructor and its braces. `Some{4n}` is a constructor, and `Some {4n}` is rejected.
- Operators need a space on each side: `a + b` , not `a+b` .
- `==` appears only in types. Compare values with `U32.is_eq(a, b)` .
- Signatures write `IO(A)` and `Chan(A)` with parentheses, and datatypes with angle brackets, as in `Maybe<&2, Nat>` . A `do` header takes angle brackets: `do IO<Unit>:` .

4. KINDS, QUANTITIES, AND ERASURE

Form	Meaning
`Type = Kind(&1)`	affine: a value may be used at most once
`Data = Kind(&2)`	reusable: `+` may copy a value of this kind
`Kind(a)`	the kind is a parameter, with `a` equal to `&1` or `&2`. The guide also lists `&0`; this tutorial does not use it
`a <&> b`	the smaller of two quantities, which a type holding two element types takes (from the guide; not run here)
`is Data`, `is Type`	the kind of a declared type
`List<U32> = List<&1, U32>`; `+List<U32> = List<&2, U32>`	`+` in front of a type gives the reusable form
`-x`	erased: exists only for the checker, appears only in types and proofs, and is deleted
`x`	affine: used at most once; dropping it is free
`+x`	reusable: the type must be `Data`
`~f`	template: code substituted at compile time; its argument is closed; one copy per distinct argument
`f(Nat, ...)`	an erased argument is passed explicitly, with no marker at the call
`a` in a parameter list	short for `-a: Quant`

```
import Base

def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n:
      Nil{}
    case 1n+p:
      x <> replicate(A, p, x)

def main() -> List<U32>:
  replicate(U32, 3n, 7)
```

```
[7, 7, 7]
```

```
import Base

def dup_k(-A: Kind(&2), +x: A) -> A & A:
  (x, x)

def main() -> Nat & Nat:
  dup_k(Nat, 6n)
```

```
(6n, 6n)
```

`replicate` mixes three kinds of parameter. `-A` is erased, `n` is affine, and `+x` is reusable because each step uses `x` twice. `<>` puts the head in front of the rest.

- A `Type` argument is written at the call: `identity(U32, 42)`. Bend infers no type arguments (section 12).
- `+` needs `Data`. A function is a `Type` and cannot be copied, so pass a template `~f` instead (chapter 03).
- An erased argument does not reach the compiled program. Compile the same program with `-A` and without it, and compare the C: the version without `-` passes the type in and drops it (chapter 03).

5. TYPES

Form	Meaning
`Nat` , `U32` , `F32` , `Bool` , `Char` , `String`	base types; a `Nat` literal ends in `n` , as in `3n`
`Unit` , `Empty`	one value, `Unit{}` ; no value
`A -> B`	a function, which is a `Type` , so a closure is affine
`@x: A -> B`	a dependent function: `B` may name `x`
`@-x: A -> B`	a dependent function whose argument is erased
`A & B`	a pair
`&x: A -> B`	a dependent pair: `B` may name the first component `x`
`A | B`	a sum: `Inl{a}` or `Inr{b}`
`D<A>` , `D<&2, A>`	a datatype applied to arguments
`Array<A>`	a one-owner array, which is a `Type`
`IO(A)` , `Chan(A)`	an effect that returns `A` ; a channel of `A`
`Result<&1, &1, E, A>`	a fallible result: error `E` , value `A`
`{a == b : T}` , `{a != b : T}`	equality and its negation: propositions
a def that returns `Type`	a type computed from values, such as `IsZero(n)`

```
import Base

def IsZero(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      Empty

def needs_ne(e: {1n != 0n : Nat}) -> Nat:
  1n

def require_zero(n: Nat, e: IsZero(n)) -> Nat:
  n

def main() -> Nat:
  require_zero(0n, Unit{})
```

```
0n
```

```
import Base

def pick(b: Bool) -> U32 | Bool:
  match b:
    case True{}:
      Inl{4}
    case False{}:
      Inr{False{}}

def main() -> U32 | Bool:
  pick(False{})
```

```
Inr{False{}}
```

6. TERMS AND VALUES

Form	Meaning	Example
`42`, `1.5`, `3n`, `'c'`, `"s"`	literals: `U32`, `F32`, `Nat`, `Char`, `String`	`(6 * 7 : U32)`
`K{a, b}`	a constructor applied: `Ctor(K, [a, b])`	`Segment{3n}`
`(a, b)`	a pair	`(19, 23)`
`[a, b]`	a list: `Con{a, Con{b, Nil{}}}`	`[5, 10, 15]`
`h <> t`	a head placed in front of a list	`x <> replicate(A, p, x)`
`x => e`	a lambda: a closure, which is affine	`x => (x + 1 : U32)`
`+x => e`	a lambda whose binder is reusable	`+y => (y + y : U32)`
`f(a, b)`	a call	`twice(3n)`
`f!(a)`	a call on the GPU, with the parallel calls inside it; the CPU on a machine without a GPU	`pow2!(12n)`
`t(~g, a)`	a call with a template argument	`twice(~inc, 40)`
`(e : T)`, `{t : T}`	an annotated term	`+t = {Tok{3n} : Tok}`
`[v : T*n]`	an array of `n` slots; `n` is a power of two	`[0 : U32*8n]`
`[v : T^d]`	an array of `2^d` slots	`[0 : U32^3n]`
`a[i]`, `a[i] <- v`	a read returns the array with the element; a write returns the array. Indexes wrap	`a[9]` on eight slots is `a[1]`

```
import Base

def inc(x: U32) -> U32:
  (x + 1 : U32)

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~inc, 40)
```

```
42
```

```
import Base

def value(r: Array<U32> & U32) -> U32:
  match r:
    case (array, item):
      item

def main() -> U32:
  a = [0 : U32*8n]
  a[9] <- 7
  value(a[1])
```

```
7
```

- A constructor takes its type from its context. In a `let` with no context, annotate it: `+t = {Tok{3n} : Tok}` (chapter 03).

- The braces after a constructor name hold its fields in a `type` (`Segment{length: Nat}`), its arguments in a term (`Segment{3n}`), and its binders in a pattern (`case Segment{length}:`) (chapter 02).
- There is no `if` . Branch with `match` on `True{}` and `False{}` (chapter 01).

7. STATEMENTS AND MATCHING

Form	Meaning
`x = v`	a `let` : affine
`+x = v`	a reusable `let` : the type must be `Data`
`-x: T = v`	an erased `let` : the type is required
`x: T = v`	a typed `let` : required inside `do`
`(a, b) = v` , `K{x, y} = v`	a destructuring `let`
`a b = f(x) g(y)`	parallel calls: both start, and the line continues when both finish
`match x:` , `match a b:`	match on parameters, or on variables bound by a pattern; never on a computed call
`case K{x}:` , `case 1n+p:` , `case _ _:`	patterns; `_` matches anything; a case body may sit on the same line
`case K{+r}:`	a field bound with `+` is reusable; a plain field binder is affine, so a second use is rejected
`do M<xs, R>:`	a monadic block (section 10)

```
import Base

type Pt is Data:
  Pt{x: U32, y: U32}

def sum_pt(p: Pt) -> U32:
  Pt{x, y} = p
  (x + y : U32)

def swap(p: U32 & U32) -> U32 & U32:
  (a, b) = p
  (b, a)

def main() -> U32 & U32:
  swap((sum_pt(Pt{3, 4}), 0))
```

```
(0, 7)
```

- A recursive call must make a smaller argument. Arguments are read left to right, and each is passed unchanged until one of them is a smaller part. Otherwise the checker prints `expected : a decreasing self-call` (chapter 02).
- `@unsafe` turns that check off, and section 9 says what that costs.

8. OPERATORS

Form	Operand type	Example
`+ - * / %`	the `T` in the annotation	`(a * b : U32)`
`.&. .|. .^.`	the `T` in the annotation	`(6 .|. 1 : U32)`
`<< >>`	the `T` ; the shift count is a `Nat`	`(1 << 3n : U32)`
`< <= > >=`	the `T` ; the result is `Bool`	`(a <= b : U32)`
`&& ||`	`Bool`	`(False{} || True{})`

Form	Operand type	Example
`++`	`String`	`"ab" ++ "c"`
`T.is_eq(a, b)`	value equality	`U32.is_eq(a, b)`

```
import Base

def le(a: U32, b: U32) -> Bool:
  (a <= b : U32)

def main() -> U32 & U32 & U32 & U32 & Bool:
  ((6 .|. 1 : U32), (1 << 3n : U32), (8 >> 2n : U32), (7 % 2 : U32), le(3, 5))
```

```
(7, 8, 2, 1, True{})
```

- Each operator needs an annotation. `(a + b : U32)` covers the operators in its own parentheses. A call's arguments start a new group, so `(U32.add(a + b, 1) : U32)` is rejected (section 12).
- Braces annotate one term. `{12 : U32}` is fine, but `{6 * 7 : U32}` is rejected, so write `(6 * 7 : U32)`.
- A bare operator is rejected. The guide says it means `Nat`, but 2.0.25 rejects it, so write `(n + 1n : Nat)`.

9. LAWS, PROOFS, AND PROPERTIES

A law is a claim, and a def of the same name is its proof. The proof of `add_zero` splits the cases of `x`, uses reflexivity in each, and rewrites with the induction hypothesis.

```
import Base

law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> Nat:
  Nat.add(42n, 0n)
```

```
42n
```

Form	Meaning
`law f:`	a claim; its binders and the claim follow, indented
`for x: T`	a universal binder; `for +x: T` makes it reusable
`for y: B where P(y)`	`y` is a pair: the value and a proof of `P(y)`
`exs z: T`	a witness: the proof returns `(z, proof)`
`def f(x):`	the proof of law `f`: a def with the same name
`def M.f(x):`	the proof of a law `f` imported from module `M` (chapter 15)
`match x:` in a proof	case analysis: each case is a goal
`{==}`	proves an equality when both sides compute to one term

Form	Meaning
`%e : P`	rewrites the goal with the equation `e`; the next line continues the proof
`%e@E : P`	a named rewrite (guide): the binder before `@` is one name (section 12). This tutorial has no working example
`?name`, `?TODO`	a hole (section 3)

Properties the checker guarantees:

- A law with no def is an open claim. A hole is open too: `?TODO` makes `bend PROOF.bend` print `Error: 1 TODO found.` and refuse the file.
- A false law fails the gate. The gate prints `All terms check.` only when every law has a proof that checks (chapters 07 and 15).
- A safe recursive def terminates, because each recursive call shrinks an argument (section 7). `@unsafe` is outside these guarantees, so no law may depend on it (chapter 08).
- A plain value is used at most once. A second use is an error, `consumed more than once`.
- An erased value is deleted before the program runs (section 4).
- A call computes its arguments first (section 11).
- Nothing is inferred, so type arguments, `let` types and operator types are written (section 12).

10. EFFECTS AND DO

Form	Meaning
`IO(A)`	an effect that returns `A`
`do IO<A>:`	a block; its header takes angle brackets
`x : A <- m`	runs the effect `m` and names its result, with type `A`
`m`	a line that runs an effect whose result is `Unit`
`return v`	ends the block with a pure value. The last line is a `return` or an effect; a bare term is rejected
`IO.print(s)`, `IO.args()`, `IO.get_env(k)`	print a string; the command-line arguments; an environment variable, as a `Result`
`IO.die(-A, code, msg)`	ends the program with an exit code and a message
`IO.fork(A, m)`, `IO.join(A, c)`	start `m` in the background and get a channel for its result; wait for that result
`IO.spawn(A, m)`	starts `m` and does not wait for it
`Chan.new(A, n)`, `Chan.send(A, c, v)`, `Chan.recv(A, c)`, `Chan.close(A, c)`	a channel of `A` with room for `n` values; `Chan.recv` answers a `Maybe`, and `None{}` once the channel is closed

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    n: U32 = (6 * 7 : U32)
    IO.print(text ++ " " ++ U32.show(n))
```

```
Hello, Ada 42
```

11. EVALUATION: BEND IS STRICT

- A compiled Bend program is strict: a call computes its arguments before its body runs. A native build of `first(1n, spin(0n))`, where `spin` loops forever and `first` ignores its second argument, did not finish under `timeout 20`. With `0n` in place of `spin(0n)`, it printed `1` at once.
- The checker does not show evaluation order. `bend file.bend` normalises a pure `main`, so judge the order with a native build under `timeout`.
- To keep work unevaluated, use one of three forms: a lambda (`x => ...`), whose body runs when it is applied; a template (`~f`), which is substituted at compile time; or data that `eval` interprets, as below.
- Parentheses and annotations delay nothing. Tail calls compile to loops, and the runtime has no C stack: each definition is a segment of a flat state machine, and a call is a jump (the guide; chapter 05).

```
import Base

type Expr is Data:
  Lit{n: Nat}
  Add{a: Expr, b: Expr}

def eval(e: Expr) -> Nat:
  match e:
    case Lit{n}:
      n
    case Add{a, b}:
      Nat.add(eval(a), eval(b))

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32 & Nat & U32:
  later: U32 -> U32 = x => (x + 1 : U32)
  program : Expr = Add{Lit{2n}, Lit{3n}}
  (later(41), eval(program), twice(~(x => (x + 1 : U32)), 40))
```

```
(42, 5n, 42)
```

12. REJECTED FORMS

Each row is a program the checker rejects. The message is the checker's own, and the right column is the fix.

Wrong	The checker says	Right
`(a+b : U32)`	`expected : '='`, `observed : ':'`	`(a + b : U32)`
`x + y` with no annotation	`a type for this operator (write (a + b : Nat))`	`(x + y : U32)`
`{6 * 7 : U32}`	`a type for this operator (write (a * b : Nat))`	`(6 * 7 : U32)`
`(a == b)`	`expected : ')'`, `observed : '='`	`U32.is_eq(a, b)`
`price = 12`	`expected : an annotated term (cannot infer)`	`price = {12 : U32}`
`-n = 3n`	`expected : an annotated term (cannot infer)`	`-n: Nat = 3n`
`x = 3` inside `do`	`expected : a pattern (a binder or a constructor)`	`x : U32 = 3`
`do IO(Unit):`	`expected : '<'`, `observed : '('`	`do IO<Unit>:`
`-> IO<Unit>`	`a declared datatype (unknown: IO)`	`-> IO(Unit)`
`-> Maybe(&2, Nat)`	`expected : a family instance (write Maybe<..>)`	`-> Maybe<&2, Nat>`
`Some(4n)`	`expected : a defined name`, `observed : Some`	`Some{4n}`
`Some {4n}`	`expected : 'def', 'type' or 'law'`, `observed : '{'`	`Some{4n}`

Wrong	The checker says	Right
`use(+f: U32 -> U32)`	`expected : Data , observed : Type`	a plain `f: U32 -> U32` used once, or a template `~f`
`+t` on a `type Token is Type:`	`expected : Data , observed : Type`	declare `type Token is Data:`
`identity(42)` with `-A: Type`	`expected : Type , observed : U32`	`identity(U32, 42)`
`bad(-A: Type, x: A) -> A` returning `A`	`expected : A , observed : Type`	return `x`
`f(f(x))` with `f: U32 -> U32`	`expected : f , observed : f (consumed more than once)`	a template `~f`
a template argument that uses a local `k`	`a template applied to closed ~ arguments (k is a variable here, not comptime: pass it at run time)`	a top-level definition, as in `~inc`
`match U32.is_eq(1, 1):`	`a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)`	pass the call to a def and match its parameter
`[0 : U32*6n]`	`a power of two count (^d takes a depth)`	`[0 : U32*8n]` or `[0 : U32^3n]`
a `do` block that ends in `(6 * 7 : U32)`	`observed : U32`	end with `return v` or an effect
`loop(n)` calling itself with the same `n`	`expected : a decreasing self-call`	a call on a smaller part
`ih = add_zero(p)` in a proof, then `{==}`	`expected : 1n+Nat.add(p, 0n) , observed : 1n+p`	`%add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}`
`%add_zero(p)@ih : P`	`expected : a name before @ (a rewrite binder is one name: %e@E : P)`	`%add_zero(p) : P`
`?goal` in a case	`expected : {0n == 0n : Nat} , observed : ?goal`	fill the case with the goal, as `{==}`
`?TODO` left in a proof	`Error: 1 TODO found.`	a complete proof
a `where` pair used as a `Nat`	`expected : Nat , observed : Sigma<&1, &1, Nat, n => IsNonZero(n)>`	the claim must treat `n` as the pair; this tutorial does not show reading its value

13. NOT RUN HERE

- Foreign effects (`import "./e.c"` , `import "./e.js"`) need host files. `bend guide effects` describes them.
- `bend main.bend --publish` , `bend page.html -o dist` , and `import 0x<hash>/main.bend as P` need the package hub or the web bundler.
- A working named rewrite `%e@E : P` , and a `where` binder whose claim reads its value out of the pair (section 12).
- GPU timing: the lab machine has no GPU, so `!` runs on the CPU here.
- The `a <&> b` quantity combinator, which the guide describes and this page does not run.

NEXT

Chapter 13 maps these terms onto Haskell and OCaml. Chapter 15 uses the forms from this page in one complete program.

A.14 · bend-docs/src/content/tutorial/13-glossary.md

Bend 2 tutorial 13: glossary for Haskell and OCaml programmers · `docs-only` · updated 2026-10-18 · canonical `/tutorial/13-glossary/`

13: glossary for Haskell and OCaml programmers

Each row says what a Haskell or OCaml programmer would reach for first, and what Bend 2 does with the same idea. The last column flags a difference that will change how you write code.

CONCEPTS YOU ALREADY KNOW

You know (Haskell / OCaml)	Bend 2	Difference that matters
`data T = A | B` / `type t = A | B`	`type T` is `Data:` with constructors `A{}` and `B{}`	constructors always use braces; a field is `name: Type`
`case x of` / `match x with`	`match x:` with `case K{...}:`	the scrutinee must be a parameter or pattern variable
`if c then a else b`	`match c:` on `True{}` and `False{}`	there is no `if`
`Maybe a` / `option`	`Maybe<a, A>` in Base	the first argument is the quantity or kind
`Either a b` / `(a, b) result`	`Result<a, b, E, A>` in Base	the quantities are explicit; `IO.try` unwraps
tuples `(a, b)` / `a * b`	`A & B` and `(a, b)`	pairs are the usual way to return two values
lists `x : xs` (Haskell) / `x :: xs` (OCaml)	`Con{head, tail}`, `h <> t`, `[a, b]`	a list is affine unless its quantity is `&2`
`do` notation	`do IO<T>:` with `x : T <- act`	every bind names its type
`return x` / `pure x` (Haskell)	`return x` in a `do` block (`IO.pure`)	the value is boxed as an effect; `<-` runs it; chapter 09
`main :: IO ()`	`def main() -> IO(Unit):`	the same, with `Unit{}` as the value
`where`, local `let`	`x = v` in a body	a plain `let` is used once; `+x = v` is reusable
type classes	not present; Base uses `Type.verb` names	every operation is a named function, such as `U32.add`
`newtype` / abstract types	a `type` with one constructor	constructors are always visible
QuickCheck property	`law`	a law is a checked proof, not a sampled test
GADT with a length index	a type computed by a `def`, such as `Vec(A, n)`	the index is an ordinary function
`INLINE` or specialization	a template `~f`	each distinct argument gets its own compiled copy
linear types	affine binders, `+` for `Data`	`+` pays a reference count; there is no `Linear` keyword
`Data.Map`	`Map<&2, V>`	string keys; `get` takes its default first
mutable arrays in `ST`	`Array<T>` with `a[i] <- v`	one owner; a write returns the array
`Just x` (Haskell), `Some x` (OCaml)	`Some{x}`	a constructor takes braces with no space before them; chapter 02
`f :: a -> b` (Haskell), `val f : a -> b` (OCaml)	`def f(x: A) -> B:`	`:` gives a type, `->` gives the result, and the final `:` opens the body; chapter 12
`(e :: t)` (Haskell), `(e : t)` (OCaml)	`(e : T)`	the annotation covers the operators inside its own parentheses; chapter 12

You know (Haskell / OCaml)	Bend 2	Difference that matters
`==` as a comparison	`T.is_eq(a, b)`	`==` appears only in an equality type such as `{a == b : T}`; chapter 12
`id :: a -> a` (Haskell), `let id x = x` (OCaml)	`def id(-A: Type, x: A) -> A:`	the type is an argument: `id(U32, 42)`. Bend does not infer it; chapter 03
`{x : T}` as a constructor	`{t : T}` as an annotation	a constructor has a name before its braces (`Tok{3n}`); braces with no name annotate one term; chapter 02

TERMS SPECIFIC TO BEND 2

Affine. A binder used at most once. Dropping it is free. Chapter 03.

Strict. Compiled Bend computes a call's arguments before the body runs. To keep work unevaluated, use a lambda, a template, or data that an interpreter runs. Chapter 12, section 11, shows all three.

Quantity. The number of uses a binder may have: `&1` means at most once, and `&2` means reusable. The binder markers choose one: `+x` is reusable, `x` is used at most once, and `-x` is erased, existing only for the checker. Chapter 03.

Kind. A classifier of types that carries a usage budget, written `Kind(q)`, where `q` is a quantity. `Type` is `Kind(&1)`: its values are used at most once. `Data` is `Kind(&2)`: its values may be copied with `+`. A kind is not a quantity. The quantity is the count, and the kind is the classifier that carries it. Haskell's kinds are different; see the table below. Chapter 03.

Constructor (ctor). A named way to build a value of a `type`. `Segment{3n}` applies the constructor `Segment` to `3n`. Chapter 02.

Term and value. A term is an expression as written, such as `(6 * 7 : U32)`. Its value is what it computes to, `42`. Chapter 12, section 11.

Annotation. `(e : T)` gives the type of an expression, and `{t : T}` gives the type of one term. An operator needs parentheses. Chapter 01.

Erased (-). A binder that exists only for the checker. The compiler deletes it, so it is not a run-time argument. Chapter 03.

Runtime use. The compiled body needs the binder's value: it returns the value or computes with it. An erased binder cannot be used at run time. Chapter 03.

Explicit type argument. Bend infers almost nothing, so a type is passed at the call, as in `identity(U32, 42)`. Chapter 03.

Closure. A function value, such as `x => (x + 1 : U32)`. It is affine, so it may be called once. Chapter 03.

Array literal. `[v : T*n]` or `[v : T^d]`: a fill value `v`, an element type `T`, and either `n` slots (a power of two) or `2^d` slots. `Array.new(T, d, v)` builds the same array. Chapter 03.

Law. A claim checked by the compiler. A law has a same-named proof def. Chapter 07.

Proof. A definition whose type is a law's claim. Chapter 07.

Curry-Howard. A law is a type, and its proof is a program of that type. Chapter 07 shows what the gate checks beyond the type.

`{==}`. Proves an equality when both sides compute to the same term. Chapter 07.

Rewrite `%e : P`. Replaces the goal using the equation `e`, with `_` marking where the right-hand side goes. Chapter 07.

Hole. `?TODO` leaves a goal open, and `?name` prints the goal. Chapter 07.

`exs`. A law that asks for a witness; the proof returns it with the equation. Chapter 07.

Unit and Empty. `Unit` has one value. `Empty` has none, so a function that takes an `Empty` need only match nothing. Chapter 08.

Dependent type. A type computed by a `def` from a value, such as `IsZero(n)` or `Vec(A, n)`. A normal type such as `U32` has no parameters, and a function type such as `U32 -> U32` maps values to values. Chapter 08.

Pure (`return`). `return v` in a `do` block boxes a pure value as an effect. It is `IO.pure`. Chapter 09.

Bind (`<-`). Runs an effect and names its result with a type. It is `IO.bind`. Chapter 09.

IO value. An effect of type `IO(A)`. It has no constructors, so only `<-` opens it. Chapter 09.

Totality. Every safe recursive call shrinks its first changing argument. Chapter 02.

Fuel. A `Nat` argument that bounds a loop the outside world stops. Chapter 02.

`@unsafe`. A definition exempt from the termination check and outside the proof guarantees. Chapter 08.

Parallel call. `a b = f(x) g(y)`, two calls the compiler may run at once. Chapter 05.

Bang. `f!(x)`, the call on the GPU, with its parallel calls. Chapter 05.

Balanced fork. A parallel call whose halves do about the same work. Chapter 05.

Leaf. A tail loop over 8 to 64 items that runs on one core. Chapter 05.

Thread count. The `--threads N` setting of a native binary. It defaults to the CPU count, and the fastest value depends on the program, the input, and the machine, so measure it. Chapter 05.

`term_keep`, `ctr_take`, `rfc_seal`. Runtime names for reference counts in the emitted C. Chapter 05.

BendRT. The runtime that compiles Bend to flat C with affine ownership and balanced forks. Chapter 04.

Interaction net. A graph of cells rewritten by local rules. Chapter 04.

Cell and port. A cell has a symbol, one principal port, and auxiliary ports. Chapter 04.

Active pair. Two cells joined at their principal ports. Only these reduce. Chapter 04.

Confluence. Distinct active pairs commute, so the order does not change the result. Chapter 04.

CON, DUP, ERA. The three symbols of Lafont's symmetric interaction combinators. Chapter 04.

Duplication (dup). `! &L{x,y} = t; body` splits `t` into two uses, with label `L`. Chapter 14.

Superposition (sup). `&L{a,b}` holds two alternatives with label `L`. In a net it is a DUP cell with label `L`, read from the other side. The released Bend 2 parser rejects it. Chapter 14.

Label. The number on a `&L{}` or `! &L{}`. Equal labels pair superpositions, and different labels commute. Chapter 14.

Interaction Calculus (IC). Victor Taelin's text notation for labelled nets: λ, application, duplication, superposition, and erasure. Chapter 14.

Collapse. Rewrites a result so that no superposition or duplication remains inside its alternatives (`ic run FILE -C`). Chapter 14.

SupGen. Victor Taelin's program synthesizer, built on superposition search. Its source is not published, and released Bend 2 does not include it. Chapter 14.

Homoiconic. Code that is also data the program can build and inspect. Chapter 10 shows a limited form.

Gate. `bend PROOF.bend`. It passes only when every law is proved, with no holes and no unsafe dependency. Chapter 07.

TERMS THAT LOOK ALIKE BUT DIFFER

Term	Looks like	Actually
`Type`	Haskell's `Type`, the kind of ordinary types (`Int :: Type`)	Bend's kind `Kind(&1)`: values of these types are used at most once. Haskell values are shared and garbage-collected, so Haskell types are not affine. In Bend, `Type` is also the name of the one universe (`Type : Type`, in the guide).
`Data`	the Haskell `Data` class (`Data.Data`, `deriving Data`)	Bend's kind `Kind(&2)`: values may be copied with `+`. It has nothing to do with generic programming
`data`	Haskell's `data T = A | B` keyword	Bend declares a datatype with `type T is Data:`. `is Data` sets its reuse kind, and `is Type` makes it affine
`Kind`	Haskell's kind of a type (`Maybe :: Type -> Type`)	a classifier of types that carries a usage budget, `Kind(q)`; it is not a quantity

Term	Looks like	Actually
`quantity`	Haskell's `%1 ->` multiplicity (LinearTypes, off by default)	the number of uses a binder may have: `&1` at most once, `&2` reusable. The binder markers choose one
`+`	the addition operator	a quantity marker; `(a + b : T)` is the operator
`-A`	negation	an erased parameter
`~f`	a function value	a template argument: copied into each call at compile time. It is not a closure and does not defer work
`IO`	a monad you can run anywhere	an effect whose binds carry types; run only by the event loop
`Array`	a Haskell array	one-owner storage updated in place

NEXT

Return to the table of contents in the index.

A.15 · bend-docs/src/content/tutorial/14-superposition-and-supgen.md

Bend 2 tutorial 14: superposition, collapse, and SupGen · `runnable` · updated 2026-10-18 · canonical `/tutorial/14-superposition-and-supgen/`

14: superposition, collapse, and SupGen

This chapter is the one place in the tutorial where the code does not run on `bend`. It runs on the Interaction Calculus (IC), the calculus behind the `sup` and `dup` of chapter 06. Victor Taelin's reference implementation runs it on a CPU. Every IC program below was run with that binary, and the lines under each program are its real output.

WHAT IS VERIFIED HERE, AND WHAT IS QUOTED

- **Verified in this lab:** the rules, labels, and collapse results below, with their outputs and interaction counts (`WORK`). Reference build: `Interaction-Calculus` at commit `e9f0185`, default branch, CPU mode.
- **Quoted, not re-run:** the search results for SAT and program synthesis, the enumerator counts, and every SupGen claim. Each one names its gist or post.
- **Not verified:** wall-clock time and memory. The binary reports interaction counts and its own timing, and the timing does not compare across machines.

BUILD THE REFERENCE IMPLEMENTATION

```
git clone https://github.com/VictorTaelin/Interaction-Calculus.git
cd Interaction-Calculus && make
IC_BIN="$PWD/bin/ic" python3 bend-docs/scripts/check-tutorial.py bend-docs/src/content/tutorial
```

The checker runs `ic` blocks only when `IC_BIN` is set, and reports them as skipped otherwise.

THE CALCULUS IN SIX FORMS

Form	Syntax	Meaning
variable	x	used at most once (affine), but global: it may appear anywhere
erasure	*	a value that has been discarded
lambda	λx.t	a function

Form	Syntax	Meaning
application	`(f a)`	apply `f` to `a`
superposition	`&L{a,b}`	two alternatives `a` and `b` with label `L`
duplication	`! &L{x,y} = t; body`	splits `t` into two uses `x` and `y` with label `L`

Labels are numbers. The grammar is the one in the calculus's README.

ONE PROGRAM PER RULE

Each program is followed by the lines the binary prints. `WORK` is the number of interactions.

APP-LAM: beta reduction is one interaction.

```
(λx.x λy.y)
```

```
λ$a.$a
WORK: 1 interactions
```

APP-SUP: applying a superposition runs both alternatives.

```
(&0{λx.x,λy.y} λz.z)
```

```
&0{λ$a.$a,λ$b.$b}
WORK: 5 interactions
```

The argument `λz.z` is duplicated, so each alternative gets its own copy.

DUP-SUP, same label: the duplication routes the parts.

```
!&0{x,y}=&0{λa.a,λb.b}; (x y)
```

```
λ$a.$a
WORK: 2 interactions
```

DUP-SUP, different labels: the two commute.

```
!&0{x,y}=&1{λa.a,λb.b}; (x y)
```

```
&1{λ$a.$a,λ$b.$b}
WORK: 9 interactions
```

DUP-LAM and the erasures.

```
!&0{x,y}=λz.z; (x y)
```

```
λ$a.$a
WORK: 3 interactions
```

```
(* λx.x)
```

```
*
WORK: 1 interactions
```

```
!&0{x,y}=*; (x y)
```

```
*
WORK: 2 interactions
```

LABELS: SAME ROUTES, DIFFERENT COMMUTE

The README says a superposition "works like a pair", and a duplication of a superposition "works exactly like a pair projection" when the labels match. The two DUP-SUP programs show both cases. With equal labels the duplication takes the pair apart, and the 2 interactions are routing and the final application. With different labels the duplication cannot route the alternatives, so it duplicates each one. The superposition comes out with label 1, at a cost of 9 interactions.

Labels also decide which alternatives are paired when two superpositions meet. The next program is the AND function applied to two boolean superpositions. True is `λt.λf.t`, False is `λt.λf.f`, and `(a b F)` computes AND. First, with different labels:

```
((λa.λb.(a b λt.λf.f)) &0{λt.λf.t,λt.λf.f} &1{λt.λf.t,λt.λf.f})
```

```
&0{&1{λa.λb.a,λc.λd.d},λe.λf.f}
WORK: 18 interactions
```

Next, with the same label on both inputs:

```
((λa.λb.(a b λt.λf.f)) &0{λt.λf.t,λt.λf.f} &0{λt.λf.t,λt.λf.f})
```

```
&0{λa.λb.a,λc.λd.d}
WORK: 12 interactions
```

Equal labels pair the alternatives: the first of one input meets the first of the other, so the result has two leaves, True and False. Different labels give every combination, and AND throws away the cases that do not matter. When the first input is False, AND never looks at the second, so the second superposition is erased in that branch. The result has three leaves. Labels and erasure, not just the function, decide the shape of the answer.

HOW THIS SITS ON INTERACTION NETS

The calculus uses the same three cells as chapter 04, and gives every DUP a label. Read the text as cells:

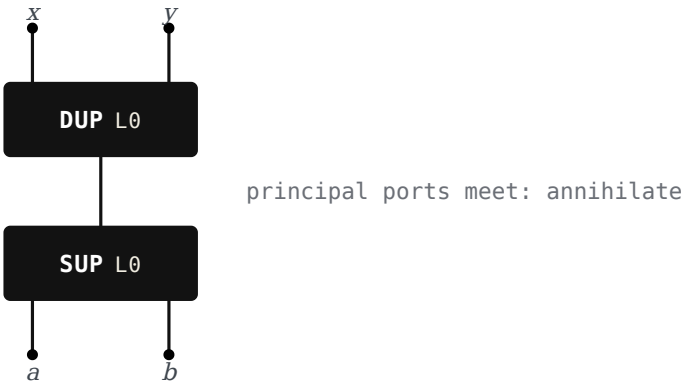
Text	Cell	Ports
`λx.t`	CON	principal: the lambda's output; aux: the variable `x` and the body `t`
`(f a)`	CON	principal: `f`; aux: `a` and the result
`!&L{x,y} = t; body`	DUP, label `L`	principal: `t`; aux: `x` and `y`; `body` is the rest of the net
`&L{a,b}`	DUP, label `L`, read from the other side	principal: the value; aux: the alternatives `a` and `b`
`*`	ERA	principal only

The rules are chapter 04's, with labels on the DUP cells. The counts are the ones the binary printed above:

Rule	In the net	Run	Interactions
APP-LAM	CON meets CON: β-reduction	`(λx.x λy.y)`	1
APP-SUP	CON meets a DUP: the application is copied	`(&0{λx.x,λy.y} λz.z)`	5
DUP-LAM	DUP meets CON: the lambda is copied, and its variable becomes a superposition	`!&0{x,y}=λz.z; (x y)`	3
DUP-SUP, same label	two DUPs with the same label annihilate; their aux wires join straight through	`!&0{x,y}=&0{λa.a,λb.b}; (x y)`	2
DUP-SUP, different labels	two DUPs with different labels commute	`!&0{x,y}=&1{λa.a,λb.b}; (x y)`	9
erasure	an ERA meets a CON or a DUP and erases it	`(* λx.x)`, `!&0{x,y}=*; (x y)`	1 and 2

A superposition is not a new kind of cell. It obeys the rules of a DUP with its label: it commutes with an application, annihilates with a duplication of the same label, and commutes with a duplication of another label. The README says a superposition "works like a pair". That describes what the cell holds, two alternatives. The cell itself is a DUP read from the other side (chapter 04 draws both readings).

With equal labels, the two cells meet at their principal ports and annihilate. Each aux wire goes straight through, so `x` is wired to `a` and `y` to `b`:



A superposition is born in DUP-LAM. A duplication that meets a lambda copies the lambda, and the lambda's variable becomes `&L{X0,X1}` with the duplication's own label `L`. The two copies of the variable are the alternatives, and the body is duplicated behind them. The count fits this reading: `!&0{f,g}=λx.x; f` takes 2 interactions, one DUP-LAM and one annihilation of the body's copy with the same label. A different label would commute instead, and take more steps.

The README also notes that without labels the system cannot clone pairs, or clone a cloned lambda. The labels are what let the net copy a superposition. None of this runs in Bend 2; the reference binary runs it.

COLLAPSE: FROM SUPERPOSITIONS TO PLAIN RESULTS

The reference binary has a collapse mode. `ic run FILE -C` rewrites a result so that no superposition or duplication remains inside its alternatives. The outermost superposition stays, and each alternative is a plain term. The first program in the previous section already shows this: its output is a tree of plain λ-terms, with the labels kept only at the top.

SEARCH BY SUPERPOSITION, WITH ERASURE AS THE FILTER

A selector turns a result into a filter. The selector `λr.(r A *)` returns `A` when `r` is True, because True takes its first argument, and erases everything else when `r` is False. Applied to the AND of the two superpositions, it keeps exactly the input pairs for which AND is True:

```
((λa.λb.((λr.(r λx.λy.x *)) (a b λt.λf.f))) &0{λt.λf.t,λt.λf.f} &1{λt.λf.t,λt.λf.f})
```

```
&0{&1{λa.λb.a,*},*}
WORK: 37 interactions
```

One alternative survives: `A` for the single pair (True, True). Every other alternative is erased to `*`. This is search in its smallest form. The program is written once, the candidates are a superposition, and erasure keeps only the candidates that pass the test. The SAT note calls this "search without loops". The search does no enumeration in the program text, and the filter does the selection.

EXERCISE: KEEP THE FALSE CASES

The selector `λr.(r * A)` keeps `A` when `r` is False, because False takes its second argument. Predict how many alternatives survive, before you run the program below.

Solution: two, not three. The pairs (False, True) and (False, False) reach the same branch of AND, and that branch never uses the second input. So the second superposition is erased in that branch, and only one alternative is produced there. Counting alternatives counts the results that reach the output, not the assignments you might expect.

```
((λa.λb.((λr.(r * λx.λy.x)) (a b λt.λf.f))) &0{λt.λf.t,λt.λf.f} &1{λt.λf.t,λt.λf.f})
```

```
&0{&1{*,λa.λb.a},λc.λd.c}
WORK: 37 interactions
```

EXERCISE: PREDICT THE LABEL

Predict the label of the result of `!&0{x,y}=&1{λa.a,λb.b}; (x y)`, and the number of interactions. Then run it.

Solution: the label is 1 and the count is 9. The duplication has label 0 and the superposition has label 1, so they commute, and the result is a superposition with label 1.

```
!&0{x,y}=&1{λa.a,λb.b}; (x y)
```

```
&1{λ$a.$a,λ$b.$b}
WORK: 9 interactions
```

SEARCH IN THE PUBLISHED MATERIAL

The author's public work applies the same idea to larger problems. These results are quoted, not re-run here.

- **Program search, add-carry** (gist `fast_dps_add_carry`, updated July 2026). A template with 16 unknown bits is written once, its candidates are superposed, two test cases are applied, and the result is collapsed. The gist reports about 36 thousand interactions for the search, against about 262 million for 65,536 separate runs of a conventional loop. It is written in HVM1 syntax. The author says the full source will not be released.
- **SAT** (gist `sat.md`, 2024). Each boolean is a superposition of True and False, a formula is applied to all assignments at once, and the result is collapsed. The gist's text reports 3 minutes in Rust against 1 second in HVM for a 32-variable instance, but the code in the gist is a 16-variable instance. The author writes that he knows little about the asymptotic behaviour.
- **Enumerating λ-terms** (gist `superposed_lambda_calculus_evaluator.c`, 2025). The enumerator builds affine λ-terms directly as a superposition of constructors such as `#Lam` and `#App`, so the candidate programs are data. The gist reports about 76 thousand interactions for an untyped search, against about 32 million for a naive enumerator, and about 3 thousand for a dependently typed version. Its header says it mirrors a file in HVM3's book. That path returned HTTP 404 when this chapter was checked, so the mirror was not confirmed.

SUPGEN: WHAT IS PUBLIC

SupGen is the synthesizer these techniques were built for. Its source is not public. The evidence:

- The DPS gist says: "We'll not release the full source, but, meanwhile, I can talk about the theory." It also says HOC "will soon (EOY?) launch an API for our DPS solution".
- A 2025 HOC thesis gist says: "SupGen, our program synthesizer that outperformed existing solutions by up to 1000x." That is a claim, and no code accompanies it.

- A post surfaced by web search, which I did not open, says SupGen "could only discover functions when given their type signatures" until a breakthrough. A YouTube demo is titled "SupGen: an AI-free code synthesis tool". Neither was checked against source.
- Taelin's GitHub account lists 261 public repositories, and none is named SupGen. HigherOrderCO's public repositories are Kind, HVM1 to HVM4, Bend1, and Bend2, which now lives at `bendlang/bend`. Other GitHub repositories named `supgen` are unrelated, except `BabySupGen`, a browser playground that describes itself as inspired by the research.
- The tutorial's own notes agree. The release-status note lists "Bend2–SupGen integration" among the launch blockers, and the HVM runtime note calls the integration "still private".

So the mechanism is public: the calculus, its labelled rules, collapse, and the examples above. The synthesizer is not.

IS A CHEAP FORK OF RUNNING CODE AVAILABLE?

Partly, and only inside the calculus.

- **Yes, as alternatives inside one term.** A fork of a function can be written as a superposition of two versions. Applying it to an input runs both versions, and the input is shared through a duplication. In the program below the main version is the identity, and the branch is NOT. One run returns both answers:

```
(&0{λx.x,λx.(x λt.λf.f λt.λf.t)} λt.λf.t)
```

```
&0{λ$a.λ$b.$a,λ$c.λ$d.$d}
WORK: 8 interactions
```

The result `&0{True, False}` holds the main version's answer and the branch's answer side by side.

- **Limits.** The fork is an alternative inside one term. It is not a copy of a running process with its own memory. Its cost is counted in interactions, which is the author's metric. Memory and wall-clock time were not measured here, and the speedups quoted above are interaction counts.
- **No, in released Bend 2.** The compiler rejects `&{a, b}` (chapter 06), and BendRT does not reduce nets (chapter 04). A Bend program compiles to a fixed binary, and a branch is a second definition that you test with laws (chapter 10).
- **The homoiconic angle.** The enumerator builds programs as data, using constructors such as `#Lam`. That is the closest public thing to code that other code can build and inspect. Released Bend 2 has no such representation. Chapter 10 shows a small `Expr` interpreter in Bend, which is a limited form of it.

WHAT TO WATCH FOR

- A Bend release that adds `sup` syntax or the SupGen integration. The search note in the tutorial's notes says to re-read it when SupGen ships.
- A published SupGen source or API. Until then, every SupGen claim in this chapter is a quoted claim.
- An HVM4 or IC release that reports wall-clock results on the programs above. The interaction counts here do not give those numbers.

NEXT

Chapter 13 is the glossary. Chapter 10 applies the fork idea to Bend's laws-first workflow.

A.16 · `bend-docs/src/content/tutorial/15-capstone.md`

Bend 2 tutorial 15: capstone, a checked command parser · `runnable` · updated 2026-10-18 · canonical `/tutorial/15-capstone/`

15: capstone, a checked command parser

This chapter builds one small program from the parts of the tutorial. It reads commands from the command line, moves a job through states, and proves properties of the transitions. The laws are the specification, and the gate checks that the code meets them.

The project has three files in one directory:

- `job.bend` : the program. Pure code comes first: types, a codec, and the transition function. The IO shell `main` comes last.
- `LAWS.bend` : the claims. A person writes them, and nobody edits them to make a proof pass (chapter 07).
- `PROOF.bend` : one proof per law. `bend PROOF.bend` is the gate.

1. THE THREE FILES

job.bend

```
import Base

type Cmd is Data:
  Start{}
  Cancel{}
  Finish{}

type State is Data:
  Queued{}
  Running{}
  Cancelled{}
  Finished{}

def encode(c: Cmd) -> String:
  match c:
    case Start{}:
      "start"
    case Cancel{}:
      "cancel"
    case Finish{}:
      "finish"

def only(hit: Bool, c: Cmd) -> Maybe<&2, Cmd>:
  match hit:
    case True{}:
      Some{c}
    case False{}:
      None{}

def decode(+s: String) -> Maybe<&2, Cmd>:
  Maybe.or(&2, Cmd, only(String.eq(s, "start"), Start{}), Maybe.or(&2, Cmd, only(String.eq(s, "cancel"), Cancel{}), only(String.eq(s, "finish"),
Finish{})))

def step(+s: State, c: Cmd) -> State:
  match s c:
    case Queued{} Start{}:
      Running{}
    case Queued{} Cancel{}:
      Cancelled{}
    case Running{} Cancel{}:
      Cancelled{}
    case Running{} Finish{}:
      Finished{}
    case _ _:
      s

def state_name(s: State) -> String:
  match s:
    case Queued{}:
      "queued"
    case Running{}:
      "running"
    case Cancelled{}:
      "cancelled"
    case Finished{}:
      "finished"

def run1(m: Maybe<&2, Cmd>) -> IO(Unit):
  match m:
    case Some{c}:
      IO.print(state_name(step(Queued{}, c)))
    case None{}:
      IO.die(Unit, 2, "unknown command")

def run2(m: Maybe<&2, Cmd>, n: Maybe<&2, Cmd>) -> IO(Unit):
  match m n:
    case Some{c} Some{d}:
      IO.print(state_name(step(step(Queued{}, c), d)))
    case _ _:
      IO.die(Unit, 2, "unknown command")

def dispatch(args: List<String>) -> IO(Unit):
  match args:
    case Nil{}:
      IO.print("usage: job CMD [CMD]")
    case Con{a, Nil{}}:
      run1(decode(a))
    case Con{a, Con{b, Nil{}}}:
      run2(decode(a), decode(b))
    case _:
      IO.die(Unit, 2, "too many commands")
```

```
def main() -> IO(Unit):
  do IO<Unit>:
    args : List<String> <- IO.args()
    dispatch(args)
```

Read the file from the top. `encode` and `decode` are inverses: `encode` writes a command as a string, and `decode` reads it back and returns a `Maybe`. `step` is the transition function, and every pair it does not list leaves the state unchanged. `dispatch` takes one or two commands, and `main` passes it the arguments. Each helper is defined above the function that uses it, so the file has no forward references (chapter 02).

LAWS.bend

```
import Base
import ./job.bend as J

law round_trip:
  for c: J.Cmd
  {J.decode(J.encode(c)) == Some{c} : Maybe<&2, J.Cmd>}

law finished_stays:
  for c: J.Cmd
  {J.step(J.Finished{}, c) == J.Finished{} : J.State}

law start_then_cancel:
  {J.step(J.step(J.Queued{}, J.Start{}), J.Cancel{}) == J.Cancelled{} : J.State}
```

`round_trip` says that every command survives encoding and decoding. `finished_stays` says that a finished job stays finished, whatever command arrives. `start_then_cancel` is a concrete claim that needs no case split: starting and then cancelling a job cancels it.

PROOF.bend

```
import Base
import ./job.bend as J
import ./LAWS.bend as Laws

def Laws.round_trip(c):
  match c:
    case J.Start{}:
      {==}
    case J.Cancel{}:
      {==}
    case J.Finish{}:
      {==}

def Laws.finished_stays(c):
  match c:
    case J.Start{}:
      {==}
    case J.Cancel{}:
      {==}
    case J.Finish{}:
      {==}

def Laws.start_then_cancel():
  {==}
```

```
All terms check.
```

`PROOF.bend` imports `job.bend`, so the gate also type-checks the program's `main`. Every proof here is a case split with `{==}` in each case, because each case computes to the same term on both sides.

4. RUN IT

The program is a command-line tool. These are the results of running it:

```
bend job.bend start
bend job.bend start cancel
bend job.bend start finish
bend job.bend
bend job.bend bogus
bend job.bend start bogus cancel
```

```
running
cancelled
finished
usage: job CMD [CMD]
unknown command        (exit status 2)
too many commands      (exit status 2)
```

5. BREAK IT ON PURPOSE

Each exercise has an answer that was checked by running the gate.

1. **A broken encoder.** Change the `"cancel"` string in `encode` to `"start"`. Predict the gate's first message.

   Answer: the gate fails in `LAWS.round_trip`:

   ```
   - expected : Some{job.Start{}}
   - observed : Some{job.Cancel{}}
   ```

   `round_trip` catches the bug because the claim names the behaviour that broke.

2. **A gap the laws do not cover.** Restore the encoder. Change `step` so that `Queued` plus `Cancel` stays `Queued`. Run the gate.

   Answer: the gate still prints `All terms check.` No law mentions that pair, so the broken transition passes. The laws are the specification, and the gate proves only what they say.

3. **Close the gap.** Add this law to `LAWS.bend` and this proof to `PROOF.bend`:

   ```
   law cancel_queued:
     {J.step(J.Queued{}, J.Cancel{}) == J.Cancelled{} : J.State}
   ```

   ```
   def Laws.cancel_queued():
     {==}
   ```

   Answer: the gate fails until `step` is fixed. It reports `expected : job.Queued{}` and `observed : job.Cancelled{}` at `LAWS.cancel_queued`. Restore `Cancelled{}` in `step`, and the gate passes again.

WHAT TO TAKE AWAY

A law names a behaviour, and the gate checks it. Passing the gate says nothing about behaviour that no law names. Write the claim first, keep it fixed, and let the proof change to match the code, not the reverse (chapter 07).

NEXT

Chapter 16 maps the repositories that assemble these pieces into real programs, and says which files to read first.

Continue to chapter 16.

A.17 · `bend-docs/src/content/tutorial/16-references.md`

Bend 2 tutorial 16: reading real Bend repositories · `docs-only` · updated 2026-10-20 · canonical `/tutorial/16-references/`

16: reading real Bend repositories

The tutorial teaches one idea per page. A repository teaches the next thing: how the pieces are arranged when the program is larger than one screen. The useful question is not which repository is biggest. It is where the boundary between pure code, effects, proofs, and measurement sits, and whether that boundary is visible in the file tree.

This chapter maps repositories worth reading. The list is not a ranking and it is not official. Each entry says what the repository demonstrates and which files to open first.

Core insight. A Bend repository is readable when its claims, its pure code, its effects, and its evidence live in separate places. Most of the repositories below differ in size, not in that shape.

THE SHAPE OF A BEND REPOSITORY

A small repository usually looks like this:

```
repo/
  README.md          the pinned compiler, how to run it, what it proves
  LAWS.bend          the claims, written by a human
  PROOF.bend         one same-named def per law
  main.bend          the entry point
  src/               pure domain modules
  tests/             golden files, unit tests, differential tests
  bench/             measurements, with the machine and version recorded
  host/ or *.c/*.js  foreign effects, kept at the edge
```

Larger repositories repeat the pattern per package or per layer:

```
repo/
  package-a/
    LAWS.bend
    PROOF.bend
    check.bend       runtime checks for the parts the checker cannot reach
    VERSION          the hub version, independent of its neighbours
    package-a.bend
    bench/
  package-b/
    ...
```

Path	What it holds	Tutorial
`LAWS.bend`	`law` declarations: the specification	07
`PROOF.bend`	`def` with the same name as each law	07
`check.bend`	ordinary tests, often for native arrays or effects	09
`src/`	pure functions, one concern per module	02
`tests/`	golden output and differential comparisons	01
`bench/`	timing, memory, and throughput runs	05
`*.c / *.js`	foreign effects behind a named signature	09

Not every repository has every row. A package with no effects has no `host/`. A teaching example may put everything in one file. What matters is that you can tell which row a file belongs to.

HOW TO READ ONE

Read in this order, and stop when you can predict the next file:

1. **README.** Find the compiler version. Bend 2 changes between releases, and a repository that pins 2.0.37 may not check on 2.0.25. Find the command that runs the proof gate.
2. **`LAWS.bend`.** Read the claims before the implementation. They tell you what the author thinks the program means.
3. **`PROOF.bend`.** Look at the shape of a proof, not every line: `{==}`, a rewrite, an induction hypothesis, or a call to a lemma.
4. **The entry point.** Find `main`, the CLI, or the test driver. Trace one path from it into one leaf module.
5. **One leaf module and its test.** This is where the writing style lives: how types are annotated, how matches are ordered, how effects are avoided.
6. **The build and the benchmark.** Check whether the numbers name the compiler, the thread count, and the machine. A number without those is an anecdote.

Useful questions to carry from repository to repository:

- Does `PROOF.bend` import the adjacent `LAWS.bend`? Without that import, the laws you read are not the ones the proof file proves.
- Is `check.bend` testing the same behaviour as the laws, or something the checker cannot express?
- Are foreign effects behind a small named module, or scattered through the domain code?
- Does the benchmark compare against another implementation, or only against itself?
- Does the README say what the repository does **not** prove?

A SHORT READING LIST

If you want to see	Read
the smallest complete laws-and-proofs project	metal-bending
a proved, benchmarked library	bend-collections
the boundary between a checker and a backend	bend-frontend
a pure game core with a thin driver	bend-craft
a large, deeply modular application	backplane
performance work with a proof obligation	elpis-fast
the package ecosystem	bend-packages

THE COMMUNITY REPOSITORIES

METAL-BENDING: THE SMALLEST COMPLETE PROJECT

Repository: AdrielSantana/metal-bending `(https://github.com/AdrielSantana/metal-bending)` · **Browser walkthrough:** adrielsantana.github.io/metal-bending `(https://adrielsantana.github.io/metal-bending/)`

A playground on Apple Silicon with seven single-concept examples, six graphics programs, and a working proof gate. The README is in Portuguese; the code is the common language.

Read it for: the smallest repository that still has the full shape. Each example is one file, and each file is one idea: `01_hello.bend` for `do`, `02_types.bend` for `type ... is Data`, `03_recursion.bend` for terminating recursion, `04_parallel.bend` for the fork shape, `05_arrays.bend` for one-owner arrays, `06_io.bend` for `IO.fork` and channels, `07_monads.bend` for `do` over `Maybe`.

Start at: `examples/02_types.bend`, then `LAWS.bend`. The file opens with `add_zero` and `append_nil` from Base, then states `total_cons` about the repository's own `src/math.bend`. That is the progression the tutorial describes: prove a Base fact, then prove a fact about your code. `gfx/01_mandelbrot.bend` is the same language doing a GPU-shaped loop in a window.

Composition lesson: one concept per file, one claim per law, and a gate the README shows failing as well as passing. The `gfx/` directory is separate from `examples/` because rendering is a different concern with a different driver.

BEND-VOXEL: AN ENGINE, AN APP, AND ACCEPTANCE SCRIPTS

Repository: aivv73/bend-voxel `(https://github.com/aivv73/bend-voxel)`

A voxel engine with destructible environments and a native window. The repository separates the editable body (`voxel.bend`), the renderer (`render.bend`), lighting (`lighting.bend`), materials (`material.bend`), and the DDA traversal (`dda.bend`) from the application shell (`atelier.bend`, `main.bend`).

Read it for: acceptance checks written in Bend rather than in a separate test framework. `scripts/reference_check.bend`, `scripts/lighting_check.bend`, `scripts/material_check.bend`, `scripts/image_check.bend`, and `scripts/ui_check.bend` are programs that compare complete outputs: cell sets, sun-depth texels, material exports, and CPU/GPU images. The `docs/` directory keeps the evidence, including the raw TSV and log files behind the DDA comparison.

Start at: `LAWS.bend` for the material and geometry invariants, then `voxel.bend` and `dda.bend`. Then read `scripts/reference_check.bend` to see how a test is just another Bend program with an exit condition.

Composition lesson: the engine is a library of pure modules, and every executable is a thin driver. The `Makefile` exposes `make test`, `make geometry`, `make lighting`, `make materials`, and `make images` so each layer of evidence can be run alone.

BEND-CRAFT: A DETERMINISTIC GAME CORE

Repository: costamatheus97/bend-craft (https://github.com/costamatheus97/bend-craft)

A block world with terrain, a first-person player, and a per-pixel voxel ray tracer. The whole game is U32 fixed point, so a run is a function of its seed and input log, and every frame is the same bytes on JavaScript, native C at one or many threads, and the GPU.

Read it for: the cleanest small example of the pure-core pattern. `world.bend` is the world as one flat `Array<U32>` and its terrain generator. `render.bend` is the Amanatides-Woo DDA and four shading tiers. `game.bend` is a pure 60 Hz `step` plus `render(state)`. `play.bend` is the only game file that reads a clock, a window, or a device; the benchmarks read them too.

Start at: the README's file list, then `game.bend` and `play.bend` side by side. The gap between them is the architecture. `tests/world.bend`, `tests/render.bend`, and `tests/game.bend` test the pure side; `run_tests.sh` runs them. `bench/` holds the scripts that produced the README's numbers, and `LOG.md` and `OPTIMIZATIONS.md` record what was tried and rejected.

Composition lesson: determinism is a design constraint, not a test added later. Keeping time and I/O in `play.bend` is what lets the same `step` run in a benchmark, a replay, and the window.

BENDOOM: A LARGE PORT, DECOMPOSED

Repository: eliesgalvira/bendoom (https://github.com/eliesgalvira/bendoom)

Doom's first level, E1M1, written in Bend 2. The README carries a "SLOP WARNING" and says there was no adult supervision of the code. Treat that as a warning about style, not about usefulness: the decomposition is the lesson.

Read it for: how to port a large C program without turning it into one file. `src/` has one module per concern: `wad.bend` for the archive, `bsp.bend` for the level tree, `render.bend`, `sim.bend`, `things.bend`, `sprites.bend`, `sfx.bend`, `opl.bend`, and `music.bend`. `doom.bend` is the game driver; `film.bend` replays a demo to video; `tests/` mirrors the module list with one test file per subject.

Start at: `LAWS.bend`, which states byte, WAD, key, sound, and simulation invariants, then `src/bytes.bend` and `src/sim.bend`. Read `nix/proof.nix` and `nix/tests.nix` to see how the proof gate and the test matrix are wired into CI.

Composition lesson: a port is a set of named modules plus a reference oracle. `tools/oracle.nu` compares frames against Chocolate Doom, so the port has an external definition of correct, not just an internal one.

ELPIS-FAST: SPEC, LAWS, PROOF, DIFF

Repository: gildrb/elpis-fast (https://github.com/gildrb/elpis-fast)

A 4-bit quantized Qwen inference engine with decode and prefill kernels, measured on one RTX 3090. It is dense and specialized; read it for the unit of work, not for a general application layout.

Read it for: the spec/laws/proof/diff discipline around each optimized kernel. A kernel such as `attn_chunk` has `attn_chunk.bend`, `attn_chunk_spec.bend`, `attn_chunk_laws.bend`, `attn_chunk_proof.bend`, and `attn_chunk_diff.py`. The `.bend` file is the fast implementation, `_spec` is the reference definition, `_laws` states their relationship, `_proof` proves it, and `_diff` compares against a trusted implementation. The `*_TABLE.bend` files are decision tables for scheduling and attention shapes.

Start at: `LAWS.bend` and `PROOF.bend` at the root, then one kernel family from the README table. The repository's "What fast means" section is as important as the code: a speedup is kept only if the output is unchanged or passes a named quality gate, and each gate records thresholds set before the runs.

Composition lesson: performance work is not exempt from specification. The spec stays readable, the optimized kernel stays fast, and the proof plus the differential test connect them.

BEND-COLLECTIONS: A LIBRARY WITH SPECS AND PROOFS

Repository: Giulio2002/bend-collections (https://github.com/Giulio2002/bend-collections)

Containers and cryptographic primitives in stock Bend, benchmarked against optimized C implementations of the same algorithms. The README has a table of containers and another for SHA-256, Keccak, BLAKE, AES-GCM, Argon2id, and the other crypto modules.

Read it for: how a real library separates the public module, the specification, and the proof. `src/containers/` holds the implementations (`dynamic_array.bend` , `deque.bend` , `balanced_search_tree.bend` , `hash_table.bend` , `lru.bend`). `spec/` holds reference definitions, `proofs/` holds the proof modules, and the root `laws.bend` , `laws_containers.bend` , `laws_crypto.bend` , and `laws_math.bend` group the claims by area. `benchmarks/bend/` runs the same workloads as the native C baselines.

Start at: `src/containers/deque.bend` and its proof under `proofs/containers/` , then `laws.bend` . The cryptographic modules show the same shape at a larger scale: a specification, a proved equality, and published test vectors.

Composition lesson: a container is not ”done” when it works. It is done when its invariants have a specification, a proof, and a measurement against the implementation it replaces.

BACKPLANE: A LARGE APPLICATION IN FLAT MODULES

Repository: i2cjak/Backplane `(https://github.com/i2cjak/Backplane)`

A large workspace for agentic hardware development, with deterministic KiCad and STEP viewers. It is a real product with a release process, not a sample.

Read it for: how a large Bend codebase stays navigable. `src/core/` is a flat set of roughly 140 small modules named after one concept each: `json.bend` , `codec.bend` , `semver.bend` , `sha256.bend` , `schematic.bend` , `pcb2d.bend` , `pcb3d.bend` , `model.bend` , `store.bend` . `src/app/` holds the view layer: `main.bend` , `layout.bend` , `render.bend` , `viewer.bend` , `theme.bend` , and one module per surface.

Start at: `src/app/main.bend` to see the entry point, then `src/core/json.bend` or `src/core/codec.bend` as a representative leaf. `LAWS.bend` and `PROOF.bend` sit at the root, so the proof gate covers the whole product with one command.

Composition lesson: flat names beat deep hierarchies when the names are specific. `src/core` is a vocabulary; `src/app` is a presentation of that vocabulary. Neither imports sideways for convenience.

BEND-FRONTEND: A CHECKER WITH A TYPED OUTPUT

Repository: ind-igo/bend-frontend `(https://github.com/ind-igo/bend-frontend)`

A composable frontend for full upstream Bend 2. It checks a program once and returns a typed, serializable `Core.Program` for a backend.

Read it for: the boundary between a language and its consumers. `src/core.bend` is the authoritative schema for the output. `src/decode.bend` turns upstream terms, patterns, declarations, and templates into that schema and rejects proof holes and missing `LAWS` imports. `src/frontend.bend` exposes `Frontend.check` and `Frontend.or_die` . `src/wire.bend` handles metadata numbers. `src/cli.bend` is a small driver. `host/adapter.js` and `host/run.js` attach the checker and I/O to the Bend process.

Start at: `src/core.bend` , then `src/decode.bend` , then `src/LAWS.bend` . The laws are about the output: `bare_idempotent` , `missing_laws_rejected` , `complete_accepted` , and `metadata_overflow_rejected` . That is a useful example of laws stated at an API boundary rather than about arithmetic.

Composition lesson: the IR is a schema, decoding is a pure step, the host adapter is the only JavaScript in the runtime path, and every backend consumes the same serializable value. `ind-igo/bend-evm` is a separate repository that does exactly that.

BEND-PACKAGES: A MAP OF THE ECOSYSTEM

Repository: lilalittle/bend-packages `(https://github.com/lilalittle/bend-packages)`

A curated, regularly refreshed overview of BendHub `(https://hub.bend-lang.com)`: top packages, source repositories, publishers, and an A–Z list. The snapshot in this chapter’s sources lists 703 packages on 2026-10-19.

Read it for: finding the next repository. The README groups picks by purpose: verified math and proofs, tensors and autodiff, web and networking, tools, games. Each row names the hub package, the version, the source repository when one is public, and the import line.

Start at: the ”Top picks” tables, then `ALL_PACKAGES.md` when you know the name you want. The README also explains the two import forms: a name and version such as `bend-mathlib@0.7.3.0/nat.bend` , which resolves to a content hash, and the raw hash form, which always resolves to the same code.

Composition lesson: packages are the unit of reuse. A repository that publishes several packages, such as `bend-kit` or `bend-ml` , shows how to keep each one independently versioned and independently proved.

EIGHT MORE REPOSITORIES

These are the repositories I would open next. They are grouped by what they teach, not by stars.

BENDLANG/BEND: THE OFFICIAL SOURCE

Repository: bendlang/bend (https://github.com/bendlang/bend)

The compiler, the `Base` library, the guide, the official demos, and the test suite. This is the authority the rest of the tutorial cites.

Read it for: `guide/GUIDE.md`, which `bend guide` prints; `bend2/base.bend`, which `bend base` prints; `demos/` for complete programs; and `tests/` for the exact syntax and error behavior the checker accepts. Read the official tests when a tutorial chapter and a community repository disagree.

Start at: `guide/GUIDE.md`, then `bend2/base.bend`, then one demo and the tests for the feature you are using. Do not copy from `Base` without checking the version: the standard library changes between 2.0.x releases.

LULZX/BENDC: A SELF-HOSTING COMPILER

Repository: Lulzx/bendc (https://github.com/Lulzx/bendc)

A compiler for Bend 2, written in Bend 2, that compiles itself and reproduces its own output byte for byte. The README reports a bootstrap fixpoint and 80 tests passing under both stages.

Read it for: how to structure a compiler as passes. `bendc.bend` is the driver, `check.bend` the type checker, `core.bend` the shared representation, and `boot.bend` the bootstrap path. Each optimization or transformation has a matching proof file: `LIT.bend` with `LITPROOF.bend`, `SUBST.bend` with `SUBSTPROOF.bend`, `INLINE.bend` with `INLINEPROOF.bend`, `DEAD.bend` with `DEADPROOF.bend`. `rt/` is the runtime, split by concern (`bendrt.h`, `native.c`, `cc.c`, `gpu.h`, `hub.c`).

Start at: `bootstrap.sh`, then `bendc.bend` and `check.bend`. Then read one small pass and its proof together. `tests/` pairs its test programs with `.out` golden files, and `tests/check/` is a numbered suite of checker errors.

Composition lesson: a compiler pass is a module with an input language, an output language, and a proof obligation. The proof file next to the pass makes that obligation visible.

PAYMOG/BEND-KIT: PACKAGES WITH THEIR OWN GATES

Repository: paymog/bend-kit (https://github.com/paymog/bend-kit)

A general-purpose library for Bend 2, split into packages such as `bytes`, `bignum`, `collections`, `concurrency`, `crypto`, `json`, `http`, and `cache`. Each package is published separately and has its own version.

Read it for: the package layout. The mature package directories contain `LAWS.bend`, `PROOF.bend`, `check.bend`, `VERSION`, the implementation (`bytes.bend`), and a `bench/` directory. `bytes/check.bend` is especially useful: it says in a comment that the laws run in the checker, where `Array` is a tree of defs, while `check.bend` runs the same operations on the runtime's native array at sizes the checker would not reach. That is a precise statement of what each gate covers.

Start at: `bytes/LAWS.bend`, `bytes/PROOF.bend`, and `bytes/check.bend`, then the README's "Same type" section. It explains why a caller must import the same package version the callee imports: a different version is a different type.

Composition lesson: one package equals one name, one version, one proof gate, and one benchmark. Dependencies are pinned by name and hash, not floating.

NUXYEL/BEND-ML: DEPENDENT TYPES IN A LIBRARY

Repository: nuxyel/bend-ml (https://github.com/nuxyel/bend-ml)

Machine learning in Bend 2 with tensor shapes in the type. A mismatched matrix product does not compile; a `reshape` needs a proof that the element count is preserved. Six packages are published on BendHub, with an MNIST demo and a GPT-2 demo.

Read it for: how dependent types shape an API. `tensor/` uses lists with `Vec<n>` and `Mat<r,c>`; `tensor-array/` uses a flat `Array<F32>` with the same type-level sizes and a `Bands<r,c>` type for parallel matrix-vector work; `autograd/` has a typed backward pass; `bpe/` proves the tokenizer round trip; `binary-read/` keeps foreign code out of the proved packages. `demos/gpt2` and `demos/mnist` are the integration tests.

Start at: the README's two shape-error examples, then `tensor-array/main.bend`, then `autograd/main.bend`. Read the package README before the code: it names the package version, its laws, and its dependencies.

Composition lesson: put the invariant in the type when the type checker can carry it, and in a law when it cannot. The shape is checked at the boundary, and every layer after that is checked for any shape.

ANGERMAN/BIRC: A PURE CORE WITH A THIN FOREIGN EDGE

Repository: angerman/birc (https://github.com/angerman/birc)

A terminal IRC client with a pure Bend core and a small C FFI for terminal rendering and POSIX sockets. The README reports more than 91% of the code in Bend, a zero-poll event loop, and a native DNS client.

Read it for: the actor-and-frame architecture. `src/bend/irc.bend` parses the protocol, `frame.bend` handles CRLF framing, `client.bend` and `buffer.bend` hold state, `submit.bend` builds outbound lines, `view.bend` lays out the screen, `session.bend` ties the pieces together, and `actor.bend` and `net.bend` do the I/O. `src/ffi/` is the thin C layer, and `src/ui/timui.h` is the rendering library.

Start at: `LAWS.bend` for the protocol and client invariants, then `src/bend/irc.bend` and `src/bend/frame.bend`. The `tests/pty/` directory is the part most projects skip: it drives the real terminal through a pseudo-terminal, including resize, paste, EOF, and UTF-8 split across reads.

Composition lesson: foreign code is acceptable when it is small, named, and at the edge. The domain stays testable because the protocol and state transitions are pure.

FABIANVEGAA/MYLSM: A SYSTEMS PROGRAM WITH PER-MODULE LAWS

Repository: FabianVegaA/mylsm (https://github.com/FabianVegaA/mylsm)

An experimental durable LSM key-value store. It writes to a log, keeps sorted tables, recovers after a restart, and exposes byte and UTF-8 views of the same database.

Read it for: law files per module. `src/` holds the storage modules (`ByteWal.bend`, `Wal.bend`, `ByteMemTable.bend`, `SstFile.bend`, and the rest), `laws/` holds a matching file for each (`laws/ByteWal.bend`, `laws/ByteMemTable.bend`, and so on), and `PROOF.bend` imports them into one gate. `app/cli/` and `app/repl/` are separate applications over the same library. `bench/smoke/`, `bench/crash/`, and `bench/workload/` cover ordinary paths, fault injection, and sustained runs.

Start at: `laws/ByteWal.bend` and `src/ByteWal.bend` together, then `app/cli/main.bend`. The README is unusually explicit about the limits: the laws cover pure decisions, not filesystem behavior, process locking, or the C and JavaScript host effects.

Composition lesson: a large systems program can keep its proof surface small by stating laws per module and importing them at the root. The `app/` layer is a client of the library, not part of it.

BENDLIB/BENDLIB: LEMMAS AND A LAW CHECKER

Repository: bendlib/bendlib (https://github.com/bendlib/bendlib)

A foundation library with `bend-mathlib` (published lemmas about `Nat`, `Bool`, `List`, `String`, `Maybe`, equality, and order) and `lawcheck`, a tool that looks for counterexamples before you try to prove a law.

Read it for: how to publish proofs as a library. `packages/bend-mathlib/` has one module per subject (`nat.bend`, `list.bend`, `bool.bend`, `maybe.bend`, `order.bend`, `sort.bend`, `perm.bend`), an `all.bend` entry point, and a README with a lemma index. `tools/lawcheck` is the complement of the proof gate: it tests the law against the implementation and shrinks a failing input.

Start at: `packages/bend-mathlib/README.md` for the lemma index, then `nat.bend` for the style. Read the `lawcheck` section of the README before writing a new law; a false law is cheaper to find by testing than by attempting a proof.

Composition lesson: proofs are reusable when they are packaged like code. The `all.bend` facade keeps imports stable while the individual modules stay focused.

777GENIUS/AWESOME-BEND: THE CURRENT INDEX

Repository: 777genius/awesome-bend (https://github.com/777genius/awesome-bend)

A curated list for current Bend 2, explicitly not Bend 1. It tracks the latest release, the hub, the official compiler, packages, tools, demos, benchmarks, papers, and community links.

Read it for: discovery and release context. The list records the changes between 2.0.x releases and warns which packages may need a newer compiler. The catalog section links to generated indexes with search and filters, which are better than a static list when you are looking for a specific package.

Start at: the "Packages", "Demos", and "Benchmarks" sections. Then check the repository's "Last checked" date; a list is only as current as its snapshot.

Composition lesson: an index is part of the ecosystem. It tells you which repositories are maintained, which are experiments, and which compiler version each one expects.

MORE TO EXPLORE

Repository	What to look at
nuxyel/llama.bend (https://github.com/nuxyel/llama.bend)	A Llama-architecture implementation checked token for token against PyTorch. `reference/` holds the Python oracle; `tests/bad_weight.bend` and `tests/bad_cap.bend` are negative tests.
rootagi/bend-json (https://github.com/rootagi/bend-json)	A JSON parser and serializer with `LAWS.bend`, `PROOF.bend`, an independent Python test suite, and four runnable examples.
ind-igo/bend-evm (https://github.com/ind-igo/bend-evm)	A backend that consumes `bend-frontend`'s `Core.Program`, lowers it to Yul, and carries a certificate plus per-contract laws.
CrocSwap/bendSVM (https://github.com/CrocSwap/bendSVM)	A Bend 2 to Solana SBF compiler with a shared SDK and several example contracts, each with its own `LAWS.bend` and `PROOF.bend`.
MatheusBBarni/bendJVM (https://github.com/MatheusBBarni/bendJVM)	A JVM interpreter organized as parser, verifier, linker, and runtime, with a differential test harness against the reference JVM.
nicolas-abril/bend2-libs (https://github.com/nicolas-abril/bend2-libs)	Many small packages in one repository (`bytes`, `deflate`, `png-decode`, `jpeg-decode`, `truetype`, `http-server`), each with its own entry and tests.
OkeyAmy/bend-voxel-bench (https://github.com/OkeyAmy/bend-voxel-bench)	An honest benchmark: a Luanti terrain port, a live race against the C++ original, and a `docs/bend-notes.md` list of the traps hit along the way.
jonathanperis/jonlib (https://github.com/jonathanperis/jonlib)	A raylib-compatible graphics and math library that replays upstream examples frame by frame against the native reference.

WHAT TO COPY, AND WHAT NOT TO

Copy these:

- **The gate.** `LAWS.bend` plus `PROOF.bend` at the root of the project, and a README command that runs it.
- **The pure core.** A domain module that takes values and returns values, with the clock, the window, the filesystem, and the network in a separate driver.
- **The named effect.** A `.c` or `.js` file behind one Bend signature, with the host boundary documented.
- **The matching test.** A `tests/` file for each domain module, and a `check.bend` for the parts the checker cannot reach.
- **The pinned version.** The compiler version in the README, and the package version or hash in the import.

Do not copy these:

- **A law that restates the implementation.** A claim such as "for every `x`, `f(x) == f(x)`" passes and says nothing. A law should name a behaviour a caller relies on.
- **A proof that hides a hole.** A `?TODO` or an `@unsafe` dependency makes the gate weaker. The tutorial's chapter 07 shows the messages.
- **A benchmark without a baseline.** A time in milliseconds with no machine, thread count, compiler version, or comparison is not evidence.
- **Foreign code as a shortcut.** If a `.c` file does the work, the Bend code is a wrapper, and the laws prove the wrapper, not the work.
- **A repository wholesale.** Bend 2 is moving. A repository pinned to 2.0.22 may use syntax that 2.0.37 rejects, and the reverse. Read its version first.

VERSIONS AND FRESHNESS

All repository links in this chapter were checked on 2026-10-20. The latest stable release named by `awesome-bend` on that date was Bend 2.0.37. The tutorial's own runnable blocks were checked with Bend 2.0.25, so a repository that requires a newer compiler may reject code that this tutorial accepts, and a repository that uses an older feature may need `bend update` or a pinned checkout.

When a repository and this tutorial disagree, the compiler in front of you decides. Run `bend version`, then run the repository's proof command. If the proof gate passes, read the code; if it fails, read the version notes before changing anything.

NEXT

Go back to the table of contents, or pick one repository and read it in the order above. The most useful next step is to take `metal-bending`'s shape, add one law about your own code, and run the gate.

PART B · Bend 2 site pages: learn + notes

The short hands-on and deep-note pages that sit beside the tutorial: hello/first-programs/parallelism/gpu/proofs/sharing/synthesis/hvm-hands-on (learn) and interaction-nets, hvm-runtime, four-interactions, optimal-computer, proof-lineage, search-by-superposition, release-status, what-is-bend2, vs-lean, vs-mojo (notes).

Source root: `bend-docs/src/content/learn · bend-docs/src/content/notes` · **21 files · 69.1 KB**

▼ In this part (file list)

- **B.1** · `bend-docs/src/content/learn/data-matching.md`
- **B.2** · `bend-docs/src/content/learn/effects-targets.md`
- **B.3** · `bend-docs/src/content/learn/first-programs.md`
- **B.4** · `bend-docs/src/content/learn/gpu.md`
- **B.5** · `bend-docs/src/content/learn/hello.md`
- **B.6** · `bend-docs/src/content/learn/hvm-hands-on.md`
- **B.7** · `bend-docs/src/content/learn/parallelism.md`
- **B.8** · `bend-docs/src/content/learn/proofs.md`
- **B.9** · `bend-docs/src/content/learn/sharing.md`
- **B.10** · `bend-docs/src/content/learn/synthesis.md`
- **B.11** · `bend-docs/src/content/learn/types-proofs.md`
- **B.12** · `bend-docs/src/content/notes/four-interactions.md`
- **B.13** · `bend-docs/src/content/notes/hvm-runtime.md`
- **B.14** · `bend-docs/src/content/notes/interaction-nets.md`
- **B.15** · `bend-docs/src/content/notes/optimal-computer.md`
- **B.16** · `bend-docs/src/content/notes/proof-lineage.md`
- **B.17** · `bend-docs/src/content/notes/release-status.md`
- **B.18** · `bend-docs/src/content/notes/search-by-superposition.md`
- **B.19** · `bend-docs/src/content/notes/vs-lean.md`
- **B.20** · `bend-docs/src/content/notes/vs-mojo.md`
- **B.21** · `bend-docs/src/content/notes/what-is-bend2.md`

B.1 · `bend-docs/src/content/learn/data-matching.md`

Data and matching · `docs-only` · updated 2026-09-05 · canonical `/learn/data-matching/`

Data and matching

Data definitions and the many shapes of `match`: algebraic data types, records, maps, pattern matching, recursion, and folds.

ALGEBRAIC DATA TYPES

Algebraic data types introduce constructor-shaped values such as `Nat`.

RECORDS

Records *(planned)*.

MAPS

Maps *(planned)*.

PATTERN MATCHING

The proofs sample matches `Nat` by constructors — brace-suffixed patterns, one arm per constructor:

```
def is_zero(n: Nat) -> Bool:
  match n:
    case Z{}:
      True
    case S{p}:
      False
```

Numeric matching works the same way (`case 0:` / `case 1+d:` , binding the predecessor).

LAMBDA MATCH

Lambda match *(planned)*.

RECURSION

Recursion is ordinary self-call, as in the parallel sum.

MUTUAL RECURSION

Mutual recursion *(planned)*.

HIGHER-ORDER FUNCTIONS

Higher-order functions *(planned)*.

CLOSURES

Closures *(planned)*.

FOLDS

Folds *(planned)*.

THE BEND CONSTRUCT

The `bend` construct that gives Bend its name awaits the public compiler.

B.2 · bend-docs/src/content/learn/effects-targets.md

Effects and targets · `docs-only` · updated 2026-09-05 · canonical `/learn/effects-targets/`

Effects and targets

Effects stay on the CPU in typed `do` blocks; pure code may fly to the GPU or compile to C, JavaScript, and Python.

IO

Effects run on the CPU in typed `do` blocks (hello) — `input` , `print` , sequenced with `<-` .

FILES

Files extend the same `IO` surface *(planned)*.

COMMAND-LINE ARGUMENTS

Command-line arguments extend the same `IO` surface *(planned)*.

COMPILING TO C

HVM4's ahead-of-time compiler targets native C *(current runtime tooling)*.

COMPILING TO JAVASCRIPT

Bend2's JavaScript target is planned.

COMPILING TO PYTHON

Bend2's Python target is planned. Flags, outputs, and the GPU story per target arrive with the toolchain — the one hard rule is already known: pure code may fly to the GPU, effects stay on the host.

B.3 · bend-docs/src/content/learn/first-programs.md

First programs · docs-only · updated 2026-09-05 · canonical /learn/first-programs/

First programs

Functions, bindings, branching, numbers, strings, tuples, lists, and imports — the first ten Bend2 concepts, in the order a new programmer meets them.

FUNCTIONS

```
import Base

# Adds two numbers.
def add(a: U32, b: U32) -> U32:
  a + b

def main() -> U32:
  add(20, 22)
```

`def` names the function, parameters carry types, the return type follows the arrow, and the body is an expression — no `return`.

LET BINDINGS

Tuple assignment splits two calls at once (the fork from parallelism). A single binding is the same shape:

```
def main() -> U32:
  x = 40 + 2
  x
```

CONDITIONALS

Branching goes through `match` on numbers and constructors.

```
def describe(n: U32) -> U32:
  match n:
    case 0:
      0
    case 1+d:
      1
```

BOOLEANS

Boolean constructors and operators *(planned)*.

NUMBERS

Numbers are typed as `U32`, including successor-style matching with `case 0:` and `case 1+d:`.

```
def identity(n: U32) -> U32:
  n
```

STRINGS AND CHARACTERS

Strings concatenate with `++` and print through `IO::print` (see hello). Characters and escaping are planned.

TUPLES

Tuples destructure in assignment position: `a, b = f(x), g(x)`.

LISTS

List syntax *(planned)*.

IMPORTS

`import Base` brings in the standard library, including `IO`.

B.4 · bend-docs/src/content/learn/gpu.md

Running on GPUs · `docs-only` · updated 2026-08-13 · verified 2026-09-05 · canonical `/learn/gpu/`

Running on GPUs

Bend2 runs a function on the GPU when the call carries `!`: `sum!(24, 0)` evaluates on the GPU where `sum(24, 0)` would use the CPU. The function is the parallel sum, unchanged — only `main` differs. That is the entire GPU programming model.

```
def main() -> U32:
  sum!(24, 0)
```

WHAT THE MARK DOES

`!` chooses **where** a call evaluates, not **how** it parallelizes. The fork inside `sum` is ordinary divide-and-conquer recursion, identical on either device; the GPU just points its thousands of threads at the same graph. Kernels, grids, blocks, and device memory do not appear in the language — the one-line contrast with Mojo from the comparison.

WHAT STAYS ON THE CPU

Pure code goes through `!`; effects stay on the host. Hello world runs its input and print in `IO` on the CPU — what flies to the GPU here is arithmetic with no `IO` in its type.

THE BACKENDS

Metal and CUDA, built on HVM4. No public toolchain ships them yet, so this page carries no measured numbers; they arrive when a compiler does.

B.5 · bend-docs/src/content/learn/hello.md

Hello world · `docs-only` · updated 2026-08-13 · verified 2026-09-05 · canonical `/learn/hello/`

Hello world

Every Bend2 program starts at `main`. This one reads a name and prints a greeting — and in doing so shows how Bend2 handles effects: not as statements that happen, but as values of type `IO` that describe what to do.

```
import Base

# Performs effects on the CPU.
def main() -> IO<Unit>:
  do IO<Unit>:
    name <- IO::input()
    IO::print("Hello, " ++ name)
```

LINE BY LINE

- `import Base` brings in the standard library, which supplies `IO`.
- `def main() -> IO<Unit>:` declares a function, Python-style, with the return type after the arrow. `IO<Unit>` is `IO` applied to `Unit`: a computation run only for its effects. (`main` need not be effectful — a pure program declares `def main() -> U32` and returns a number.)
- `do IO<Unit>:` sequences effects in a block that names its type.
- `name <- IO::input()` runs an effect and binds its result; `<-` is "run and name the answer".
- `IO::print("Hello, " ++ name)` is the block's value. Strings concatenate with `++`, library functions are reached with `::` paths, `#` starts a comment.

WHERE EFFECTS RUN

On the CPU. GPU evaluation — marking a call with `!` — is for pure functions; input and output stay on the host. That split is the whole story of the GPU page.

B.6 · bend-docs/src/content/learn/hvm-hands-on.md

HVM4 hands-on · `runnable` · updated 2026-09-05 · canonical `/learn/hvm-hands-on/`

HVM4 hands-on

This page is different from the Bend2 pages: everything here runs today. HVM4 is public, one C file, and the outputs below were produced by building it and running each snippet (gcc works; the README uses clang).

```
curl -sSL -o hvm.c https://raw.githubusercontent.com/HigherOrderCO/HVM4/main/src/hvm.c
gcc -O2 -o hvm hvm.c   # a few warnings, builds fine
./hvm file.hvm -s      # -s shows interaction/heap stats
./hvm file.hvm -s -C10 # -C collapses superpositions into separate results
```

FIRST RUN: A FUNCTION OF TWO ARGUMENTS

Books are `@name = term` lines; application is `f(a, b)` — never `f a b` (whitespace application is a parse error). Lambdas use `λ`:

```
@add = λa. λb. (a + b)
@main = @add(1, 2)
//3
```

Actual run: `3`, 3 interactions, 23 heap nodes. Three interactions for `1 + 2` tells you the price model honestly: every step is graph surgery.

SUPERPOSITION: ONE LOCATION, TWO VALUES

```
@main = (&A{1,2} + 10)
```

Actual run (`-C10`): `11` then `12`, 4 interactions, 13 nodes. Both values flowed through the same addition, sharing everything except the choice. Two corrections to the README here, found by running: the superposition needs a **label** (`&A{…}`, not `&{…}` — bare `&{}` is erasure), and there must be no spaces inside the braces.

LABELS DECIDE: ANNIHILATE OR MULTIPLY

A duplication meeting a superposition with the **same** label extracts pairwise; with **different** labels it multiplies branches:

```
@main =
  !x&A = &A{1, 2};
  [x₀, x₁]
//[1,2]          -- one result, 1 interaction
```

```
@main =
  !x&A = &B{1, 2};
  [x₀, x₁]
//[1,1]
//[2,2]          -- two results, 5 interactions
```

Both verified by running. `!x&A = v; body` binds the two copies as `x₀` and `x₁` (subscripts required); `[a, b]` is list sugar.

AFFINITY: THE COMPILER THAT SAYS NO TWICE

Variables may be used at most once. This fails:

```
@bad = λx. (x + x)   // PARSE/AFFINITY ERROR: x used twice
```

The fix is a cloned binder — `&` tells the parser to insert duplication:

```
@square = λ&x. (x * x)
@main = @square(5)
//25               -- verified: 3 interactions
```

Style rule from the primer, worth memorizing: dups are the expensive primitive, so arrange definitions as case-trees that avoid clones, and never apply an inline match (`λ{...}(x)`).

FIB, WITH A STOPWATCH

```
@fib = λ{
  0: 0
  1: 1
  λ&n. (@fib((n - 1)) + @fib((n - 2)))
}
@main = @fib(10)
//55
```

Actual run: `55` , 760 interactions, 2512 nodes, 19.33M interactions/s on an ordinary CPU. `0:` / `1:` switch on machine ints; the default arm names its binder (`λ&n.` — cloned, since `n` is used twice).

PATTERN MATCHING THAT ACTUALLY PARSES

The primer's `<>:` list-cons pattern does **not** parse on current main (`expected: name`). Match the constructor explicitly:

```
@len = λ{
  []: 0
  #CON: λh. λt. (1 + @len(t))
}
@main = @len([1, 2, 3])
//3                -- verified: 16 interactions
```

`[]` is `#NIL{}` sugar and still works; Peano naturals use `Nn` sugar with `0n:` / `1n+:` arms. Where this guide deviates from upstream docs, the binary was the tiebreaker — every snippet above exited 0.

B.7 · bend-docs/src/content/learn/parallelism.md

Automatic parallelism · `docs-only` · updated 2026-08-13 · verified 2026-09-05 · canonical `/learn/parallelism/`

Automatic parallelism

Bend2 parallelizes ordinary recursion: split the work in two, and the runtime evaluates the halves concurrently. No pragmas, no threads, no scheduler in the program. The program below sums 2^{24} numbers this way.

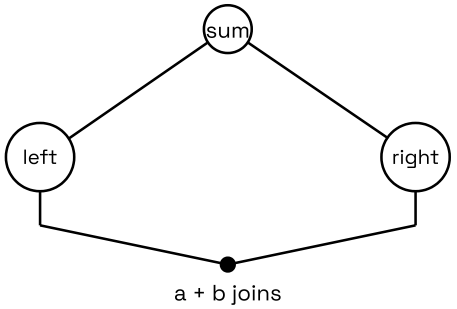
```
import Base

# Sums a range of numbers in parallel.
def sum(d: U32, i: U32) -> U32:
  match d:
    case 0:
      i
    case 1+d:
      a, b = sum(d, i*2), sum(d, i*2+1) # fork
      a + b

# Runs sum on the GPU, via `!`.
def main() -> U32:
  sum!(24, 0)
```

THE FORK

Bend2 forks where the data flow splits. `a, b = sum(d, i*2), sum(d, i*2+1)` starts both recursive calls, and nothing joins them until `a + b` needs both results. The calls share no state, so on the interaction-net runtime they are disjoint subgraphs — recall the model: disjoint pairs cannot conflict — and any idle core reduces either one without locks.



The fork: one call becomes two disjoint subgraphs; *a + b* is the only join.

THE MATCH

Bend2 pattern-matches on numbers with a successor case. `case 0:` handles zero; `case 1+d:` matches anything ≥ 1, binding `d` to one less (shadowing the argument). A branch body is an expression — no `return` anywhere. The arguments are `U32`, 32-bit unsigned integers — full 32-bit range.

WHAT IT COMPUTES

Each call owns a block of 2^d consecutive integers and splits it in half; at depth 0 the block is the single number `i`. So `sum(24, 0)` covers 0 through 16,777,215, for an exact total of 140,737,479,966,720 — which notably does not fit in a `U32`. Device selection (`!`) is a separate decision from parallelism; that is the GPU page.

B.8 · bend-docs/src/content/learn/proofs.md

Proofs · `docs-only` · updated 2026-08-21 · verified 2026-09-05 · canonical `/learn/proofs/`

Proofs

This page proves one fact end to end: addition commutes. If the previous page was the theory, this is the lab session — one theorem, one proof, every step shown. A Bend2 theorem is an `assert` block, and its proof is an ordinary `def` carrying the same name: the claim, then the evidence. Here the claim is that addition commutes, and the argument is the standard Peano one, induction on the first argument.

```
import Base

# Theorem: "for all numbers a and b, a + b equals b + a".
assert add_comm:
  for all a : Nat
  for all b : Nat
  {add(a, b) = add(b, a) : Nat}

# Proof: case analysis, induction, rewrites (`%`).
def add_comm(a, b):
  match a:
    case Z{}:
      %add_zero(b)
      {=}
    case S{p}:
      %add_succ(b, p)
      %add_comm(p, b)
      {=}
```

THE STATEMENT

Propositions are types. `for all a : Nat` (two words) quantifies over the naturals; `{add(a, b) = add(b, a) : Nat}` is an equality between `Nat` expressions, braces around the proposition. The `assert` names the obligation; the `def` of the same name discharges it, parameter types supplied by the assert.

THE PROOF

Case analysis plus rewriting. `match a:` splits on `Nat`'s constructors — `Z{}` zero, `S{p}` successor, brace-suffixed patterns. Each `%` line rewrites the goal with a named equation: `%add_zero(b)` and `%add_succ(b, p)` apply lemmas from `Base`; `%add_comm(p, b)`, the recursive call, **is** the induction hypothesis. `{=}` closes a branch once rewriting made both sides identical.

FOLLOWING THE GOALS

Track each branch's goal and the `%` lines read plainly. Zero case: goal `add(Z, b) = add(b, Z)`; left reduces by definition, `%add_zero(b)` rewrites `add(b, Z)` to `b`, `{=}` closes `b = b`. Successor case: goal `add(S(p), b) = add(b, S(p))`; left reduces to `S(add(p, b))`, `%add_succ(b, p)` turns the right into `S(add(b, p))`, the induction hypothesis equates them for `{=}`.

NEXT TO LEAN

Term-level proofs, Agda register: recursion as induction, no tactic layer (`simp`, `omega`) in sight. Whether that trade pays off is the comparison.

B.9 · bend-docs/src/content/learn/sharing.md

Sharing, duplication, superpositions · `docs-only` · updated 2026-09-05 · canonical `/learn/sharing/`

Sharing, duplication, superpositions

Sharing one value between two uses is where Bend2's cost model lives: the fork shares nothing, duplication is explicit, and superpositions branch. Runtime claims link to runnable HVM4.

The one idea The fork shares nothing (disjoint subgraphs), so it is free. Sharing one value between two uses needs explicit duplication — and duplicated functions meet superpositions, which is where HVM's power lives.

SHARING AND DUPLICATION

```
def main() -> U32:
  x = 40 + 2      # one value...
  x + x           # ...two uses: this is the duplication point
```

Bend2's types are linear by default (at most one live use per binder; duplication written `+x`, only `Data` kinds duplicable). Where the checker demands it, shared values duplicate explicitly rather than by accident — the language-level face of the runtime primitive.

UNSCOPED LAMBDAS

Unscoped lambdas exist in the runtime today (`λ$x` binders — run them in HVM4 hands-on). Their Bend2 surface syntax *(planned)*.

SUPERPOSITIONS

Superpositions are deliberately introduced later, in HVM4 Runtime 9.4. Their Bend2 surface syntax *(planned)*.

EVALUATION ORDER

Evaluation needs no annotations — order is the scheduler's business, and confluence keeps every schedule correct.

MEASURING SPEEDUP

Measuring speedup is a release-day exercise: same program, CPU call vs `!` call, with the runtime's interaction counters as the stopwatch *(planned)*.

B.10 · bend-docs/src/content/learn/synthesis.md

Synthesis with SupGen · `docs-only` · updated 2026-09-05 · canonical `/learn/synthesis/`

Synthesis with SupGen

SupGen turns precise types into programs: holes mark the unfinished, specifications set the target, and search fills the gap.

The one idea Precise types give synthesis a search target and the checker an acceptance condition. SupGen proposes implementations; the type checker disposes. See search by superposition for the machinery that makes each guess cheap.

HOLES

A hole marks an unfinished program the synthesizer should fill — the `?name` of Kind's grammar, expected in Bend2's surface *(planned)*.

TYPES AS SPECIFICATIONS

"Sorted list", "parser round-tripping with its printer": types precise enough to pin behavior down. Writing them is already proving half the program; the rest is search.

SUPGEN

Named on the March 2026 launch-blocker list, demonstrated nowhere public. Interface, examples, and limits arrive with the release — this page will be rewritten from the compiler.

B.11 · bend-docs/src/content/learn/types-proofs.md

Types and proofs, step by step · `docs-only` · updated 2026-09-05 · canonical `/learn/types-proofs/`

Types and proofs, step by step

From annotations to proofs: how Bend2 types carry values, state equalities, and discharge obligations by recursion. The full worked proof is on the proofs page.

WHAT IS A TYPE-BASED PROVER?

Every compiler checks your code a little. A type-based prover checks it all the way down — not just "is this a number?" but "is this claim true for every possible input?"

Three analogies, from familiar to exact:

A blueprint inspector. A function signature is a promise about shape: takes two numbers, returns a number. A normal type checker verifies the shape. A prover verifies the *behavior written into the shape*: takes any two numbers, returns their sum regardless of order. The checker is an inspector who refuses to sign off until the building matches the blueprint — before anyone moves in, i.e. before the program ever runs.

A fact-checker, not a spell-checker. Testing runs examples: it tries a million pairs and finds no counterexample. Proving covers the cases you didn't try — including the one your test suite missed. A test says "it worked every time we looked"; a proof says "there is nowhere for a bug to hide."

Claims are types; evidence is code. This is the whole trick, called *propositions as types*. The type `{add(a, b) = add(b, a) : Nat}` is the claim "addition commutes." A program with exactly that type *is* the evidence — it can only exist if the claim holds, the way a key can only exist if it fits its lock. Writing the proof means constructing that program; checking it means the compiler trying the key. In Bend2 you state the claim with `assert` and deliver the evidence with an ordinary `def` of the same name. No separate proof language, no tactic scripts: the proof is a program, and the checker simply runs its type discipline over it.

TYPE ANNOTATIONS

Every posted `def` annotates parameters and results (`a: U32` , `-> U32` , `-> IO<Unit>`).

POLYMORPHISM

Polymorphism — functions generic over types *(planned)*.

DEPENDENT TYPES

Types can mention values: the proofs sample quantifies `for all a : Nat` and states equality `{add(a, b) = add(b, a) : Nat}` as a type. Sized lists ("a list of exactly this length") and equality types are the same idea pushed further *(planned)*.

SIZED LISTS

Sized lists *(planned)*.

EQUALITY

Equality appears in the public theorem sample as `{add(a, b) = add(b, a) : Nat}` .

INDUCTION

Induction in Bend2 is recursion: the `add_comm` proof calls itself as the induction hypothesis, and each `%` rewrite is checked.

TOTALITY

The guarantee that functions terminate, which is what makes proofs sound, is enforced by the Live linear mode *(planned)*.

B.12 · bend-docs/src/content/notes/four-interactions.md

The four interactions · `stable` · updated 2026-09-05 · canonical `/notes/four-interactions/`

The four interactions

The interaction-net model says rewrites are local and confluent. The Interaction Calculus says which four rewrites. Everything else in HVM — numbers, constructors, pattern matching — is practical extension. The theory below comes from HVM4's own `docs/theory/interaction_calculus.md` ; the traces are quoted from it.

WHY LAZY NEEDS TWO NEW FORMS

Strict evaluation computes everything now; lazy computes on demand but recomputes shared values used twice. Haskell memoizes thunks, which works for data and breaks inside lambdas. The calculus instead makes sharing structural, with two dual primitives (Lafont's fan nodes, opposite polarities):

- **Duplication** `!x&L = v; t` — one value in two locations (x_0 , x_1).
- **Superposition** `&L{a, b}` — two values in one location.

Duplication is incremental: cloning `[1,2,3]` peels one layer per step, and the tail's duplication only fires when both copies are touched. Work is never duplicated — even inside lambdas.

THE FOUR RULES

Two create computation, two propagate it. Superpositions are introduced here, after the language and basic runtime foundations:

SUPERPOSITIONS

Superposition `&L{a, b}` puts two values in one location. It is the runtime primitive behind branching search and shared evaluation; start with the HVM4 hands-on examples before treating it as a programming technique.

Two create computation, two propagate it:

1. **APP-LAM** — application eliminates lambda: `(λx.body)(arg)` binds `x ← arg` in `body` . Ordinary β.
2. **DUP-SUP** (same label) — duplication eliminates superposition: `!x&L = &L{a,b}; t` becomes $x_0 \leftarrow a$, $x_1 \leftarrow b$ in `t` . Pair projection.
3. **APP-SUP** — application propagates through superposition: `(&L{a,b})(c)` becomes `!x&L = c; &L{a(`x_0`), b(`x_1`)}` . Applying a pair distributes over both elements.
4. **DUP-LAM** — duplication propagates through lambda: `!f&L = λx.body` yields two lambdas with a **superposed bound variable**, `x ← &L{…}` , and the body shared behind a new dup. This is the deep one: the variable temporarily escapes its scope, which is exactly why the calculus needs global variables — and exactly how sharing extends inside lambdas.

With different labels, DUP-SUP commutes instead of annihilating, nesting one superposition inside the other (the cross product from the hands-on page, formalized).

OPTIMAL SHARING, ON PAPER

Duplicate a lambda containing `(2+2)` , apply the copies to different arguments — the addition computes **once** and the result `4` flows to both copies. The trace (abridged from the theory doc) pivots on one step: DUP-LAM shares the body behind `b` , OP2-NUM fires a single time marked "shared!", and both branches observe `4` . Memoization could never do this: the shared work sits inside a duplicated function body.

CHURCH 2^2 IN 14 INTERACTIONS

Applying Church-2 to itself is the literature's sharing benchmark. The full trace runs APP-SUP, DUP-LAM, and DUP-SUP (both modes) across a network where bound variables visibly escape scope mid-computation — then collapses to `λf.λx.f(f(f(f(x))))` . HVM completes it in **14 interactions**, which is optimal. The point for Bend2: this is the machinery that makes "the evaluator schedules" more than a slogan.

COLLAPSE: READING THE ANSWER OUT

Reduction has three destinations: WNF (head only), SNF (full, keeps SUPs/DUPs), CNF (collapsed back to pure lambda terms via DUP-VAR/DUP-APP cleanup). The `-C` flag selects collapse — the enumeration of `11` , `12` you saw hands-on. Proofs about programs are proved against SNF; humans read CNF.

ONE WORD PER TERM

The machine guest of honor: every term is one 64-bit word — `TAG (8) | EXT (24) | VAL (32)` . Tags name the node kind (APP, LAM, SUP, DUP, NUM, constructors by arity, MAT/SWI, OP2, REF…); EXT carries the dup label, op code, or binder level; VAL is a heap slot or an unboxed `u32` .



One HVM term: a single 64-bit word. Numbers live unboxed in VAL.

When a binder fires, its slot is overwritten with a substitution cell (SUB bit set) so linked variables resolve without side tables. Static book definitions stay immutable with de Bruijn levels; ALO nodes expand them lazily, one layer at a time, preserving sharing. That is the whole memory story — and why "a cell is a tagged machine word" from the [model](./interaction-nets.md) page is literal, not metaphor.

B.13 · bend-docs/src/content/notes/hvm-runtime.md

The HVM4 runtime · `stable` · updated 2026-09-05 · canonical `/notes/hvm-runtime/`

The HVM4 runtime

If interaction nets are the idea, HVM4 is the machine: public, a single C file built with one compiler invocation, with its memory layout and interaction rules documented next to the code. It is the half of Bend2 you can read today — the language, the type checker, and SupGen remain private, but the calculus, the memory layout, and the rules are inspectable, and they constrain what the language above them can be: parallel by default, superposition-capable, compiled rather than interpreted.

Run it yourself Every claim below with a number next to it was reproduced by building HVM4 and running the snippet — see HVM4 hands-on. The repository carries no license text, so treat this as study, not a grant to build on it.

THE CALCULUS UNDERNEATH

HVM4 implements the Interaction Calculus: affine variables (each used at most once — which makes garbage collection cheap and parallelism simple), global lambdas with no scope boundaries (enabling continuations and linear encodings), and two first-class primitives most runtimes lack:

- **Superposition** `&A{a, b}` : one term holding two values. `(&A{1,2} + 10)` yields both `11` and `12` in 4 interactions, sharing all work that does not depend on the choice. This is how an evaluator explores many branches of a search space while paying for the common structure once.
- **Duplication** `!x&A = v; body` : one value feeding two uses (x_0 , x_1) without copying up front — the incremental copy described on the interaction nets page. Same label annihilates against a superposition; different labels multiply branches.

```
@main = (&A{1,2} + 10)
//11
//12
```

METAL AND CUDA BACKENDS

HVM4 ships both GPU backends. The CUDA port was produced overnight by a coding model working against the Metal implementation as executable specification and test oracle; its benchmark puts CUDA ahead of Metal on RTX hardware at roughly 10x parallel C for most programs.

WHAT THE AOT COMPILER CHANGES

An interpreted evaluator pays dispatch on every rewrite: fetch the node, branch on its kind, mutate the graph. Compiling a program's rules ahead of time to native C removes the dispatch and specializes memory access per rule — attacking exactly the constant factor that lets a tuned CPU core rival GPU throughput on many workloads. The AOT compiler was one of three items on the March 2026 launch-blocker list, alongside the GPU runtime (since resolved) and SupGen integration (still private).

9.1 RUNTIME MODEL

An HVM program is a **book** of named terms. A definition such as `@main = ...` is a reusable static term. Evaluation instantiates the term into a mutable graph, then repeatedly finds an active pair: two principal ports connected together.

Each interaction consumes a small local configuration and replaces it with another configuration. The evaluator does not walk the entire program to decide what is safe to run. Locality and confluence make that decision independent for every active pair.

```
clang -O2 -o hvm src/hvm.c
./hvm file.hvm -s       # interaction statistics
./hvm file.hvm -s -C10  # collapse alternatives
./hvm file.hvm -D       # print intermediate steps
```

9.2 SURFACE SYNTAX

The runtime language is deliberately smaller than Bend2. Definitions use `@name = term`; application uses parentheses, not whitespace application: `@add(1, 2)`, not `@add 1 2`.

```
@add = λa. λb. (a + b)
@main = @add(1, 2)
//3
```

Form	Meaning
`λx. body`	Lambda abstraction
`(f x)` or `f(x)`	Application
`#Pair{a,b}`	Constructor
`λ{#Pair: ...}`	Constructor match
`!x&A = value; body`	Explicit duplication with label `A`
`&A{left,right}`	Superposition with label `A`
x_0, x_1	The two branches of a duplicated value
`&{}`	Erasure
`@name`	Book reference

Variables are affine: an ordinary variable may be used at most once. When the same value must be used twice, write a cloned binder (`λ&x`) or an explicit duplication. This is what lets the evaluator represent copying as graph interaction rather than a global memory operation.

9.3 THE 64-BIT TERM

Every term is represented by one 64-bit word:

```
+--------+------------------------+--------------------------------+
| TAG 8  | EXT 24                 | VAL 32                         |
+--------+------------------------+--------------------------------+
```

`TAG` identifies the term kind. `EXT` carries metadata such as an operation code, constructor identity, duplication label, or binder level. `VAL` contains an immediate number or an index into the heap/book.

Dynamic terms live in the mutable heap; static definitions live in the immutable book. ALO allocation terms bridge them: a static term expands into a dynamic term only when evaluation forces that layer, preserving compact definitions and sharing.

9.4 EVALUATION AND COLLAPSE

HVM distinguishes the amount of a term it evaluates:

- **WNF** — weak normal form; expose the head shape.
- **SNF** — strong normal form; reduce the whole term while preserving superpositions and duplication nodes.

- **CNF** — collapsed normal form; enumerate ordinary readable results.

Collapsing is a presentation step, not the same thing as evaluation. A superposition is not automatically a list, and collapsing it too early can destroy the sharing that made the computation cheap.

9.5 GPU BOUNDARY

HVM4 has Metal and CUDA backends. The GPU does not change the interaction rules; it changes where independent graph work is scheduled. Bend2 keeps input/output on the host and sends pure computation to a device call marked with `!`.

Read next: The four interactions, HVM4 hands-on, and Search by superposition.

B.14 · bend-docs/src/content/notes/interaction-nets.md

Interaction nets, explained · `stable` · updated 2026-09-05 · canonical `/notes/interaction-nets/`

Interaction nets, explained

Every runtime problem that matters is a copying problem. Using a value twice means duplicating its computation or its data; throwing it away means reclaiming the memory. Conventional machines solve both with coordination — locks, reference counts, stop-the-world collectors — and coordination is what stops programs from scaling. Linear logic's great export was making this explicit: treat values as resources consumed exactly once, and copying becomes a first-class operation you can see, price, and control. Interaction nets (Lafont, POPL 1990) are the machine that takes that lesson literally.

The big insight: no consensus Ten thousand workers can rewrite one program with zero coordination — no locks, no tags, no collector — because the model makes conflicting steps *unrepresentable*. A step touches exactly two adjacent cells, so two steps never share a cell, so no interleaving can go wrong. Consensus isn't optimized away. It is unnecessary by construction.

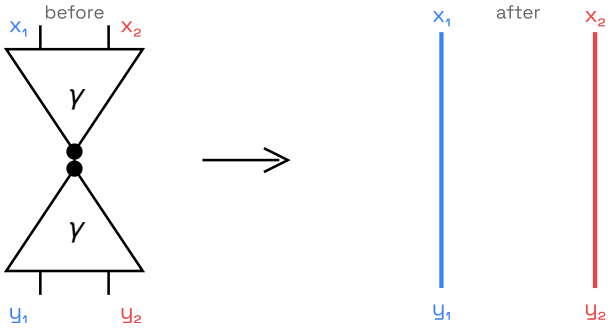
WHY COORDINATION IS UNNECESSARY

- **No coordination.** Redexes are disjoint before any analysis runs; any idle worker fires any pending pair. Parallelism is scheduling, not a language feature.
- **Copying is incremental and local.** Duplication copies one cell per step, interleaved with the computation consuming the copies — never a whole structure in one gulp. Shared subgraphs stay shared until use forces the split.
- **Erasure is garbage collection.** Throwing a value away is just another local rewrite (ε), with no collector thread pausing anything.
- **Every step is atomic.** Each rule completes in constant time with a clear physical mapping — unlike substitution, which can copy an unbounded term.
- **Perfectly symmetric.** No rule has an active side and a passive side: two cells meet as equals through their principal ports. The whole system is meetings, never commands.
- **Three symbols suffice.** γ, δ, ε under six rules compute anything — the smallest universal machine with all of the above intact.

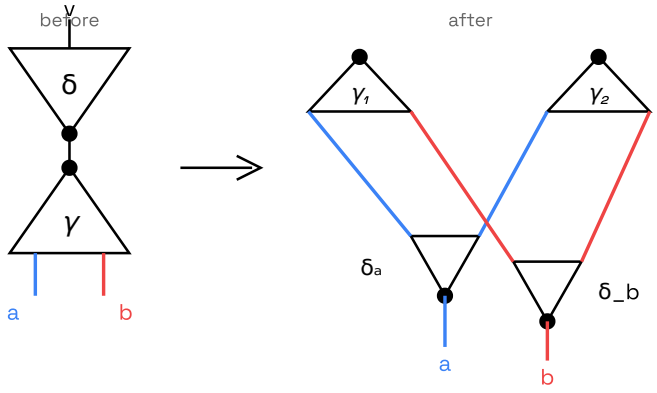
COPYING, ERASURE, AND CONFLUENCE

A program is a graph, and sharing means what it says: two parents point at one child. When both parents need the child, a duplicator (δ) walks it — one cell per interaction, copying each cell and moving on while the rest of the net keeps reducing around it. Nothing stops, nothing locks, and work common to both copies happens once, before the split. Contrast the alternatives: textual substitution duplicates unevaluated work blindly; memcpy-style cloning drags in unused substructure; reference counting serializes every touch. Here the copy *is* the computation, interleaved cell by cell.

The dual move is annihilation: when a symbol meets itself, both vanish and the wires join through — erasure and pairing handled by the same geometry. In the drawings below, matching colors mark the same wire continued across the step. The richer interaction between duplication and alternative values belongs to the advanced runtime chapter; first understand the local graph and its mathematical guarantees.



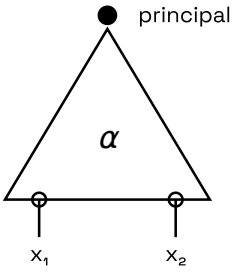
annihilate Same meets same: both cells vanish, and each auxiliary wire splices straight through to its counterpart — blue continues to blue, red to red. Nothing is copied, nothing observed.



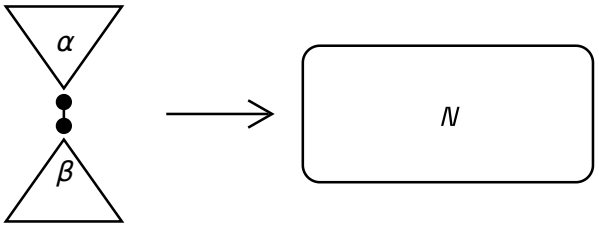
commute Distinct symbols pass through each other: γ is copied once per δ branch and δ once per γ wire. Follow the colors — each original wire (blue *a*, red *b*) gets its own duplicator, and each γ copy takes one copy from each.

THE MODEL IN ONE MINUTE

A program is a graph of **cells**. A cell is one constructor or operation — a number, a `+`, a list node, a λ. It has one **principal port** (drawn at the apex, the only place it can react) and any number of auxiliary ports where arguments and results wire in. Computation happens in exactly one configuration: two cells wired principal-port to principal-port, called an **active pair**. For each pair of symbols there is at most one rule, which consumes the two cells and writes a fixed replacement on the same free ports.



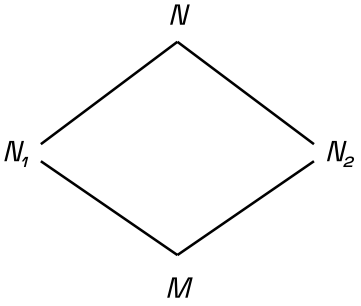
A cell: symbol α, one principal port at the apex, auxiliary ports below.



A rule firing: the active pair α ⋈ β is consumed and the fixed net N takes its place.

THE TWO THEOREMS THAT CARRY EVERYTHING

Confluence (1990). Fire two different active pairs in either order and both orders land on the same net in one step — the pairs share no cell, so firing one leaves the other intact.



diamond One-step confluence: firing a then b, or b then a, lands on the same net M.

Consequences: every order reaches the same result (no scheduler can be wrong), every route takes the same number of steps (work is a property of the program, so profiling counts mean something), and any idle worker may fire any pair it finds.

Universality from three symbols (1997). γ δ ε under six rules suffice for anything computable — and every rule is one of the two symmetric meetings above. That is the whole machine: meet, annihilate, commute, repeat — at any scale, in any order, with no one in charge.

THE Λ-CALCULUS, WITH FINE PRINT

Encode abstraction and application as cells and β-reduction becomes a single constant-time interaction. The difficulty moves to duplication: Lamping's POPL 1990 algorithm answered optimally (never duplicate work in Lévy's sense), and Asperti and Mairson proved in 1998 that full optimality's bookkeeping is unbounded — optimal counts β-steps but does not promise cheap accounting. The HVM line draws the engineering conclusion: duplication is a **primitive** of the calculus, not transparent substitution, trading Lamping-completeness for constant factors.

The mystery, solved Taelin's "mystery" post asked why some functions run *faster the more work you give them* — e.g. N repeated applications in O(log N). Answer: functions that **fuse under self-composition** (their composed normal form stays constant size) can be exponentiated by squaring, exactly like integers — *provided* the runtime normalizes inside lambdas, which only optimal evaluators do. A `copy` prefix can even *make* a function fuse, yielding apparent "negative complexity". Fusion at runtime is also why β-optimality gives Haskell-style deforestation for free.

IN THE MACHINE

A cell is a tagged machine word, an active pair is a queue entry, an interaction is a few loads and stores — none of it locked, because disjointness plus confluence keeps every schedule correct. The price is the constant factor: every step is graph surgery against real memory, and grinding it down is what the runtime generations are for. Numbers meet the same wall: a pure net would write 1000 as a thousand cells, so real runtimes store machine integers inside the node word, and the width left after ports and tags sets the range.

B.15 · `bend-docs/src/content/notes/optimal-computer.md`

Towards an optimal computer · `stable` · updated 2026-09-05 · canonical `/notes/optimal-computer/`

Towards an optimal computer

This page distills the founding essay of the Higher Order Company. It is the "why" behind every runtime, language, and proof system in these notes — read it before anything else if you want the motive, not just the mechanism.

The one claim Turing machines and the λ-calculus are equivalent in *computability* — but a model can be inherently less efficient than another. Church–Turing says nothing about *computation*.

1936, TWICE

Humanity answered "what is computation" twice in the same year. Turing distilled the common components of early machines into one universal device — the ancestor of every processor, and, through the procedural mindset, of C, Fortran, Java, Python. Church, independently, distilled the common components of branches of mathematics into the λ-calculus — the ancestor of Haskell, Clojure, Agda. Both compute everything computable. The choice between them was treated as taste. Taelin's argument: that was the wrong question. The right one is which model wastes less.

WHAT EACH MODEL GETS WRONG

The procedural model's parallel primitives — mutexes, atomics — are a contrived solution to synchronization, so programmers still write sequentially by default; global state and loops make whole classes of bugs nearly unavoidable. The functional model handles parallelism and correctness far better in principle — yet functional programs stay mostly single-threaded. Why? Performance: the λ-calculus's fundamental operation, **substitution, is not atomic**. Substituting an argument can copy an unboundedly large term, so it cannot complete in bounded steps and has no physical mapping. Workarounds only move the problem — shared references inhibit parallelism, garbage collection isn't atomic. The functional paradigm never got its efficient machine, never went mainstream, and its proof tools never caught up.

ANNIHILATION AND COMMUTATION

In 1997 Lafont proposed the interaction combinators, where substitution breaks into two truly atomic laws: **commutation** (creates and copies information) and **annihilation** (observes and destroys it). Unlike SKI — whose K erases and whose S copies unboundedly large structures — every combinator step finishes in constant time with a clear physical mapping, and all of it is inherently parallel. Everything good in the other models survives; the pathologies mostly vanish. Both the Turing machine and the λ-calculus emulate efficiently onto combinators, while the reverse is not true — equivalent in computability, strictly more capable in computation. Perhaps, the essay suggests, machines and substitutions aren't fundamental at all, and some alien civilization does mathematics and computing entirely in annihilation and commutation.

Where this goes The atomic-step claim becomes the interaction-net model; the efficiency claim becomes the four interactions; the company built four runtimes and two languages on top. Each page tests one corner of this essay.

B.16 · `bend-docs/src/content/notes/proof-lineage.md`

Where the proofs come from · `stable` · updated 2026-09-05 · canonical `/notes/proof-lineage/`

Where the proofs come from

Prerequisites

- What is Bend2? — the language this lineage leads to

Bend2's dependent types did not appear from nowhere. They are the third generation of one author's argument: **proofs are a developer-productivity tool**, an evolution of types the way TypeScript was an evolution of JavaScript. Understanding the lineage makes Bend2's strangest choices (linear everything, Dead vs Live, synthesis-first) read as conclusions rather than quirks.

PROOF-ORIENTED PROGRAMMING

Proofs are not only a way to certify mathematics. In HOC's tradition they are a programming tool: precise specifications make generated programs checkable, so synthesis can propose and the kernel can reject incorrect answers.

FORMALITY: PROOFS AS SUPERPOWERS

Formality (Taelin, ~2018, "a modern programming language featuring formal proofs — now written in itself") opens with the thesis outright: forget mathematics and security, the boring stuff — adding types to untyped languages already demonstrated the productivity jump, and formal proofs are the next step of that same curve. The language is there to make the developer faster, not the mathematician happier. Every Bend2 proof decision descends from this sentence: in the Lean comparison, the program is the product.

Taelin's essay on proving vs testing makes the mechanism explicit: auditing doesn't scale, tests sample behavior, proofs cover it — and the cheapest proof is one the machine helps write. That is the exact slot SupGen occupies in Bend2: synthesis proposes, the checker disposes.

KIND: THE MINIMAL THEORY

Kind ("a modern proof language") is the visible ancestor of Bend2's type system — and it is deliberately tiny. Its core grammar fits on one screen: dependent function `∀(x:T)U`, lambda, application, self-types, datatypes with constructors, pattern matching, holes (`?name`) for unfinished proofs, and `*` as the type of types. No tactic language, no standard library ceremony: just enough theory to state and check precise types.

Kind2's conversion checker in three lines To decide *A == B*: if textually identical, true; else reduce both to weak normal form and check similarity field-by-field. No "seen equations" map (Kind1's slower fix), no constructor flags — just identical → reduce → similar, in exactly that order. It covers everything Coq's checker covers, and the order-sensitivity is load-bearing: swap it and self-referential types (like *List Char* where *Char* aliases *#U60*) loop forever.

That design moral — the obvious algorithm works if reductions happen in precisely the right order — runs through all of HOC's work, from evaluators to checkers. Bend2 inherits the shape (quantifiers as types, programs as proofs) and adds the linearity discipline plus the Dead/Live mode split that lets it keep `Type : Type` without letting paradoxes near live code.

WHAT THIS PREDICTS ABOUT BEND2

Three predictions, all checkable at release: proofs will stay term-level (recursion as induction, rewrites as `%`, no tactic layer — tactics serve mathematicians, and Bend2 serves developers); the standard library will grow specifications first (precise types are SupGen's search targets); and the killer feature will read as productivity, not certainty — proofs replacing test suites rather than chasing foundations. If release-day Bend2 instead ships a tactic engine, this page is wrong in an interesting way, and will say so.

B.17 · bend-docs/src/content/notes/release-status.md

Release status · `prerelease` · updated 2026-08-27 · verified 2026-09-05 · canonical `/notes/release-status/`

Release status

No. As of 2026-08-27, Bend2 has not been publicly released. The current estimate is approximately 2026-09-02.

THE DATED TRAIL

- **2026-03-05**: launch-blocker list — HVM4 AOT compiler, HVM4 GPU runtime, Bend2–SupGen integration.
- **2026-07-18**: "Bend2 complete"; CUDA backend ported overnight from the Metal reference.
- **2026-08-12**: delay post (≈08-28 implied).
- **2026-08-27**: launch plan — copiable-kind fix, weekend testing, Monday presentation recording, possible Wednesday launch (≈09-02).

WHAT "RELEASED" WILL MEAN HERE

A public compiler (or repository with code), a license, and runnable examples. Until all three exist, every Bend2 sample on this site stays labeled documentation-only. This site is unofficial and not affiliated with Higher Order Company.

B.18 · bend-docs/src/content/notes/search-by-superposition.md

Search by superposition · `docs-only` · updated 2026-09-05 · canonical `/notes/search-by-superposition/`

Search by superposition

Superposition is usually introduced as "two values in one location" with a two-line arithmetic demo. Its serious use is different: **search without loops**. Apply a function to a superposition of its entire domain and you get a superposition of its entire image — with the evaluator sharing work across all guesses at once.

Status The experiments below ran on earlier HVM syntax and are **documentation-only** here: shapes and numbers as reported by Taelin, not re-run. The mechanism (first-class SUP + collapse) is unchanged in HVM4.

TRUTH TABLES FOR FREE

Evaluating `(And {True False} {True False})` yields the complete truth table — `{{True False} {False False}}` — because applying a function to a superposed domain produces the superposed image. So brute force stops being written as nested loops:

```
# instead of this:
for x0 in [T, F]:
    for x1 in [T, F]:
        ...
        print(F(x0, x1, ...))
# write this:
print(F({T,F}, {T,F}, ...))
```

The second form shares computation across guesses wherever the guesses agree — which, for structured problems, is nearly everywhere.

SAT: 3 MINUTES IN RUST, 1 SECOND IN HVM

Taelin's one-line-style SAT solver feeds all 2^{16} assignments of a random 16-variable 3-SAT instance as superposed booleans, lets the formula evaluate once across the whole superposition, then collapses per-label results back into the satisfying assignment. A 32-variable instance: ~3 minutes of Rust brute force vs ~1 second superposed. The honest caveat is his own: this was a new observation, asymptotics unknown — but the shape of the win (shared prefixes across guesses) is exactly what the machinery predicts.

PROGRAM SEARCH: 65,536 CANDIDATES, 36K INTERACTIONS

The Discrete Program Search series pushes the same idea up a level: synthesize the add-carry function from a template with 16 unknown bits. Conventionally that's 65,536 separate runs (~262M interactions at ~4k per call). Superposed — one template holding all candidates, two test cases, collapse — it returns the bit vector `[0,1,1,0,1,0,0,1,1,1,1,0,1,0,0,0]` in **36k interactions**: under one interaction per guess, a ~7200x win that grows with the search space. Enumeration and loops are fully replaced by superposition and collapse; the follow-ups extend it to DSL enumeration and superposed λ-terms.

WHY THIS PAGE EXISTS IN BEND2'S NOTES

Bend2's SupGen — synthesis from precise types — is this technique aimed at programs instead of bits: types give the search a target, the checker an acceptance test, and superpositions make each guess cost less than one. The DPS posts even sketch the product shape (examples in, function out). When SupGen ships, re-read this page first.

B.19 · bend-docs/src/content/notes/vs-lean.md

Bend2 vs Lean · `prerelease` · updated 2026-08-30 · verified 2026-09-05 · canonical `/notes/vs-lean/`

Bend2 vs Lean

Prerequisites

- Proofs — the `add_comm` sample both sides prove
- Where the proofs come from — Formality, Kind, and the productivity thesis

Both live on the propositions-as-types side of Curry–Howard: a specification is a type, a proof is a program, checking is type checking. What differs is which side pays the bills.

LEAN: THE PROOF IS THE PRODUCT

Lean 4 (de Moura, Lean FRO, Apache 2, self-hosted through C) exists to formalize: mathematics, verified algorithms, specifications. Its center of mass is mathlib — a half-million-item library built by hundreds of contributors, so new proofs import lemmas instead of rebuilding them. Automation means tactics: decision procedures, simp sets, `omega`, a decade of accumulated machinery. Execution is efficient sequential C with task-level concurrency; nobody chooses Lean because their proof needs a GPU.

BEND2: THE PROGRAM IS THE PRODUCT

Bend2's types exist to make machine-generated code checkable: synthesis (SupGen) proposes, the checker disposes. Its ancestor is Kind, Higher Order Company's proof language. The type theory is linear with no hidden exceptions: a plain binder allows at most one live use, duplication is written `+x`, only `Data` kinds may be duplicated. The checker splits in two — a Dead mode where specifications live unrestricted and may diverge, and a Live mode where code and proofs are linear and terminating — which keeps paradox-shaped terms on the Dead side despite `Type : Type` (what Girard's paradox makes inconsistent as a logic; Lean buys the same safety with a universe hierarchy plus termination checker).

The one public proof (addition commutes, August 2026) is term-level, Agda register: the function's own recursion is the induction, lemmas apply as `%` rewrites, `{=}` closes goals. No `simp`, no `omega`, no visible tactic layer at all.

THE SAME THEOREM IN LEAN 4

For comparison, here is how a Lean user proves the same fact today — tactics orchestrating library lemmas, in about four lines:

```
-- Illustrative: shows the Lean style, not typechecked for this page.
theorem add_comm' (a b : Nat) : a + b = b + a := by
  induction a with
  | zero => rw [Nat.zero_add, Nat.add_zero]
  | succ n ih => rw [Nat.succ_add, Nat.add_succ, ih]
```

Read the two side by side. Lean's version says *how to search*: `induction` picks the principle, `rw` fires named equations from the library, and the library (`Nat.zero_add`, `Nat.add_succ`, hundreds of thousands of siblings in mathlib) does the heavy lifting. Bend2's version *is* the search result already: the recursion is the induction, each `%` line is one rewrite step, `{=}` checks the endpoints match. No search language exists because, in Bend2's story, the machine (SupGen) performs the search and the human reads the trace. Which side of that trade pays off — tactic scripts over a giant library, or machine-found term proofs over precise types — is the open bet.

Watch it run in the infoview, goal by goal — input and output in one transcript, OCaml-toplevel style:

```
⊢ a + b = b + a
-- induction a with
-- ┌ case zero, ⊢ 0 + b = b + 0
-- │   rw [Nat.zero_add] → ⊢ b = b + 0
-- │   rw [Nat.add_zero] → no goals
-- └ case succ, ⊢ (n + 1) + b = b + (n + 1)
-- rw [Nat.succ_add] → ⊢ (n + b) + 1 = b + (n + 1)
-- rw [Nat.add_succ] → ⊢ (n + b) + 1 = (b + n) + 1
-- rw [ih] → no goals
```

A couple of remarks:

- Every rewrite names a lemma that must already exist — in mathlib, or proved just above. Bend2's `%` lines work the same way over `Base`.
- The `ih` in the last step *is* the induction hypothesis, exactly where Bend2 puts the recursive call `%add_comm(p, b)`. Same proof, two costumes: search script vs. found term.

THE DEFAULT TODAY

Lean, for anything needing a proof now — verified algorithms, formalized specs, mathematics. The library asymmetry is not close: 500,000+ items versus no public Bend2 library. Bend2 joins proofs to programs that run on HVM4 and to synthesis from precise types — a combination nobody else is selling, and nothing you can download.

B.20 · bend-docs/src/content/notes/vs-mojo.md

Bend2 vs Mojo · prerelease · updated 2026-08-30 · verified 2026-09-05 · canonical /notes/vs-mojo/

Bend2 vs Mojo

Prerequisites

- Automatic parallelism — the fork both sides argue about
- Running on GPUs — what ! means before comparing it to kernels

Both want high-level GPU code. They assign the scheduling job to opposite actors: in Mojo, **you** schedule; in Bend2, **the evaluator** does.

MOJO: YOU ARE THE SCHEDULER

Mojo (Modular, designed under Chris Lattner on MLIR, shipped and pre-1.0 with production users) gives explicit control: you write kernels against a grid/block/device-memory model, manage buffers and synchronization, use explicit SIMD types, and one source compiles to NVIDIA, AMD, and Apple targets. Ownership and borrowing govern values. That control is exactly what makes dense, regular, arithmetic-heavy workloads fast — performance there lives in tiling, layout, and occupancy, and no runtime discovers a tiling for you. If your sentence starts "I would have written CUDA", Mojo is today's answer for a Python-shaped codebase.

BEND2: THE EVALUATOR IS THE SCHEDULER

Bend2 programs are ordinary recursive functions with no kernels, no grids, no memory hierarchy in the programmer's model. Recursion over a tree fans out because the branches are disjoint subgraphs — not because anyone scheduled it. The entire device decision is one character: `sum!(24, 0)` runs on the GPU where `sum(24, 0)` would not. Each reduction step still moves graph nodes through memory — the constant factor every graph-reduction benchmark argues about.

WHERE EACH WINS

Mojo wins anywhere you could name the kernel: inference serving, image pipelines, dense linear algebra, HPC stencils — with full-width IEEE numerics and explicit SIMD today. Bend2's bet is everything irregular and recursive, where writing the kernel is the hard part and the evaluator can find work you never named. Until it ships, that side of the table is a design claim with a runtime (HVM4) behind it, not a benchmark you can run.

B.21 · bend-docs/src/content/notes/what-is-bend2.md

What is Bend2? · prerelease · updated 2026-08-30 · verified 2026-09-05 · canonical /notes/what-is-bend2/

What is Bend2?

Bend2 is an unreleased programming language from Higher Order Company. Its one-sentence idea: **write ordinary functional code, get parallelism, proofs, and even synthesized programs for free** — because the runtime evaluates your program as a graph where independent pieces cannot interfere, so scheduling needs no annotations from you.

Three pieces combine here. First, functional code that parallelizes automatically: recursion that splits work runs on all cores without threads, locks, or pragmas. Second, a dependent type system in the tradition of Higher Order Company's Kind: types can state properties like "this list is sorted", and the checker rejects code that violates them. Third, program synthesis (called SupGen): types precise enough to pin down behavior give a searcher something to aim at and the checker something to accept or reject.

WHY THIS COMBINATION MATTERS

Each piece exists elsewhere on its own. Array languages parallelize; Lean and Kind have dependent types; research languages synthesize code. The bet is that together they multiply: precise types make generated code checkable, so a synthesizer can propose and the checker can dispose — and everything runs on a runtime that parallelizes without being asked. The HVM runtime schedules independent graph rewrites; it never needed annotations and still does not.

WHAT THE LANGUAGE LOOKS LIKE

Bend2 reads like Python with types. A function is `def` with typed parameters and a return type after the arrow. Effects run in `do` blocks over an explicit `IO` type. A call runs on the GPU when its name carries `!`. A theorem is an `assert` block whose proof is an ordinary `def` of the same name. Each construct gets its own page under Learn by Example.

STATUS: WHAT ACTUALLY EXISTS

As of 2026-08-30: the public `VictorTaelin/Bend2` repository is empty — no code, no branches, no binary, no package, no license. What is public: four posted code samples (walked through in Learn by Example) and the HVM4 runtime. Taelin's estimate at last check points around 2026-09-02; track the release status page, which is updated as the public record changes.

PART C · Bend 2 skills (.opencode → bend2*, verbatim)

The bend-gme project's routable knowledge skills for Bend 2: a router skill (bend2) plus bend2-language, bend2-proofs, bend2-ownership, bend2-superposition, bend2-effects-targets, bend2-performance, bend2-app and bend2-runtime. Each is a SKILL.md plus reference chapters; they were distilled from the same sources as Parts A/B/H but are maintained separately, so they drift slightly and add project-specific facts (e.g. the JS lane, hub pinning).

Source root: `bend-gme/version-1/.opencode/skills` · **49 files · 384.8 KB**

▼ In this part (file list)

- **C.1** · `bend-gme/version-1/.opencode/skills/bend2/SKILL.md`
- **C.2** · `bend-gme/version-1/.opencode/skills/bend2/references/01-tutorial-index.md`
- **C.3** · `bend-gme/version-1/.opencode/skills/bend2/references/02-what-is-bend2.md`
- **C.4** · `bend-gme/version-1/.opencode/skills/bend2/references/03-hello.md`
- **C.5** · `bend-gme/version-1/.opencode/skills/bend2/references/04-release-status.md`
- **C.6** · `bend-gme/version-1/.opencode/skills/bend2/references/05-glossary.md`
- **C.7** · `bend-gme/version-1/.opencode/skills/bend2-language/SKILL.md`
- **C.8** · `bend-gme/version-1/.opencode/skills/bend2-language/references/01-first-programs.md`
- **C.9** · `bend-gme/version-1/.opencode/skills/bend2-language/references/02-data-and-recursion.md`
- **C.10** · `bend-gme/version-1/.opencode/skills/bend2-language/references/03-syntax-reference.md`
- **C.11** · `bend-gme/version-1/.opencode/skills/bend2-language/references/04-rules-and-errors.md`
- **C.12** · `bend-gme/version-1/.opencode/skills/bend2-language/references/05-first-programs-concepts.md`
- **C.13** · `bend-gme/version-1/.opencode/skills/bend2-language/references/06-data-matching-concepts.md`
- **C.14** · `bend-gme/version-1/.opencode/skills/bend2-proofs/SKILL.md`
- **C.15** · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/01-laws-and-proofs.md`
- **C.16** · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/02-dependent-types-and-totality.md`
- **C.17** · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/03-proofs-concepts.md`
- **C.18** · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/04-types-and-proofs-concepts.md`
- **C.19** · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/05-proof-lineage.md`
- **C.20** · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/06-bend-vs-lean.md`
- **C.21** · `bend-gme/version-1/.opencode/skills/bend2-ownership/SKILL.md`
- **C.22** · `bend-gme/version-1/.opencode/skills/bend2-ownership/references/01-ownership-and-copying.md`
- **C.23** · `bend-gme/version-1/.opencode/skills/bend2-ownership/references/02-sharing-concepts.md`
- **C.24** · `bend-gme/version-1/.opencode/skills/bend2-superposition/SKILL.md`
- **C.25** · `bend-gme/version-1/.opencode/skills/bend2-superposition/references/01-duplication-superposition.md`
- **C.26** · `bend-gme/version-1/.opencode/skills/bend2-superposition/references/02-superposition-and-supgen.md`
- **C.27** · `bend-gme/version-1/.opencode/skills/bend2-superposition/references/03-branching-and-self-change.md`
- **C.28** · `bend-gme/version-1/.opencode/skills/bend2-superposition/references/04-synthesis-concepts.md`
- **C.29** · `bend-gme/version-1/.opencode/skills/bend2-superposition/references/05-search-by-superposition.md`
- **C.30** · `bend-gme/version-1/.opencode/skills/bend2-effects-targets/SKILL.md`
- **C.31** · `bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/01-io-files-and-targets.md`
- **C.32** · `bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/02-effects-targets-concepts.md`
- **C.33** · `bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/03-js-target-facts.md`
- **C.34** · `bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/04-minebender-notes.md`
- **C.35** · `bend-gme/version-1/.opencode/skills/bend2-performance/SKILL.md`
- **C.36** · `bend-gme/version-1/.opencode/skills/bend2-app/SKILL.md`
- **C.37** · `bend-gme/version-1/.opencode/skills/bend2-app/references/01-capstone.md`
- **C.38** · `bend-gme/version-1/.opencode/skills/bend2-app/references/02-reading-real-repos.md`
- **C.39** · `bend-gme/version-1/.opencode/skills/bend2-runtime/SKILL.md`
- **C.40** · `bend-gme/version-1/.opencode/skills/bend2-runtime/references/01-interaction-nets.md`

- **C.41** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/02-parallelism-and-runtime.md
- **C.42** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/03-parallelism-concepts.md
- **C.43** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/04-gpu-concepts.md
- **C.44** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/05-hvm-hands-on.md
- **C.45** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/06-interaction-nets-notes.md
- **C.46** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/07-four-interactions.md
- **C.47** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/08-hvm-runtime-notes.md
- **C.48** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/09-optimal-computer.md
- **C.49** · bend-gme/version-1/.opencode/skills/bend2-runtime/references/10-bend-vs-mojo.md

C.1 · bend-gme/version-1/.opencode/skills/bend2/SKILL.md

Bend 2

A strict, checked functional language whose values are affine, whose laws are proved by the checker, and which compiles to C, native code or JavaScript.

MODEL

```
source.bend ─ checker (types, ownership, termination, laws) ─► OK | Error
            ─ BendRT compiler ─► C / native binary / JS module (-o out.c | out | out.js)
```

- Values are **affine**: a binder is used at most once unless its kind allows copies (`+n`). Ownership is a type-level fact, not a runtime one.
- **Laws** are types. A `law` is proved by a same-named `def` ; the checker runs the proof. A failed proof is a compile error.
- **Strict** evaluation. No laziness, no exceptions, no network, no DOM.
- Honest limits (`references/01-tutorial-index.md` , "Honest limits"): released Bend 2 does **not** run on interaction nets (BendRT is a flat C evaluator), has **no** SupGen, no `sup` syntax, no homoiconic self-change. "Fast" is a measurement, never a theorem.
- **Superposition and sharing only work on HVM5, and HVM5 does not exist yet as a usable runtime.** SUP values and lazy (incremental DUP) sharing are interaction-calculus runtime features; BendRT has neither, so a second use of a value is a real copy. The HVM5 kernel in `bend-runtime/hvm5` is a research prototype (`hvm5-ic`). Nothing in this repo may depend on them (ADR-031).

VERSIONS

- Local toolchain: **bend 2.0.25** (`bend version`); every snippet in these skills was checked with it.
- Project pin: **bend 2.0.39** (`packages/core/BEND_VERSION` , CI installs it with a SHA-256 check). 2.0.25 cannot emit a library without `main` ; JS library mode needs ≥ 2.0.39 (`../bend2-effects-targets/references/03-js-target-facts.md`).
- Version differences that change what you type: `../bend2-language/references/04-rules-and-errors.md` §"Version differences" (e.g. `bend PROOF.bend --verdict` does not exist in 2.0.25).

COMMANDS

Command	Does
`bend guide`	the official guide for the installed version; read it before writing Bend
`bend file.bend --check-only`	check the file and its imports, run nothing. A **program with an effect** (any `import "./x.js"`) exits 1 on the foreign-code verdict — the gate, not a type error (rule 8)
`bend file.bend [args]`	check, then run `main`
`bend file.bend -o out.js`	build JS. **Slow**: 0.6–8 s per test file, growing with its import graph; the run itself is ~0.1 s: rare, and in the background

Command	Does
`bend file.bend -o out.c` / `-o out`	build C / native. **The installed 2.0.39 binary on this devcontainer cannot** (`TypeError: … 'FL.book.tlds[k].c'`, even for hello world). Re-check first (`bend hello.bend -o /tmp/hello && /tmp/hello`); if it builds, use the plain command and delete this note. While it fails, use the source CLI of the same tag: `bun /tmp/bend-src/bend2/main.ts file.bend -o out`. Native and JS are different runtimes — measure both, see `bend2-runtime`
`bend PROOF.bend`	the hand-kept proof list; prefer `proof.mjs` (every `LAWS\*.bend`), and `proof:changed` in the loop
`bend base [--types|<name>]`	print Base (the standard library)

Always set `BEND_NO_TELEMETRY=1` in scripts and CI.

PROJECT CHECKLIST

1. Each module imports only modules below it; Bend has no mutual recursion across definitions, so use one tagged type instead of mutually recursive ones.
2. Prove as much as possible, with **small** laws: each layer's invariants and their same-named proofs go in its `LAWS*.bend`, stated `for x` where possible, each checking in well under a second. Enumerations are a last resort in `LAWS_TABLE_*` files. While iterating, prove only what the change reaches (`proof:changed`); run the full `proof` in the background before the commit (`bend2-proofs`).
3. Integers for anything exact (money in cents, time as day/minute ints, blob data as U32 words). F32 only behind a `Real` facade.
4. Tail recursion (or chunked loops) on anything that walks a large list: the JS target has a call-stack limit.
5. Never design around superposition, SupGen or lazy sharing: search is explicit enumeration, A/B is running both, a copy costs its size.
6. Bend never does I/O in this app. JS fetches, lexes and draws; Bend decides (`bend2-effects-targets`).
7. **Reuse before you write**: check `bend base` (and the layer below) for an existing helper, then **the Bend project directory** at builtwithbend.com `(https://builtwithbend.com/)` (147 builds, filterable by Libraries / Developer tools; the source of every entry is on GitHub) and the Bend hub `(https://hub.bend-lang.com)` (`import pkg@version/main.bend`), then the repo. Wrap Base, do not re-derive it; record `reused:` or `own:` in the module header. **For the full name list, use lilalittle/bend-packages** `(https://github.com/lilalittle/bend-packages)` — a curated A–Z of the hub refreshed weekly (`ALL_PACKAGES.md`: 1,036 packages, 201 named, grouped by proofs / math / tooling). It is the only complete list of names: `packages.json?limit=100` caps at 100 and 948 of the hub's entries have no name at all, so a term that "does not exist" in the API may simply be unnamed there and named here. **Before you pin a hub package, check Bend Crater** `(https://pedrovioliv.github.io/bend-crater/)` — every hub package file type-checked against every `bend` release (2.0.8 → 2.0.39, hourly, no package code runs). It is the only source for "does this pin compile on my compiler", and its data is queryable without the page: `…/bend-crater/data/{packages.json,releases.json,results.jsonl}` (`{pkg,file,bend,status,error,ms}`). Two caveats: it is **file-level**, so a `pass` on a submodule says nothing about the entry file; and `unsafe` is the `--check-only` verdict on a foreign effect, **not** a failure — every FFI package (`http`, `time`, `zlib`, `wire`, `dns`, `websocket`) reads `unsafe` on every release, and its gate is still `bend file.bend -o /tmp/x.js`. **Then read the dependency edges, not just the description.** `import 0x<hash>/file.bend` is Bend's only dependency mechanism, so a package's imports are its composition (what it assumes you already have) and its importers are worked examples of it in use — usually a better "how do I use this" answer than the one-line summary. Both come from the crawl: `bun tools/hub-index/query.mjs <term> --deps` prints `deps →` (upstream, resolved to names where the hub exposes them) and `used by N` (downstream, grouped across versions). Two rules when reading it: dependents pin a **hash**, so a fresh release legitimately shows zero importers while the library is everywhere; and the hub's own ranking at hub.bend-lang.com/packages `(https://hub.bend-lang.com/packages)` sorts by dependents/hot — that is popularity, not correctness. Prefer a package whose importers you recognise, and one whose upstream list is short.
8. **`Bool.pick` is a function, not a branch**: both arms are evaluated before it returns one. A recursion in each arm is 2^n — a 250-file filter that hangs for half an hour instead of slowing down. Branch with `match` on a `Bool` parameter or use `List.filter(~A, ~(x => …), xs)`, which matches internally. (Measured: `tools/bend-blobs/rank.bend`, 2^250 → 0.8 s.) The rest of the method — timing, input bisection, CPU profiles on the emitted JS, the cost table — is `bend2-performance`.
9. **Check before you compile**: `bend <module> --check-only` (~0.2 s) is the inner loop for a **pure** module; `-o core.mjs` is not. For a module that imports a foreign effect, `--check-only` always exits 1 with `defs rely on unsafe or foreign code` — that verdict rejects the effect itself, and it fires for the guide's own `Clock.now` example too, so it says nothing about your file. There the gate is `bend file.bend -o /tmp/x.js`. Keep cheap laws in `LAWS.bend` and whole-range enumerations in a separate exhaustive gate, because their cost is the range they normalize, not the file size.

10. **There is no incremental build in Bend 2.0.39.** Every `bend` call re-checks the file and its whole import closure, and caches nothing (a second identical run takes the same time). Speed comes from **small import closures** (`num/hash`, 1 file: 0.17 s; `bt/vm`, 9 files: 3.3 s), from selecting what a change reaches (`*:changed`), and from our content cache for compiled JS-lane modules.
11. **Check directly, run compiled.** `--check-only` is the fast check. Running `bend tests/x.bend` evaluates `main` *inside the checker*: an interpreter whose cost is the test's computation, never cached (`store/tests/codec` 13 s, `chain/tests/real_gme` 32 s). Compiled (`bend -o` + node) the same tests take 0.45 s and 0.3 s, and the runner caches compiled modules by content. The whole suite takes 223 s direct, 46 s compiled cold, and 5 s warm. `test:bend:changed` therefore compiles every pure-`main` test by default; the direct lane is for `IO` mains and the background parity run (`test:bend:direct`).

ROUTING

Task	Skill
syntax, ADTs, records, matching, recursion, termination, diagnostics	`bend2-language`
"consumed more than once", `+` / `-` markers, arrays, maps, closures, folds	`bend2-ownership`
`law`, `{==}`, induction, holes, dependent types, `@unsafe`, the gate	`bend2-proofs`
interaction nets, BendRT, parallel calls, threads, GPU, measuring cost	`bend2-runtime`
slow or hanging code: timing, input bisection, CPU profiles, the cost model, the traps	`bend2-performance`
`do` blocks, files, args, C/JS/native targets, JS library + Web Worker FFI	`bend2-effects-targets`
`dup` / `sup`, labels, collapse, SupGen, forking and testing branches	`bend2-superposition`
structuring a large app, user programs as AST + VM, capstone, reading repos	`bend2-app`
the HVM5 Interaction-Calculus runtime (kernel in Bend, code as data)	`hvm5-ic`
research → plan → spec → laws → tests → iterate, with many agents	`research-to-release`

VERIFY

```
export BEND_NO_TELEMETRY=1
bend version
bend path/to/module.bend --check-only                    # the module you edit
bun run --cwd packages/core check:changed                # modules that import it
bun run --cwd packages/core proof:changed                # law files it reaches (no tables)
bun run --cwd packages/core test:bend:changed            # behaviour: tests it reaches, compiled + cached
bun run --cwd packages/core test:bend:direct > /tmp/direct.log 2>&1 &   # lane parity, background
# full gate before a commit (all in the background; CI runs the first four of these and the build):
#   bun run fmt:check && bun run lint && bun run check && bun run test
#   bun run --cwd packages/core proof > /tmp/proof.log 2>&1 &
# `--check-only` = well-formedness (types, ownership, termination, laws); it never runs main, so it cannot
# tell you a test passes. Use test:bend:changed for behaviour, proof* for laws.
```

REFERENCES

File	Covers	Open when
`references/01-tutorial-index.md`	tutorial map, reading path, honest limits, how examples are checked	starting out; deciding what to read
`references/05-glossary.md`	Haskell/OCaml → Bend terms	a term is unfamiliar
`references/02-what-is-bend2.md`	what Bend 2 is, in one page	explaining Bend to someone
`references/04-release-status.md`	what is released vs announced	someone claims a feature exists

File	Covers	Open when
`references/03-hello.md`	the smallest program	first run on a new machine

C.2 · bend-gme/version-1/.opencode/skills/bend2/references/01-tutorial-index.md

--- title: "Bend 2 tutorial: table of contents" description: "A beginner's path through Bend 2 for Haskell and OCaml programmers: syntax and checker rules, ownership, interaction nets, parallel runs, laws and proofs, and what Bend 2 does not have." updated: 2026-10-28 verified: 2026-10-20 status: runnable canonical: /tutorial/ sources:

- https://bend2.dev/contents/ `(https://bend2.dev/contents/)`
- https://bend2.dev/llms.txt `(https://bend2.dev/llms.txt)`
- https://bend2.dev/learn/examples.json `(https://bend2.dev/learn/examples.json)`
- https://bend2.dev/errors/diagnostics.json `(https://bend2.dev/errors/diagnostics.json)`
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md `(https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)`
- https://github.com/777genius/awesome-bend `(https://github.com/777genius/awesome-bend)`
- https://hub.bend-lang.com `(https://hub.bend-lang.com)` ---

Bend 2 tutorial

This tutorial teaches Bend 2 to someone who already knows Haskell or OCaml. It covers the syntax and the checker's rules, how values are owned and copied, the interaction-net model behind the language's history, parallel execution, and how to state and prove properties as *laws*.

It is unofficial and not affiliated with Higher Order Company. The reference is bend2.dev `(https://bend2.dev/)` and the official guide, which `bend guide` prints from your installed compiler.

WHAT YOU CAN DO AT THE END

- Read a Bend 2 program, including its annotations, and predict whether the checker accepts it.
- State a law, prove it with a same-named `def`, and read the checker's messages when a proof fails.
- Say when a parallel call really runs in parallel, and how to measure it.
- Say what Bend 2 does **not** have (SupGen, `sup` syntax, copy-on-write forks of running code, homoiconic self-modification), and what to use instead.
- Open a real Bend repository, find its laws, entry point, tests, and effect boundary, and say what it proves and what it only measures.

TABLE OF CONTENTS

#	Chapter	You learn	Status
01	First programs	installing, the four run modes, functions, `let`, `match` on `Bool`, numbers, strings, tuples, lists	runnable
02	Data and recursion	algebraic data types, records, nested and multi-scrutinee matching, termination, imports, definition order	runnable
03	Ownership and copying	affine binders, `+`, `-`, `Data` versus `Type`, kinds, arrays, maps, templates, closures, folds	runnable
04	Interaction nets	cells, ports, active pairs, rules, confluence, the symmetric interaction combinators, labels and the text of the Interaction Calculus, β as one step	conceptual, pen and paper
05	Parallelism and the runtime	parallel calls, the fork shape, leaves, `!` GPU calls, `--threads`, measuring, auditing the emitted C	runnable
06	Duplication, superposition, SupGen	`dup` and `sup`, labels, search by superposition, and what Bend 2 does not have	conceptual, with checked parse errors

#	Chapter	You learn	Status
07	Laws and proofs	laws, `{==}`, match refinement, induction, rewrites, holes, the LAWS and PROOF layout, the gate	runnable
08	Dependent types and totality	types computed from values, `Unit` and `Empty`, length-indexed lists, termination, `@unsafe`	runnable
09	IO, files, and targets	`do` blocks, effects, fork and join, channels, files, arguments, C, JavaScript, and native builds	runnable
10	Branching and self-change	what SupGen, copy-on-write forks, and homoiconic code would need, and what Bend 2 offers instead	runnable, design
11	Rules and errors	the checker's rules, every official diagnostic with its real message, version differences	reference
12	Syntax reference	the model in three layers (kinds, types, values), every syntax form on one sheet, operator namespaces versus type ascriptions, laws and proofs, evaluation, and the checker's messages for the common mistakes	runnable
13	Glossary for Haskell and OCaml programmers	term-by-term mapping and definitions	reference
14	Superposition, collapse, and SupGen	the Interaction Calculus behind `sup` and `dup`, labels, collapse, search with erasure, what is public about SupGen, and whether a cheap fork exists	runnable, needs the Interaction Calculus binary
15	Capstone: a checked command parser	a job-queue parser built from constructors, `Maybe`, two proved laws, and command-line arguments	runnable
16	Reading real Bend repositories	a curated map of community and official repositories: what each one demonstrates, which files to open first, and which composition patterns to copy	reference

READING PATH

Read 01, 02, and 03 in order. They teach the language itself. Then read 07 and 08 (laws, proofs, dependent types), 05 (parallelism), and 09 (effects). Chapters 04 and 06 are the machine model and history: read them before 10, which applies everything to forking and testing branches. Keep 11 to 13 open as references, with 12 for every syntax form. Finish with 15, which combines the parts into one checked program. Then use 16 as a map of real repositories to read next.

HOW TO READ THE BOXES

Some sections set their key points apart in three kinds of box:

> **Core insight.** The one idea that the next part depends on.

> **Definition.** What a term means in this tutorial.

Composed of: a `text` block lists what a thing is made of. A `-` starts each part, a `|` between names means one of them, and indentation shows nesting. So `CON | DUP | ERA` reads as one of three cells.

VERSIONS

- Checked with **bend 2.0.25** on 2026-10-17. Run `bend version` to confirm yours.
- The examples on bend2.dev are pinned to **2.0.36** (commit `ae1101c`). 39 of its 42 examples print the documented output on 2.0.25. The other three print `All terms check.` where the site prints `ALL PROOFS CHECK`. 8 of 9 official diagnostics produce an error on 2.0.25, with the wording differences listed in chapter 11. The ninth uses `f?` names that 2.0.25 cannot parse.

- Version differences that change what you type are listed in chapter 11. The most important one: `bend PROOF.bend --verdict` does not exist in 2.0.25.

HONEST LIMITS

1. **Bend 2 does not run on interaction nets.** Chapter 04 explains the model behind the language's history. The released compiler emits a flat C evaluator called BendRT, which has no active pairs at runtime.
2. **The released compiler has no SupGen, no `sup` syntax, and no homoiconic self-modification.** Chapter 06 shows the evidence. `&{1, 2}` is a parse error, and nothing in `bend base` or `bend guide` describes SupGen.
3. **"Cheap" and "fast" are measurements, not theorems.** A law proves a property of the values a program computes. Cost, memory, and speed come from running the program and counting what the emitted C does.

HOW THE EXAMPLES ARE CHECKED

Each ```` ```bend ```` block is a complete program. `scripts/check-tutorial.py` compiles and runs every one of them with the installed `bend`:

- a block followed by an ```` ```output ```` block must print exactly that text;
- a block followed by an ```` ```error ```` block must fail with those lines;
- a block followed by a ```` ```contains ```` block must print those lines;
- a block with no expectation must pass `bend FILE --check-only`.

`ic` and `ic-collapse` blocks run Victor Taelin's Interaction Calculus reference binary when `IC_BIN` points to it, and are reported as skipped otherwise. Fences marked `text`, `sh`, or `bend-syntax` are documentation and are not run.

```
python3 bend-docs/scripts/check-tutorial.py bend-docs/src/content/tutorial
```

C.3 · bend-gme/version-1/.opencode/skills/bend2/references/02-what-is-bend2.md

--- title: What is Bend2? description: "Bend2 in one idea first, then the three pieces and the honest release status." updated: 2026-08-30 verified: 2026-09-05 status: prerelease canonical: /notes/what-is-bend2/ sources:

- https://bend2.dev/notes/what-is-bend2/ (https://bend2.dev/notes/what-is-bend2/)
- https://github.com/VictorTaelin/Bend2 (https://github.com/VictorTaelin/Bend2) ---

What is Bend2?

Bend2 is an unreleased programming language from Higher Order Company. Its one-sentence idea: **write ordinary functional code, get parallelism, proofs, and even synthesized programs for free** — because the runtime evaluates your program as a graph where independent pieces cannot interfere, so scheduling needs no annotations from you.

Three pieces combine here. First, functional code that parallelizes automatically: recursion that splits work runs on all cores without threads, locks, or pragmas. Second, a dependent type system in the tradition of Higher Order Company's Kind: types can state properties like "this list is sorted", and the checker rejects code that violates them. Third, program synthesis (called SupGen): types precise enough to pin down behavior give a searcher something to aim at and the checker something to accept or reject.

WHY THIS COMBINATION MATTERS

Each piece exists elsewhere on its own. Array languages parallelize; Lean and Kind have dependent types; research languages synthesize code. The bet is that together they multiply: precise types make generated code checkable, so a synthesizer can propose and the checker can dispose — and everything runs on a runtime that parallelizes without being asked. The HVM runtime schedules independent graph rewrites; it never needed annotations and still does not.

WHAT THE LANGUAGE LOOKS LIKE

Bend2 reads like Python with types. A function is `def` with typed parameters and a return type after the arrow. Effects run in `do` blocks over an explicit `IO` type. A call runs on the GPU when its name carries `!`. A theorem is an `assert` block whose proof is an ordinary `def` of the same name. Each construct gets its own page under Learn by Example.

STATUS: WHAT ACTUALLY EXISTS

As of 2026-08-30: the public `VictorTaelin/Bend2` repository is empty — no code, no branches, no binary, no package, no license. What is public: four posted code samples (walked through in Learn by Example) and the HVM4 runtime. Taelin's estimate at last check points around 2026-09-02; track the release status page, which is updated as the public record changes.

C.4 · bend-gme/version-1/.opencode/skills/bend2/references/03-hello.md

--- title: "Hello world" description: "Effects in Bend2: main returns an IO action sequenced in a do block." updated: 2026-08-13 verified: 2026-09-05 status: docs-only canonical: /learn/hello/ sources:

- https://bend2.dev/learn/hello/ `(https://bend2.dev/learn/hello/)` ---

Hello world

Every Bend2 program starts at `main`. This one reads a name and prints a greeting — and in doing so shows how Bend2 handles effects: not as statements that happen, but as values of type `IO` that describe what to do.

```
import Base

# Performs effects on the CPU.
def main() -> IO<Unit>:
  do IO<Unit>:
    name <- IO::input()
    IO::print("Hello, " ++ name)
```

LINE BY LINE

- `import Base` brings in the standard library, which supplies `IO`.
- `def main() -> IO<Unit>:` declares a function, Python-style, with the return type after the arrow. `IO<Unit>` is `IO` applied to `Unit`: a computation run only for its effects. (`main` need not be effectful — a pure program declares `def main() -> U32` and returns a number.)
- `do IO<Unit>:` sequences effects in a block that names its type.
- `name <- IO::input()` runs an effect and binds its result; `<-` is "run and name the answer".
- `IO::print("Hello, " ++ name)` is the block's value. Strings concatenate with `++`, library functions are reached with `::` paths, `#` starts a comment.

WHERE EFFECTS RUN

On the CPU. GPU evaluation — marking a call with `!` — is for pure functions; input and output stay on the host. That split is the whole story of the GPU page.

C.5 · bend-gme/version-1/.opencode/skills/bend2/references/04-release-status.md

--- title: Release status description: "Is Bend2 out? No — and here is the dated trail of estimates." updated: 2026-08-27 verified: 2026-09-05 status: prerelease canonical: /notes/release-status/ sources:

- https://bend2.dev/ `(https://bend2.dev/)`
- https://bend2.dev/notes/bend2-release-date/ `(https://bend2.dev/notes/bend2-release-date/)` ---

Release status

No. As of 2026-08-27, Bend2 has not been publicly released. The current estimate is approximately 2026-09-02.

THE DATED TRAIL

- **2026-03-05**: launch-blocker list — HVM4 AOT compiler, HVM4 GPU runtime, Bend2–SupGen integration.
- **2026-07-18**: "Bend2 complete"; CUDA backend ported overnight from the Metal reference.
- **2026-08-12**: delay post (≈08-28 implied).
- **2026-08-27**: launch plan — copiable-kind fix, weekend testing, Monday presentation recording, possible Wednesday launch (≈09-02).

WHAT "RELEASED" WILL MEAN HERE

A public compiler (or repository with code), a license, and runnable examples. Until all three exist, every Bend2 sample on this site stays labeled documentation-only. This site is unofficial and not affiliated with Higher Order Company.

C.6 · bend-gme/version-1/.opencode/skills/bend2/references/05-glossary.md

--- title: "Bend 2 tutorial 13: glossary for Haskell and OCaml programmers" description: "Bend 2 terms with their Haskell and OCaml counterparts, and the differences that matter: affine binders, explicit kinds, checked laws, interaction nets, and the runtime." updated: 2026-10-28 verified: 2026-10-18 status: docs-only canonical: /tutorial/13-glossary/ sources:

- https://bend2.dev/learn/ (https://bend2.dev/learn/)
- https://bend2.dev/notes/bend2-vs-lean/ (https://bend2.dev/notes/bend2-vs-lean/)
- https://bend2.dev/notes/interaction-nets-explained/ (https://bend2.dev/notes/interaction-nets-explained/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

13: glossary for Haskell and OCaml programmers

Each row says what a Haskell or OCaml programmer would reach for first, and what Bend 2 does with the same idea. The last column flags a difference that will change how you write code.

CONCEPTS YOU ALREADY KNOW

You know (Haskell / OCaml)	Bend 2	Difference that matters
`data T = A | B` / `type t = A | B`	`type T` is `Data:` with constructors `A{}` and `B{}`	constructors always use braces; a field is `name: Type`
`case x of` / `match x with`	`match x:` with `case K{...}:`	the scrutinee must be a parameter or pattern variable
`if c then a else b`	`match c:` on `True{}` and `False{}`	there is no `if`
`Maybe a` / `option`	`Maybe<a, A>` in Base	the first argument is the quantity or kind
`Either a b` / `(a, b) result`	`Result<a, b, E, A>` in Base	the quantities are explicit; `IO.try` unwraps
tuples `(a, b)` / `a * b`	`A & B` and `(a, b)`	pairs are the usual way to return two values
lists `x : xs` (Haskell) / `x :: xs` (OCaml)	`Con{head, tail}`, `h <> t`, `[a, b]`	a list is affine unless its quantity is `&2`
`do` notation	`do IO<T>:` with `x : T <- act`	every bind names its type
`return x` / `pure x` (Haskell)	`return x` in a `do` block (`IO.pure`)	the value is boxed as an effect; `<-` runs it; chapter 09
`main :: IO ()`	`def main() -> IO(Unit):`	the same, with `Unit{}` as the value
`where`, local `let`	`x = v` in a body	a plain `let` is used once; `+x = v` is reusable
type classes	not present; Base uses `Type.verb` names	every operation is a named function, such as `U32.add`
`newtype` / abstract types	a `type` with one constructor	constructors are always visible
QuickCheck property	`law`	a law is a checked proof, not a sampled test
GADT with a length index	a type computed by a `def`, such as `Vec(A, n)`	the index is an ordinary function

You know (Haskell / OCaml)	Bend 2	Difference that matters
`INLINE` or specialization	a template `~f`	each distinct argument gets its own compiled copy
linear types	affine binders, `+` for `Data`	`+` pays a reference count; there is no `Linear` keyword
`Data.Map`	`Map<&2, V>`	string keys; `get` takes its default first
mutable arrays in `ST`	`Array<T>` with `a[i] <- v`	one owner; a write returns the array
`Just x` (Haskell), `Some x` (OCaml)	`Some{x}`	a constructor takes braces with no space before them; chapter 02
`f :: a -> b` (Haskell), `val f : a -> b` (OCaml)	`def f(x: A) -> B:`	`:` gives a type, `->` gives the result, and the final `:` opens the body; chapter 12
`(e :: t)` (Haskell), `(e : t)` (OCaml)	`{e : T}`	a real ascription; `(e : T)` is different: it chooses the operator namespace for the operators in `e`; chapter 12
`==` as a comparison	`T.is_eq(a, b)`	`==` appears only in an equality type such as `{a == b : T}`; chapter 12
`id :: a -> a` (Haskell), `let id x = x` (OCaml)	`def id(-A: Type, x: A) -> A:`	the type is an argument: `id(U32, 42)`. Bend does not infer it; chapter 03
`{x : T}` as a constructor	`{t : T}` as a type ascription	a constructor has a name before its braces (`Tok{3n}`); braces with no name ascribe one term; chapter 12

TERMS SPECIFIC TO BEND 2

Affine. A binder used at most once. Dropping it is free. Chapter 03.

Strict. Compiled Bend computes a call's arguments before the body runs. To keep work unevaluated, use a lambda, a template, or data that an interpreter runs. Chapter 12, section 11, shows all three.

Quantity. The number of uses a binder may have: `&1` means at most once, and `&2` means reusable. The binder markers choose one: `+x` is reusable, `x` is used at most once, and `-x` is erased, existing only for the checker. Chapter 03.

Kind. A classifier of types that carries a usage budget, written `Kind(q)`, where `q` is a quantity. `Type` is `Kind(&1)`: its values are used at most once. `Data` is `Kind(&2)`: its values may be copied with `+`. A kind is not a quantity. The quantity is the count, and the kind is the classifier that carries it. Haskell's kinds are different; see the table below. Chapter 03.

Constructor (ctor). A named way to build a value of a `type`. `Segment{3n}` applies the constructor `Segment` to `3n`. Chapter 02.

Term and value. A term is an expression as written, such as `(6 * 7 : U32)`. Its value is what it computes to, `42`. Chapter 12, section 11.

Annotation. `(e : T)` chooses the operator namespace for the operators in `e`, and `{t : T}` checks that the single term `t` has type `T`. `(e : T)` is not a general ascription: `(a + b : U32)` is `U32.add(a, b)`, `(a <= b : U32)` returns `Bool`, and with no operator `(x : U32)` is just `x`. Chapter 12, section 3.

Erased (-). A binder that exists only for the checker. The compiler deletes it, so it is not a run-time argument. Chapter 03.

Runtime use. The compiled body needs the binder's value: it returns the value or computes with it. An erased binder cannot be used at run time. Chapter 03.

Explicit type argument. Bend infers almost nothing, so a type is passed at the call, as in `identity(U32, 42)`. Chapter 03.

Closure. A function value, such as `x => (x + 1 : U32)`. It is affine, so it may be called once. Chapter 03.

Array literal. `[v : T*n]` or `[v : T^d]`: a fill value `v`, an element type `T`, and either `n` slots (a power of two) or `2^d` slots. `Array.new(T, d, v)` builds the same array. Chapter 03.

Law. A claim checked by the compiler. A law has a same-named proof def. Chapter 07.

Proof. A definition whose type is a law's claim. Chapter 07.

Curry-Howard. A law is a type, and its proof is a program of that type. Chapter 07 shows what the gate checks beyond the type.

`{==}`. Proves an equality when both sides compute to the same term. Chapter 07.

Rewrite `%e : P`. Replaces the goal using the equation `e`, with `_` marking where the right-hand side goes. Chapter 07.

Hole. `?TODO` leaves a goal open, and `?name` prints the goal. Chapter 07.

`exs`. A law that asks for a witness; the proof returns it with the equation. Chapter 07.

Unit and Empty. `Unit` has one value. `Empty` has none, so a function that takes an `Empty` need only match nothing. Chapter 08.

Dependent type. A type computed by a `def` from a value, such as `IsZero(n)` or `Vec(A, n)`. A normal type such as `U32` has no parameters, and a function type such as `U32 -> U32` maps values to values. Chapter 08.

Pure (`return`). `return v` in a `do` block boxes a pure value as an effect. It is `IO.pure`. Chapter 09.

Bind (`<-`). Runs an effect and names its result with a type. It is `IO.bind`. Chapter 09.

IO value. An effect of type `IO(A)`. It has no constructors, so only `<-` opens it. Chapter 09.

Totality. Every safe recursive call shrinks its first changing argument. Chapter 02.

Fuel. A `Nat` argument that bounds a loop the outside world stops. Chapter 02.

`@unsafe`. A definition exempt from the termination check and outside the proof guarantees. Chapter 08.

Parallel call. `a b = f(x) g(y)`, two calls the compiler may run at once. Chapter 05.

Bang. `f!(x)`, the call on the GPU, with its parallel calls. Chapter 05.

Balanced fork. A parallel call whose halves do about the same work. Chapter 05.

Leaf. A tail loop over 8 to 64 items that runs on one core. Chapter 05.

Thread count. The `--threads N` setting of a native binary. It defaults to the CPU count, and the fastest value depends on the program, the input, and the machine, so measure it. Chapter 05.

`term_keep`, `ctr_take`, `rfc_seal`. Runtime names for reference counts in the emitted C. Chapter 05.

BendRT. The runtime that compiles Bend to flat C with affine ownership and balanced forks. Chapter 04.

Interaction net. A graph of cells rewritten by local rules. Chapter 04.

Cell and port. A cell has a symbol, one principal port, and auxiliary ports. Chapter 04.

Active pair. Two cells joined at their principal ports. Only these reduce. Chapter 04.

Confluence. Distinct active pairs commute, so the order does not change the result. Chapter 04.

CON, DUP, ERA. The three symbols of Lafont's symmetric interaction combinators. Chapter 04.

Duplication (dup). `! &L{x,y} = t; body` splits `t` into two uses, with label `L`. Chapter 14.

Superposition (sup). `&L{a,b}` holds two alternatives with label `L`. In a net it is a DUP cell with label `L`, read from the other side. The released Bend 2 parser rejects it. Chapter 14.

Label. The number on a `&L{}` or `! &L{}`. Equal labels pair superpositions, and different labels commute. Chapter 14.

Interaction Calculus (IC). Victor Taelin's text notation for labelled nets: λ, application, duplication, superposition, and erasure. Chapter 14.

Collapse. Rewrites a result so that no superposition or duplication remains inside its alternatives (`ic run FILE -C`). Chapter 14.

SupGen. Victor Taelin's program synthesizer, built on superposition search. Its source is not published, and released Bend 2 does not include it. Chapter 14.

Homoiconic. Code that is also data the program can build and inspect. Chapter 10 shows a limited form.

Gate. `bend PROOF.bend`. It passes only when every law is proved, with no holes and no unsafe dependency. Chapter 07.

TERMS THAT LOOK ALIKE BUT DIFFER

Term	Looks like	Actually
`Type`	Haskell's `Type`, the kind of ordinary types (`Int :: Type`)	Bend's kind `Kind(&1)`: values of these types are used at most once. Haskell values are shared and garbage-collected, so Haskell types are not affine. In Bend, `Type` is also the name of the one universe (`Type : Type`, in the guide).
`Data`	the Haskell `Data` class (`Data.Data`, `deriving Data`)	Bend's kind `Kind(&2)`: values may be copied with `+`. It has nothing to do with generic programming
`data`	Haskell's `data T = A | B` keyword	Bend declares a datatype with `type T is Data:`. `is Data` sets its reuse kind, and `is Type` makes it affine
`Kind`	Haskell's kind of a type (`Maybe :: Type -> Type`)	a classifier of types that carries a usage budget, `Kind(q)`; it is not a quantity
`quantity`	Haskell's `%1 ->` multiplicity (LinearTypes, off by default)	the number of uses a binder may have: `&1` at most once, `&2` reusable. The binder markers choose one
`+`	the addition operator	a quantity marker; `(a + b : T)` is the operator
`-A`	negation	an erased parameter
`~f`	a function value	a template argument: copied into each call at compile time. It is not a closure and does not defer work
`IO`	a monad you can run anywhere	an effect whose binds carry types; run only by the event loop
`Array`	a Haskell array	one-owner storage updated in place

NEXT

Return to the table of contents in the index.

C.7 · bend-gme/version-1/.opencode/skills/bend2-language/SKILL.md

Bend 2: the language

Three layers, checked top-down: kinds say how a value may be used, types say what it is, values compute.

MODEL

- **Declarations**: `import`, `type T is Data:` (constructors), `def f(x: A) -> B:`, `law name:`.
- **Values** are built by constructors (`Segment{3n}`) and taken apart by `match` (exhaustive).
- **Numbers carry their type in the literal**: `3n` is `Nat`. The numeric types are `Nat`, `U32` and `F32`; there is **no F64, I32 or U64**. `U32` wraps mod 2^{32} (`../bend2-effects-targets/references/03-js-target-facts.md`).
- **Strict**: arguments are evaluated before the call.
- **Termination is checked**: recursion must shrink a structural argument unless marked `@unsafe` (`bend2-proofs`).
- Full sheet: `references/03-syntax-reference.md` (§1 model … §12 rejected forms).

RULES

1. Every `match` is exhaustive; add the missing constructor rather than a catch-all you cannot justify.
2. Define before use where the checker needs it (definition order and imports: `references/02-data-and-recursion.md`).
3. No mutual recursion between definitions. Merge the types into one tagged type and dispatch with `match`.
4. Use `Nat` (`0n`, `1n+p`) for structural recursion and proofs; use U32 for data and arithmetic that must be fast; never use F32 where an integer is exact.
5. Prefer records with named fields for anything with more than two fields.

6. Read the error's `expected` / `observed` / `Location` lines before changing code (`references/04-rules-and-errors.md` §"Reading an error").

PATTERNS (CHECKED WITH BEND 2.0.25)

```
import Base

type Shape is Data:
  Point{}
  Segment{length: Nat}

def measure(shape: Shape) -> Nat:
  match shape:
    case Point{}:
      0n
    case Segment{length}:
      length

def main() -> Nat:
  measure(Segment{3n})
```

Output: `3n`. Structural recursion on `Nat` uses `case 0n:` / `case 1n+p:` (see the law example in `bend2-proofs`).

PITFALLS

Message / symptom	Cause	Fix
`observed : n (consumed more than once)`	an affine binder used twice	mark it copyable `+n` or restructure (`bend2-ownership`)
non-exhaustive match	a constructor has no `case`	add the case
termination error on a recursive call	the argument does not shrink	recurse on a sub-structure, or justify `@unsafe`
a site example prints `ALL PROOFS CHECK`, 2.0.25 prints `All terms check.`	version difference	expected; see `references/04-rules-and-errors.md` §"Version differences"
`&{1, 2}` parse error	`sup` syntax is not in released Bend 2	see `bend2-superposition`

Every official diagnostic with its exact text: `references/04-rules-and-errors.md` §"The official diagnostics" and §"Errors this tutorial found beyond the official list".

VERIFY

```
BEND_NO_TELEMETRY=1 bend file.bend --check-only   # pure modules: types, ownership, termination (~0.2 s)
BEND_NO_TELEMETRY=1 bend file.bend                # run main inside the checker: an interpreter, fine for tiny
                                                  # mains, slow for real tests (13-32 s measured), F32 unreduced
BEND_NO_TELEMETRY=1 bend file.bend -o /tmp/x.mjs  # compile (0.3-7 s by import graph), then run with node: how
                                                  # tests run (test-bend.mjs caches it by content)
```

REFERENCES

File	Covers	Open when
`references/01-first-programs.md`	install, run modes, functions, `let`, `match` on Bool, numbers, strings, tuples, lists	writing a first module
`references/02-data-and-recursion.md`	ADTs, records, nested and multi-scrutinee match, termination, imports, definition order	modeling domain data
`references/04-rules-and-errors.md`	the checker's rules, every diagnostic, version differences	any error message

File	Covers	Open when
`references/03-syntax-reference.md`	every syntax form, operators vs ascriptions, evaluation, rejected forms	unsure how to write something
`references/05-first-programs-concepts.md`, `references/06-data-matching-concepts.md`	short concept pages	a quick refresher

RELATED

`bend2` (router) · `bend2-ownership` (copy errors) · `bend2-proofs` (laws, termination) · `bend2-app` (large codebases).

C.8 · `bend-gme/version-1/.opencode/skills/bend2-language/references/01-first-programs.md`

--- title: "Bend 2 tutorial 01: first programs" description: "Install Bend 2, learn the four ways to run a file, and write functions, let bindings, Bool matching, numbers, strings, tuples, and lists that the checker accepts." updated: 2026-10-28 verified: 2026-10-18 status: runnable canonical: /tutorial/01-first-programs/ sources:

- https://bend2.dev/learn/hello/ (https://bend2.dev/learn/hello/)
- https://bend2.dev/learn/functions/ (https://bend2.dev/learn/functions/)
- https://bend2.dev/learn/numbers/ (https://bend2.dev/learn/numbers/)
- https://bend2.dev/notes/bend2-syntax-primer/ (https://bend2.dev/notes/bend2-syntax-primer/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

01: first programs

Every program in this tutorial was checked with `bend 2.0.25`. Blocks marked `output` show what the compiler printed. Blocks marked `error` show the checker rejecting a program, and the lines are the real messages.

INSTALL AND CHECK THE TOOLCHAIN

```
curl -fsSL https://bend-lang.com/install.sh | sh
bend version
bend guide
```

`bend version` prints the compiler version. `bend guide` prints the language reference that ships with the compiler, and `bend base` prints the standard library. Read the guide before you write Bend code: it is short and it is the authority.

`bend update` installs the latest compiler. The bend2.dev examples are checked against 2.0.36, which has the `--verdict` option. Version 2.0.25 does not have it, and chapter 07 explains what that option needs.

THE FOUR WAYS TO RUN A FILE

Command	What it does
`bend file.bend`	Checks the file, then runs `main`. An `IO` `main` runs compiled; a pure `main` is evaluated by the checker, which is slow for big work.
`bend file.bend --check-only`	Checks the file and its imports, and runs nothing. Use it after every definition.
`bend file.bend -o file`	Builds a native binary with clang 14 or later. Then run `./file --threads 8`.
`bend file.bend -o file.c`	Emits one C file. `-o file.js` emits JavaScript instead.

Two options matter for the binaries: `--threads N` sets the number of CPU threads, and `--gpu off` runs `!` calls on the CPU.

HELLO, WORLD

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

```
Hello, world!
```

- `import Base` loads the prelude. Every program in this tutorial starts with it.
- `def main() -> IO(Unit):` declares the entry point. `IO(Unit)` is an effect that returns `Unit`, the type with one value `Unit{}`. In Haskell, this is `main :: IO ()`.
- `do IO<Unit>:` opens a block whose lines run in order, like Haskell's `do`.
- `IO.print` is an effect defined in Base.

FUNCTIONS

```
import Base

def square(+x: U32) -> U32:
  (x * x : U32)

def main() -> U32:
  square(7)
```

```
49
```

- Parameter types follow `:`, and the result type follows `->`.
- The last expression is the result. There is no `return`.
- Every operator needs an annotation, even on `Nat`: `(n + 1n : Nat)`. The type in the parentheses chooses the operator namespace, and one annotation can cover a whole operator expression, as in `(x * x * x : U32)`. A bare `x * x` is rejected, as the next example shows. The bundled guide (`bend guide`) still says a bare operator means `Nat`; the 2.0.25 checker rejects that. An annotation covers only the operators inside its own parentheses: in `(U32.add(a + b, 1) : U32)`, the `a + b` needs its own annotation. Chapter 12 lists every rule of this kind.
- `+x` makes the parameter reusable, because `x` appears twice. Chapter 03 explains why plain parameters can be used only once.

```
import Base

def add(x: U32, y: U32) -> U32:
  x + y

def main() -> U32:
  add(40, 2)
```

```
- message  : a type for this operator (write (a + b : Nat))
```

In Haskell, `square :: Word32 -> Word32` and `square x = x * x` would compile without any annotation. Bend asks for the annotation because it does almost no type inference, and that keeps its error messages precise.

The type is not only a result type. Bend's `+` is a call to a name in the type's namespace, so `(x * x : U32)` means `U32.mul(x, x)`. The `U32` in the parentheses chooses that name; the declared `-> U32` cannot. "Braces or parentheses" below shows the difference between the two colon forms.

READING `:`, `->`, AND `=`

Three marks do most of the work:

- `name: Type` says *has type*. It appears in parameters, constructor fields, and `for` binders.

- `(expr : Type)` chooses the operator namespace for the operators in `expr`, and `{term : Type}` checks that one term has type `Type`, as in `{12 : U32}`. They are not interchangeable: see “Braces or parentheses” below. Bend does almost no inference, so you write the type where the checker needs it.
- `A :` at the end of a line *opens a block*: a body, a `case`, a `do`, or a `law`.
- `->` separates the parameters from the result type. `=` binds a value. `A :` never means ”equals”.

A signature reads from left to right:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name
```

- `name: String` gives the parameter `name` its type.
- `-> IO(String)` gives the result: an effect that returns a `String`.
- The final `:` opens the body. The `do IO<String>:` header is a separate form, with angle brackets. Chapter 12 lists the rules for both.

LET BINDINGS

```
import Base

def main() -> U32:
  +price = {12 : U32}
  tax = {3 : U32}
  (price + price + tax : U32)
```

```
27
```

- `tax = {3 : U32}` binds a value used once.
- `+price = {12 : U32}` binds a value that the body uses twice.
- Inside a `do` block, a let needs its type, as in `n : U32 = (6 * 7 : U32)`. Chapter 09 covers `do`.

WHY A LET NEEDS A TYPE

A `let` has no type to take from, and Bend does no inference, so a bare literal is rejected:

```
import Base

def main() -> U32:
  +price = 12
  (price + price : U32)
```

```
- expected : an annotated term (cannot infer)
```

The fix is an annotation, `{12 : U32}`. That is not a constructor: a constructor has a name in front of its braces, as in `Tok{3n}` (chapter 02). Braces with no name hold one term and its type.

BRACES OR PARENTHESES

The two colon forms do different jobs. Braces ascribe a type to one term, such as `{12 : U32}` or `{x : U32}`: the checker requires that term to have that type. An operator expression needs parentheses, because the parenthesized form resolves the operators:

```
import Base

def main() -> U32:
  {6 * 7 : U32}
```

```
- message  : a type for this operator (write (a * b : Nat))
```

Write `(6 * 7 : U32)` instead. That is `U32.mul(6, 7)`: the `U32` before `)` names the operator namespace, not the result type. `(a <= b : U32)` is `U32.is_le(a, b)` and returns `Bool`; `(x : U32)` with no operator does nothing at all, so write `{x : U32}` to require `U32`. A def's `-> U32` cannot supply this namespace, which is why the annotation stays inside the body.

The parentheses do not delay the work either: Bend is strict, so the term runs when its definition runs (chapter 12, section 11).

CONDITIONALS: MATCH, NOT IF

Bend has no `if`. You branch with `match` on `True{}` and `False{}`, and the scrutinee must be a parameter, not a computed call:

```
import Base

def choose(test: Bool, yes: U32, no: U32) -> U32:
  match test:
    case True{}:
      yes
    case False{}:
      no

def main() -> U32:
  choose(U32.is_lt(3, 5), 10, 20)
```

```
10
```

The computed comparison `U32.is_lt(3, 5)` is passed to `choose`, and `choose` matches its own parameter. Matching the call directly is rejected:

```
import Base

def main() -> U32:
  match U32.is_eq(1, 1):
    case True{}:
      1
    case False{}:
      0
```

```
- message  : a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)
```

NUMBERS

Bend has three numeric types. `Nat` is an unbounded natural, written with an `n` suffix (`3n`). `U32` is a 32-bit unsigned integer, and its arithmetic wraps. `F32` is a 32-bit float. This chapter uses `Nat` and `U32`.

```
import Base

def main() -> String:
  n = Nat.add(20n, 22n)
  w = (6 * 7 : U32)
  Nat.show(n) ++ " / " ++ U32.show(w)
```

```
"42 / 42"
```

`Nat` patterns use `0n` and `1n+p`, where `p` is the predecessor (chapter 02). `U32` wraps at 2^32:

```
import Base

def main() -> U32:
  (4294967295 + 1 : U32)
```

```
0
```

`F32` values print as written in a checked program (`1.5`). `F32` arithmetic, however, is declared in Base as a law, so the checker shows `F32.mul(1.5, 2.0)` as an unevaluated term. A native build computes it, and `IO.print(F32.show(F32.mul(1.5, 2.0)))` prints `3`.

STRINGS, TUPLES, AND LISTS

Strings join with `++`. `String.length` returns a `Nat`:

```
import Base

def main() -> IO(Unit):
  text = String.to_upper("bend")
  IO.print(text ++ " " ++ Nat.show(String.length("proof")))
```

```
BEND 5
```

A pair has type `A & B` and value `(a, b)`:

```
import Base

def sum(pair: U32 & U32) -> U32:
  match pair:
    case (a, b):
      (a + b : U32)

def main() -> U32:
  sum((19, 23))
```

```
42
```

A list is `Nil{}` or `Con{head, tail}`, and `[5, 10, 15]` builds one:

```
import Base

def sum(xs: List<U32>) -> U32:
  match xs:
    case Nil{}:
      0
    case Con{head, tail}:
      (head + sum(tail) : U32)

def main() -> U32:
  sum([5, 10, 15])
```

```
30
```

The `: U32` around `head + sum(tail)` is the operator annotation from “Braces or parentheses” above: it selects `U32.add`. The declared `-> U32` cannot supply that namespace. The list in that example is consumed as the function walks it. Chapter 03 explains which lists can be reused and how `List<&2, U32>` changes that.

EXERCISES

1. Double and add. Write `double_plus` so that it returns `2a + b`. The parameter `a` is used twice, so it needs `+`. Then `main` should print `25`.

```
import Base

def double_plus(+a: U32, b: U32) -> U32:
  (a + a + b : U32)

def main() -> U32:
  double_plus(10, 5)
```

```
25
```

2. A label from a comparison. Write `label(n)` that returns `"small"` when `n < 10` and `"big"` otherwise. Remember that `match` cannot inspect `U32.is_lt(n, 10)` directly, so pass it to a helper.

```
import Base

def pick(test: Bool) -> String:
  match test:
    case True{}:
      "small"
    case False{}:
      "big"

def label(n: U32) -> String:
  pick(U32.is_lt(n, 10))

def main() -> String:
  label(42)
```

```
"big"
```

NEXT

Chapter 02 covers data types, records, recursion and its termination rule, nested patterns, imports, and definition order.

C.9 · bend-gme/version-1/.opencode/skills/bend2-language/references/02-data-and-recursion.md

--- title: "Bend 2 tutorial 02: data and recursion" description: "Algebraic data types, records, nested and multi-scrutinee matching, the termination rule, imports, and definition order, with the real checker messages." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/02-data-and-recursion/ sources:

- https://bend2.dev/learn/adts/ (https://bend2.dev/learn/adts/)
- https://bend2.dev/learn/records/ (https://bend2.dev/learn/records/)
- https://bend2.dev/learn/pattern-matching/ (https://bend2.dev/learn/pattern-matching/)
- https://bend2.dev/learn/recursion/ (https://bend2.dev/learn/recursion/)
- https://bend2.dev/learn/totality/ (https://bend2.dev/learn/totality/)
- https://bend2.dev/learn/imports/ (https://bend2.dev/learn/imports/)
- https://bend2.dev/learn/definition-order/ (https://bend2.dev/learn/definition-order/) ---

02: data and recursion

This chapter covers the data types you define, how a `match` reads them, and the rule that makes recursion safe to check. Haskell and OCaml readers will recognize almost everything. The differences are the explicit annotations and the fixed order of definitions.

ALGEBRAIC DATA TYPES

```
import Base

type Shape is Data:
  Point{}
  Segment{length: Nat}

def measure(shape: Shape) -> Nat:
  match shape:
    case Point{}:
      0n
    case Segment{length}:
      length

def main() -> Nat:
  measure(Segment{3n})
```

```
3n
```

- `type Shape is Data:` declares a type and says its values may be reused (chapter 03). Each indented line is a constructor.
- Constructors always use braces, even without fields: `Point{}`.
- Fields are named inside the braces: `Segment{length: Nat}` builds `Segment{3n}`.
- Haskell's `data Shape = Point | Segment { length :: Nat }` is the same declaration.

TYPES, CONSTRUCTORS, AND VALUES

A type is a set of values: `Shape`. A constructor (often shortened to *ctor*) is a named way to build one of those values: `Point{}` and `Segment{length: Nat}` are the two constructors declared in `type Shape`. In Haskell, `data Shape = Point | Segment Nat` declares the same two constructors. A value is what a constructor builds: `Point{}` and `Segment{3n}` are both values of type `Shape`.

A name followed by braces, `K{...}`, is always a constructor. The braces do different jobs depending on where they appear:

- In a `type` declaration, `Segment{length: Nat}` lists the fields of the constructor.
- In an expression, `Tok{3n}` applies the constructor to `3n`. The compiler reads it as `Ctor(Tok, [3n])`.
- In a `match` pattern, `case Segment{length}:` takes the value apart and binds the field.

Braces without a name are a different form. `{12 : U32}` is an annotation, `Ann(12, U32)`, and it is not a constructor. Chapter 12 lists every use of braces.

```
import Base

type Shape is Data:
  Point{}
  Segment{length: Nat}

def main() -> Shape & Shape:
  (Point{}, Segment{3n})
```

```
(Point{}, Segment{3n})
```

Both values are built directly, and no `match` is needed to make them. `Point{}` takes no fields, and `Segment{3n}` takes one.

Base declares its own types the same way. `bend base --types` prints them:

Type	Constructors	Example value
`Nat`	`Zero{}`, `Succ{pred}`	`2n`, which is `Succ{Succ{Zero{}}}`

Type	Constructors	Example value
`Bool`	`False{}` , `True{}`	`True{}`
`Unit`	`Unit{}`	`Unit{}`
`Maybe<a, A>`	`None{}` , `Some{value}`	`Some{42}`
`Result<a, b, E, A>`	`Fail{error}` , `Done{value}`	`Done{5}`
`List<a, A>`	`Nil{}` , `Con{head, tail}`	`[1, 2]` , which is `Con{1, Con{2, Nil{}}}`
`String`	`SNil{}` , `SCon{head, tail}`	`"hi"` , which is `SCon{'h', SCon{'i', SNil{}}}`

Angle brackets name types. `Maybe<&2, U32>` is a type, and `Some{42}` is a value of that type. In Haskell, `Maybe a` is a type constructor and `Just x` is a data constructor. Bend keeps the two apart the same way.

A constructor needs its type from context. When a `let` has no context, write the type:

```
import Base

type Tok is Data:
  Tok{n: Nat}

def show(t: Tok) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  t : Tok = Tok{3n}
  show(t)
```

```
3n
```

RECORDS

A record is a type with one constructor:

```
import Base

type Point is Data:
  Point{x: U32, y: U32}

def translate(p: Point) -> Point:
  match p:
    case Point{x, y}:
      Point{(x + 1 : U32), (y + 2 : U32)}

def main() -> Point:
  translate(Point{3, 4})
```

```
Point{4, 6}
```

`match` consumes `p` and binds its fields. The result is a new `Point` . Nothing is mutated.

RECURSION AND THE TERMINATION RULE

Bend repeats work by recursion, and every recursive call must be checked to terminate. The rule: a recursive call reads its arguments from left to right, and each argument must be passed unchanged until one of them is a smaller part of its parameter. Put the argument that shrinks first.

```
import Base

def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}:
      acc
    case Con{head, tail}:
      sum(tail, (acc + head : U32))

def main() -> U32:
  sum([1, 2, 3, 4], 0)
```

```
10
```

`tail` is a smaller part of `xs`, which the `match` exposed, and `xs` is the first argument. The accumulator changes freely because it comes after the shrinking argument.

Counting with a machine integer does not work, because subtraction gives the checker no structural evidence. Count with a `Nat` and match its predecessor. The parameter often called *fuel* bounds a loop that the outside world decides when to stop:

```
import Base

def count_down(fuel: Nat, acc: U32) -> U32:
  match fuel:
    case 0n:
      acc
    case 1n+remaining:
      count_down(remaining, (acc + 1 : U32))

def main() -> U32:
  count_down(5n, 0)
```

```
5
```

A recursive call that passes its input unchanged is rejected:

```
import Base

def loop(n: Nat) -> Nat:
  loop(n)

def main() -> Nat:
  loop(1n)
```

```
- expected : a decreasing self-call
```

Mutual recursion is not allowed in a safe definition. Merge the two functions into one, with an extra argument that selects which branch to run. Chapter 11 covers the `@unsafe` escape hatch and why a law must never depend on it.

MATCHING IN DEPTH

Patterns nest. This one inspects the second element of a list:

```
import Base

def second(xs: List<U32>) -> U32:
  match xs:
    case Nil{}:
      0
    case Con{h, Con{k, rest}}:
      k
    case Con{h, Nil{}}:
      h

def main() -> U32:
  second([4, 9, 7])
```

```
9
```

`match` can take several scrutinees, and `_` matches anything:

```
import Base

def both(a: Nat, b: Nat) -> Nat:
  match a b:
    case 0n 0n:
      0n
    case 0n 1n+q:
      1n+q
    case 1n+p 0n:
      1n+p
    case 1n+p 1n+q:
      Nat.add(p, q)

def first_or_zero(a: Nat, b: Nat) -> Nat:
  match a b:
    case 0n _:
      0n
    case 1n+p _:
      1n+p

def main() -> Nat & Nat:
  (both(3n, 4n), first_or_zero(5n, 7n))
```

```
(5n, 5n)
```

Three rules govern every `match`:

1. The scrutinee is a parameter or a variable bound by an earlier pattern. `match U32.is_eq(1, 1):` is rejected (chapter 01).
2. Matches are exhaustive. This one omits `False{}`:

```
import Base

def pick(b: Bool) -> U32:
  match b:
    case True{}:
      1
```

```
- expected : cases for False
```

3. Scrutinees follow binder order. A `let` cannot come before a `match` on a parameter.

ABSENCE WITHOUT NULL: MAYBE

Bend has no null. A value that may be missing has type `Maybe<&2, A>`: `None{}` means no value, and `Some{x}` wraps one.

```
import Base

def add_strings(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a)
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> Maybe<&2, U32>:
  add_strings("40", "2")
```

```
Some{42}
```

Inside `do Maybe<&2, U32>:`, the first `None` ends the block with `None`, as the guide describes. `Maybe.default` turns the result back into a plain value:

```
import Base

def add_strings(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a)
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> U32:
  Maybe.default(&2, U32, add_strings("40", "oops"), 0)
```

```
0
```

A division that refuses to divide by zero uses the helper pattern from chapter 01: the helper matches its own parameter. `b` is marked `+` because the function uses it twice:

```
import Base

def pick_div(zero: Bool, a: Nat, b: Nat) -> Maybe<&2, Nat>:
  match zero:
    case True{}:
      None{}
    case False{}:
      Some{Nat.div(a, b)}

def safe_div(a: Nat, +b: Nat) -> Maybe<&2, Nat>:
  pick_div(Nat.is_eq(b, 0n), a, b)

def main() -> Maybe<&2, Nat>:
  safe_div(10n, 3n)
```

```
Some{3n}
```

IMPORTS AND MODULES

An import gives a file a name. The alias is local to the importing file:

numbers.bend

```
import Base

def three() -> Nat:
  3n
```

main.bend

```
import Base
import ./numbers.bend as Numbers

def main() -> Nat:
  Numbers.three()
```

```
3n
```

`import Base` needs no alias. Imports come before definitions in a file.

DEFINITION ORDER

Bend checks each definition against the definitions above it. Calling a function written below it is an error:

```
import Base

def main() -> U32:
  answer()

def answer() -> U32:
  42
```

```
- expected : a defined name
```

Move `answer` above `main` and the program checks. Datatypes follow a related rule. In `bend 2.0.25`, a constructor field may only name a type that is already defined above it:

```
import Base

type Outer is Data:
  Wrap{inner: Inner}

type Inner is Data:
  Leaf{n: Nat}

def main() -> Nat:
  0n
```

```
- expected : a defined name
```

The bend2.dev guide says datatypes may appear in any order. The compiler we checked does not allow it, so define the inner type first.

EXERCISES

1. Length of a list. Write `length(xs)` that returns the number of elements as a `Nat`. The head is not needed, and an unused binder is dropped without any annotation.

```
import Base

def length(xs: List<U32>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{head, tail}:
      1n+length(tail)

def main() -> Nat:
  length([1, 2, 3, 4])
```

```
4n
```

2. Largest element. Write `max_of(xs, best)` that folds a list and keeps the larger of each element and the running best. Use `U32.max`, which is in Base.

```
import Base

def max_of(xs: List<U32>, best: U32) -> U32:
  match xs:
    case Nil{}:
      best
    case Con{head, tail}:
      max_of(tail, U32.max(head, best))

def main() -> U32:
  max_of([3, 9, 4], 0)
```

```
9
```

NEXT

Chapter 03 explains why a plain binder can be used only once, and what `+` costs.

C.10 · bend-gme/version-1/.opencode/skills/bend2-language/references/03-syntax-reference.md

--- title: "Bend 2 tutorial 12: syntax reference" description: "The Bend 2 language on one sheet: the three layers (kinds, types, values), erasure and quantities, every syntax form, operator namespaces versus type ascriptions, constructors, laws and proofs, evaluation, and the checker's messages for the common mistakes." updated: 2026-10-28 verified: 2026-10-18 status: runnable canonical: /tutorial/12-syntax-reference/ sources:

- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md `(https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)` ---

12: syntax reference

This page is the language on one sheet. Read sections 1 and 4 once, then use the tables as a lookup. Section 3 separates the two colon forms: `(e : T)` selects an operator namespace, while `{t : T}` ascribes a type. Blocks marked `bend` are checked by the tutorial gate with `bend 2.0.25`. The table rows were also run with that compiler when this page was written, except the forms that section 13 lists as not run. Section 12 gives the checker's exact messages for the common mistakes.

1. THE MODEL

Three layers. Each one classifies the layer below it.

Layer	Classifies	Examples
Kind	types	`Type` (`Kind(&1)`: used at most once), `Data` (`Kind(&2)`: copied with `+`)
Type	values	`Nat`, `U32`, `Shape`, `U32 -> U32`, `Maybe<&2, Nat>`, `Array<U32>`, `IsZero(n)`
Value	the bottom layer	`6n`, `Point{}`, `(1, 2)`, `[1, 2]`, `x => e`, `[0 : U32*8n]`

Types are terms, so a definition can return one. The guide splits a program in two. *Live* code runs. *Dead* code is types, erased arguments and proofs: the checker reads it, and the compiled program never runs it.

```
import Base

def twice(+n: Nat) -> Nat:
  Nat.add(n, n)

def identity(-A: Type, x: A) -> A:
  x

def main() -> Nat:
  identity(Nat, twice(3n))
```

```
6n
```

In that call, `identity` is code. `Nat` is an erased argument, so the compiled program never receives it. `twice(3n)` is a value computed at run time.

2. DECLARATIONS

Form	Meaning
`import Base`	the prelude; every file starts with it
`import ./m.bend as M`	the definitions of `m.bend` are named `M.x`
`type T is Data:`	a datatype whose values may be reused; `is Type` makes it affine
`type D<a, -A: Kind(a)> is Kind(a):`	a datatype whose kind comes from a parameter; one constructor per line, as `K{x: A}`
`def f(x: A, -y: B, +z: C) -> T:`	a definition: `x` is used at most once, `-y` is erased, `+z` is reusable; the body follows the `:`
`def t(~g: A -> B, x: A) -> B:`	a template parameter; template parameters come first
`law f:` with `def f(x):`	a claim, and its proof: a def with the same name
`@unsafe def f(...)`	skips the termination check; no law may depend on it
`def e(x: A) -> IO(B):` with `import "./e.c"`	a foreign effect built from a host file (section 13)

```
import Base

type Box<a, -A: Kind(a)> is Kind(a):
  Box{value: A}

def unbox(a, -A: Kind(a), b: Box<a, A>) -> A:
  match b:
    case Box{value}:
      value

def main() -> U32:
  unbox(&1, U32, Box{7})
```

```
7
```

3. MARKS AND PUNCTUATION

Mark	Meaning	Example
`name: T`	has type: a parameter, a field, a `for` binder	`x: U32`
`(e : T)`	operator namespace: resolves the operators in its expression group to the head of `T`; not a general type ascription	`(a + b : U32)`
`{t : T}`	type ascription of one term: checks that `t` has type `T`	`{12 : U32}`, `{x : U32}`
`K{...}`	a constructor (ctor); the brace touches the name	`Segment{3n}`
`{a == b : T}`	equality: a proposition	`{Nat.add(x, 0n) == x : Nat}`
`{a != b : T}`	negated equality, the same as `{a == b : T} -> Empty`	`{1n != 0n : Nat}`

Mark	Meaning	Example
`{==}`	reflexivity: proves an equality when both sides compute to one term	
`->`	separates the parameters from the result type	
`=`	binds a value; `a : ` never means equals	`x = v`
`<-`	binds the result of an effect inside `do`	`x : T <- m`
a line ending in `:`	opens a block: a body, a `case`, a `do`, a `law`, a `type`	
`#`	a comment to the end of the line	
`?name`, `?TODO`	a hole: `?name` shows the goal; `?TODO` leaves it open	

- A name directly before braces makes a constructor. Braces with no name make an annotation (`:`), an equality (`==`), or a proof (`{==}`).
- No space goes between a constructor and its braces. `Some{4n}` is a constructor, and `Some {4n}` is rejected.
- Operators need a space on each side: `a + b`, not `a+b`.
- `==` appears only in types. Compare values with `U32.is_eq(a, b)`.
- Signatures write `IO(A)` and `Chan(A)` with parentheses, and datatypes with angle brackets, as in `Maybe<&2, Nat>`. A `do` header takes angle brackets: `do IO<Unit>:`.

OPERATOR ANNOTATIONS ARE NOT TYPE ASCRIPTIONS

`(e : T)` and `{t : T}` both contain a colon, but they do different jobs.

- `{t : T}` checks one term: `t` must have type `T`. It is the ordinary ascription form, as in `{12 : U32}` or `{x : U32}`.
- `(e : T)` selects the namespace for the operators in its expression group. Each `+`, `-`, `<`, and so on becomes a call to a definition named after the head of `T`: `(a + b : U32)` is `U32.add(a, b)`, and `(a <= b : U32)` is `U32.is_le(a, b)`.
- The result type is whatever the selected function returns. It need not be `T`: the comparison above returns `Bool`.
- With no operator inside, `(e : T)` has no effect. `(x : U32)` is just `x`, so a `Nat` `x` remains a `Nat`; write `{x : U32}` to require `U32`.

Written	Resolves to	Result
`(a + b : U32)`	`U32.add(a, b)`	`U32`
`(a <= b : U32)`	`U32.is_le(a, b)`	`Bool`
`(x : U32)`	`x`	the type of `x`; `U32` is ignored

The namespace is the head of `T`: `(a + b : List<U32>)` asks for `List.add`, not `U32.add`. There is no polymorphic `+`, and the checker does not use the operands or the declared result type to choose a namespace. That is why a def's `-> U32` cannot replace `( ... : U32)`, and why a bare `x + y` is rejected even in a function that returns `U32`.

Parentheses are part of the operator form. Braces do not resolve operators, so `{a + b : U32}` fails; `(a + b : U32)` works. The annotation reaches the operators in its own expression group; a call argument starts a new group, so in `(U32.add(a + b, 1) : U32)` the `a + b` still needs its own parentheses.

4. KINDS, QUANTITIES, AND ERASURE

Form	Meaning
`Type = Kind(&1)`	affine: a value may be used at most once
`Data = Kind(&2)`	reusable: `+` may copy a value of this kind
`Kind(a)`	the kind is a parameter, with `a` equal to `&1` or `&2`. The guide also lists `&0`; this tutorial does not use it

Form	Meaning
`a <&> b`	the smaller of two quantities, which a type holding two element types takes (from the guide; not run here)
`is Data` , `is Type`	the kind of a declared type
`List<U32> = List<&1, U32>` ; `+List<U32> = List<&2, U32>`	`+` in front of a type gives the reusable form
`-x`	erased: exists only for the checker, appears only in types and proofs, and is deleted
`x`	affine: used at most once; dropping it is free
`+x`	reusable: the type must be `Data`
`~f`	template: code substituted at compile time; its argument is closed; one copy per distinct argument
`f(Nat, ...)`	an erased argument is passed explicitly, with no marker at the call
`a` in a parameter list	short for `-a: Quant`

```
import Base

def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n:
      Nil{}
    case 1n+p:
      x <> replicate(A, p, x)

def main() -> List<U32>:
  replicate(U32, 3n, 7)
```

```
[7, 7, 7]
```

```
import Base

def dup_k(-A: Kind(&2), +x: A) -> A & A:
  (x, x)

def main() -> Nat & Nat:
  dup_k(Nat, 6n)
```

```
(6n, 6n)
```

`replicate` mixes three kinds of parameter. `-A` is erased, `n` is affine, and `+x` is reusable because each step uses `x` twice. `<>` puts the head in front of the rest.

- A `Type` argument is written at the call: `identity(U32, 42)` . Bend infers no type arguments (section 12).
- `+` needs `Data` . A function is a `Type` and cannot be copied, so pass a template `~f` instead (chapter 03).
- An erased argument does not reach the compiled program. Compile the same program with `-A` and without it, and compare the C: the version without `-` passes the type in and drops it (chapter 03).

5. TYPES

Form	Meaning
`Nat` , `U32` , `F32` , `Bool` , `Char` , `String`	base types; a `Nat` literal ends in `n` , as in `3n`
`Unit` , `Empty`	one value, `Unit{}` ; no value

Form	Meaning
`A -> B`	a function, which is a `Type`, so a closure is affine
`@x: A -> B`	a dependent function: `B` may name `x`
`@-x: A -> B`	a dependent function whose argument is erased
`A & B`	a pair
`&x: A -> B`	a dependent pair: `B` may name the first component `x`
`A | B`	a sum: `Inl{a}` or `Inr{b}`
`D<A>`, `D<&2, A>`	a datatype applied to arguments
`Array<A>`	a one-owner array, which is a `Type`
`IO(A)`, `Chan(A)`	an effect that returns `A`; a channel of `A`
`Result<&1, &1, E, A>`	a fallible result: error `E`, value `A`
`{a == b : T}`, `{a != b : T}`	equality and its negation: propositions
a def that returns `Type`	a type computed from values, such as `IsZero(n)`

```
import Base

def IsZero(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      Empty

def needs_ne(e: {1n != 0n : Nat}) -> Nat:
  1n

def require_zero(n: Nat, e: IsZero(n)) -> Nat:
  n

def main() -> Nat:
  require_zero(0n, Unit{})
```

```
0n
```

```
import Base

def pick(b: Bool) -> U32 | Bool:
  match b:
    case True{}:
      Inl{4}
    case False{}:
      Inr{False{}}

def main() -> U32 | Bool:
  pick(False{})
```

```
Inr{False{}}
```

6. TERMS AND VALUES

Form	Meaning	Example
`42`, `1.5`, `3n`, `'c'`, `"s"`	literals: `U32`, `F32`, `Nat`, `Char`, `String`	`(6 * 7 : U32)`
`K{a, b}`	a constructor applied: `Ctor(K, [a, b])`	`Segment{3n}`
`(a, b)`	a pair	`(19, 23)`
`[a, b]`	a list: `Con{a, Con{b, Nil{}}}`	`[5, 10, 15]`
`h <> t`	a head placed in front of a list	`x <> replicate(A, p, x)`
`x => e`	a lambda: a closure, which is affine	`x => (x + 1 : U32)`
`+x => e`	a lambda whose binder is reusable	`+y => (y + y : U32)`
`f(a, b)`	a call	`twice(3n)`
`f!(a)`	a call on the GPU, with the parallel calls inside it; the CPU on a machine without a GPU	`pow2!(12n)`
`t(~g, a)`	a call with a template argument	`twice(~inc, 40)`
`(e : T)`	operator namespace for the expression; see section 3	`(a + b : U32)`
`{t : T}`	checked type ascription of one term	`+t = {Tok{3n} : Tok}`
`[v : T*n]`	an array of `n` slots; `n` is a power of two	`[0 : U32*8n]`
`[v : T^d]`	an array of `2^d` slots	`[0 : U32^3n]`
`a[i]`, `a[i] <- v`	a read returns the array with the element; a write returns the array. Indexes wrap	`a[9]` on eight slots is `a[1]`

```
import Base

def inc(x: U32) -> U32:
  (x + 1 : U32)

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~inc, 40)
```

```
42
```

```
import Base

def value(r: Array<U32> & U32) -> U32:
  match r:
    case (array, item):
      item

def main() -> U32:
  a = [0 : U32*8n]
  a[9] <- 7
  value(a[1])
```

```
7
```

- A constructor takes its type from its context. In a `let` with no context, annotate it: `+t = {Tok{3n} : Tok}` (chapter 03).
- The braces after a constructor name hold its fields in a `type` (`Segment{length: Nat}`), its arguments in a term (`Segment{3n}`), and its binders in a pattern (`case Segment{length}:`) (chapter 02).
- There is no `if`. Branch with `match` on `True{}` and `False{}` (chapter 01).

7. STATEMENTS AND MATCHING

Form	Meaning
`x = v`	a `let`: affine
`+x = v`	a reusable `let`: the type must be `Data`
`-x: T = v`	an erased `let`: the type is required
`x: T = v`	a typed `let`: required inside `do`
`(a, b) = v`, `K{x, y} = v`	a destructuring `let`
`a b = f(x) g(y)`	parallel calls: both start, and the line continues when both finish
`match x:`, `match a b:`	match on parameters, or on variables bound by a pattern; never on a computed call
`case K{x}:`, `case 1n+p:`, `case _ _:`	patterns; `_` matches anything; a case body may sit on the same line
`case K{+r}:`	a field bound with `+` is reusable; a plain field binder is affine, so a second use is rejected
`do M<xs, R>:`	a monadic block (section 10)

```
import Base

type Pt is Data:
  Pt{x: U32, y: U32}

def sum_pt(p: Pt) -> U32:
  Pt{x, y} = p
  (x + y : U32)

def swap(p: U32 & U32) -> U32 & U32:
  (a, b) = p
  (b, a)

def main() -> U32 & U32:
  swap((sum_pt(Pt{3, 4}), 0))
```

```
(0, 7)
```

- A recursive call must make a smaller argument. Arguments are read left to right, and each is passed unchanged until one of them is a smaller part. Otherwise the checker prints `expected : a decreasing self-call` (chapter 02).
- `@unsafe` turns that check off, and section 9 says what that costs.

8. OPERATORS

Form	Operand type	Example
`+ - * / %`	the `T` in the annotation	`(a * b : U32)`
`.&. .|. .^.`	the `T` in the annotation	`(6 .|. 1 : U32)`
`<< >>`	the `T`; the shift count is a `Nat`	`(1 << 3n : U32)`
`< <= > >=`	the `T`; the result is `Bool`	`(a <= b : U32)`

Form	Operand type	Example
`&& ||`	`Bool`	`(False{} || True{})`
`++`	`String`	`"ab" ++ "c"`
`T.is_eq(a, b)`	value equality	`U32.is_eq(a, b)`

```
import Base

def le(a: U32, b: U32) -> Bool:
  (a <= b : U32)

def main() -> U32 & U32 & U32 & U32 & Bool:
  ((6 .|. 1 : U32), (1 << 3n : U32), (8 >> 2n : U32), (7 % 2 : U32), le(3, 5))
```

```
(7, 8, 2, 1, True{})
```

- Each operator needs an annotation. `(a + b : U32)` covers the operators in its own parentheses. A call's arguments start a new group, so `(U32.add(a + b, 1) : U32)` is rejected (section 12).
- `T` is the operator namespace, not the result type. `(a <= b : U32)` calls `U32.is_le` and returns `Bool`; a def's `-> U32` does not replace the parenthesized `: U32`.
- Braces annotate one term. `{12 : U32}` is fine, but `{6 * 7 : U32}` is rejected, so write `(6 * 7 : U32)`.
- A bare operator is rejected. The guide says it means `Nat`, but 2.0.25 rejects it, so write `(n + 1n : Nat)`.

9. LAWS, PROOFS, AND PROPERTIES

A law is a claim, and a def of the same name is its proof. The proof of `add_zero` splits the cases of `x`, uses reflexivity in each, and rewrites with the induction hypothesis.

```
import Base

law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> Nat:
  Nat.add(42n, 0n)
```

```
42n
```

Form	Meaning
`law f:`	a claim; its binders and the claim follow, indented
`for x: T`	a universal binder; `for +x: T` makes it reusable
`for y: B where P(y)`	`y` is a pair: the value and a proof of `P(y)`
`exs z: T`	a witness: the proof returns `(z, proof)`
`def f(x):`	the proof of law `f`: a def with the same name
`def M.f(x):`	the proof of a law `f` imported from module `M` (chapter 15)

Form	Meaning
`match x:` in a proof	case analysis: each case is a goal
`{==}`	proves an equality when both sides compute to one term
`%e : P`	rewrites the goal with the equation `e`; the next line continues the proof
`%e@E : P`	a named rewrite (guide): the binder before `@` is one name (section 12). This tutorial has no working example
`?name`, `?TODO`	a hole (section 3)

Properties the checker guarantees:

- A law with no def is an open claim. A hole is open too: `?TODO` makes `bend PROOF.bend` print `Error: 1 TODO found.` and refuse the file.
- A false law fails the gate. The gate prints `All terms check.` only when every law has a proof that checks (chapters 07 and 15).
- A safe recursive def terminates, because each recursive call shrinks an argument (section 7). `@unsafe` is outside these guarantees, so no law may depend on it (chapter 08).
- A plain value is used at most once. A second use is an error, `consumed more than once.`
- An erased value is deleted before the program runs (section 4).
- A call computes its arguments first (section 11).
- Nothing is inferred, so type arguments, `let` types and operator types are written (section 12).

10. EFFECTS AND DO

Form	Meaning
`IO(A)`	an effect that returns `A`
`do IO<A>:`	a block; its header takes angle brackets
`x : A <- m`	runs the effect `m` and names its result, with type `A`
`m`	a line that runs an effect whose result is `Unit`
`return v`	ends the block with a pure value. The last line is a `return` or an effect; a bare term is rejected
`IO.print(s)`, `IO.args()`, `IO.get_env(k)`	print a string; the command-line arguments; an environment variable, as a `Result`
`IO.die(-A, code, msg)`	ends the program with an exit code and a message
`IO.fork(A, m)`, `IO.join(A, c)`	start `m` in the background and get a channel for its result; wait for that result
`IO.spawn(A, m)`	starts `m` and does not wait for it
`Chan.new(A, n)`, `Chan.send(A, c, v)`, `Chan.recv(A, c)`, `Chan.close(A, c)`	a channel of `A` with room for `n` values; `Chan.recv` answers a `Maybe`, and `None{}` once the channel is closed

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    n: U32 = (6 * 7 : U32)
    IO.print(text ++ " " ++ U32.show(n))
```

```
Hello, Ada 42
```

11. EVALUATION: BEND IS STRICT

- A compiled Bend program is strict: a call computes its arguments before its body runs. A native build of `first(1n, spin(0n))`, where `spin` loops forever and `first` ignores its second argument, did not finish under `timeout 20`. With `0n` in place of `spin(0n)`, it printed `1` at once.
- The checker does not show evaluation order. `bend file.bend` normalises a pure `main`, so judge the order with a native build under `timeout`.
- To keep work unevaluated, use one of three forms: a lambda (`x => ...`), whose body runs when it is applied; a template (`~f`), which is substituted at compile time; or data that `eval` interprets, as below.
- Parentheses and annotations delay nothing. Tail calls compile to loops, and the runtime has no C stack: each definition is a segment of a flat state machine, and a call is a jump (the guide; chapter 05).

```
import Base

type Expr is Data:
  Lit{n: Nat}
  Add{a: Expr, b: Expr}

def eval(e: Expr) -> Nat:
  match e:
    case Lit{n}:
      n
    case Add{a, b}:
      Nat.add(eval(a), eval(b))

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32 & Nat & U32:
  later: U32 -> U32 = x => (x + 1 : U32)
  program : Expr = Add{Lit{2n}, Lit{3n}}
  (later(41), eval(program), twice(~(x => (x + 1 : U32)), 40))
```

```
(42, 5n, 42)
```

12. REJECTED FORMS

Each row is a program the checker rejects. The message is the checker's own, and the right column is the fix.

Wrong	The checker says	Right
`(a+b : U32)`	`expected : '='` , `observed : ':'`	`(a + b : U32)`
`x + y` with no annotation	`a type for this operator (write (a + b : Nat))`	`(x + y : U32)`
`{6 * 7 : U32}`	`a type for this operator (write (a * b : Nat))`	`(6 * 7 : U32)`
`(a == b)`	`expected : ')'` , `observed : '='`	`U32.is_eq(a, b)`
`price = 12`	`expected : an annotated term (cannot infer)`	`price = {12 : U32}`
`-n = 3n`	`expected : an annotated term (cannot infer)`	`-n: Nat = 3n`
`x = 3` inside `do`	`expected : a pattern (a binder or a constructor)`	`x : U32 = 3`
`do IO(Unit):`	`expected : '<'` , `observed : '('`	`do IO<Unit>:`
`-> IO<Unit>`	`a declared datatype (unknown: IO)`	`-> IO(Unit)`
`-> Maybe(&2, Nat)`	`expected : a family instance (write Maybe<..>)`	`-> Maybe<&2, Nat>`

Wrong	The checker says	Right
`Some(4n)`	`expected : a defined name , observed : Some`	`Some{4n}`
`Some {4n}`	`expected : 'def', 'type' or 'law' , observed : '{'`	`Some{4n}`
`use(+f: U32 -> U32)`	`expected : Data , observed : Type`	a plain `f: U32 -> U32` used once, or a template `~f`
`+t` on a `type Token is Type:`	`expected : Data , observed : Type`	declare `type Token is Data:`
`identity(42)` with `-A: Type`	`expected : Type , observed : U32`	`identity(U32, 42)`
`bad(-A: Type, x: A) -> A` returning `A`	`expected : A , observed : Type`	return `x`
`f(f(x))` with `f: U32 -> U32`	`expected : f , observed : f (consumed more than once)`	a template `~f`
a template argument that uses a local `k`	`a template applied to closed ~ arguments (k is a variable here, not comptime: pass it at run time)`	a top-level definition, as in `~inc`
`match U32.is_eq(1, 1):`	`a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)`	pass the call to a def and match its parameter
`[0 : U32*6n]`	`a power of two count (^d takes a depth)`	`[0 : U32*8n]` or `[0 : U32^3n]`
a `do` block that ends in `(6 * 7 : U32)`	`observed : U32`	end with `return v` or an effect
`loop(n)` calling itself with the same `n`	`expected : a decreasing self-call`	a call on a smaller part
`ih = add_zero(p)` in a proof, then `{==}`	`expected : 1n+Nat.add(p, 0n) , observed : 1n+p`	`%add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}`
`%add_zero(p)@ih : P`	`expected : a name before @ (a rewrite binder is one name: %e@E : P)`	`%add_zero(p) : P`
`?goal` in a case	`expected : {0n == 0n : Nat} , observed : ?goal`	fill the case with the goal, as `{==}`
`?TODO` left in a proof	`Error: 1 TODO found.`	a complete proof
a `where` pair used as a `Nat`	`expected : Nat , observed : Sigma<&1, &1, Nat, n => IsNonZero(n)>`	the claim must treat `n` as the pair; this tutorial does not show reading its value

13. NOT RUN HERE

- Foreign effects (`import "./e.c"` , `import "./e.js"`) need host files. `bend guide effects` describes them.
- `bend main.bend --publish` , `bend page.html -o dist` , and `import 0x<hash>/main.bend as P` need the package hub or the web bundler.
- A working named rewrite `%e@E : P` , and a `where` binder whose claim reads its value out of the pair (section 12).
- GPU timing: the lab machine has no GPU, so `!` runs on the CPU here.
- The `a <&> b` quantity combinator, which the guide describes and this page does not run.

NEXT

Chapter 13 maps these terms onto Haskell and OCaml. Chapter 15 uses the forms from this page in one complete program.

C.11 · bend-gme/version-1/.opencode/skills/bend2-language/references/04-rules-and-errors.md

--- title: "Bend 2 tutorial 11: checker rules and errors" description: "The rules the Bend 2 checker enforces, each official diagnostic with the message bend 2.0.25 prints, the errors this tutorial found beyond the official list, and the differences between 2.0.25 and the bend2.dev examples." updated: 2026-10-28 verified: 2026-10-17 status: runnable canonical: /tutorial/11-rules-and-errors/ sources:

- https://bend2.dev/errors/ (https://bend2.dev/errors/)
- https://bend2.dev/errors/diagnostics.json (https://bend2.dev/errors/diagnostics.json)
- https://bend2.dev/notes/ai-agents-bend/ (https://bend2.dev/notes/ai-agents-bend/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

11: checker rules and errors

This chapter is a reference. The checker runs before any program executes, and it stops with a message when a rule fails. Each rule below is taught in an earlier chapter, and each official diagnostic appears here with the text `bend 2.0.25` prints. The official index on bend2.dev was written against 2.0.36, and its wording differs in places. Both wordings are in the tables.

THE RULES

Rule	Why it exists	Chapter
A definition is defined above its first use	the checker reads top to bottom	02
Recursive calls shrink their first changing argument	termination makes proofs sound	02, 08
Matches are exhaustive	every value must have a branch	02
A match inspects a parameter or a pattern variable	a computed value has no evidence to split	01, 03
Every operator carries a type annotation	the checker does not infer arithmetic types	01
A constructor in a `let` needs its type from context	the checker cannot infer it otherwise	02
A plain binder is used at most once	ownership and freeing	03
`+` marks reuse of a `Data` value only	only `Data` may be copied	03
The result has the declared type	ordinary typing	01
Every name is defined in the file or an import	no silent fallbacks	02
Every law has a same-named proof, with no holes	the gate	07
A proof does not depend on `@unsafe` or foreign code	unsafe code is outside the guarantee	08, 11

THE OFFICIAL DIAGNOSTICS

Each heading names the diagnostic on bend2.dev. The code is the broken program from the official catalogue, and the error lines are what `bend 2.0.25` printed for it. Where a chapter already shows the program and its message, this page links to it.

A VARIABLE WAS CONSUMED MORE THAN ONCE

Chapter 03 shows the program and its fix. The message is `- expected : n` and `- observed : n (consumed more than once)`.

AN OPERATOR HAS NO TYPE ANNOTATION

Chapter 01 shows the program and its fix. The message is `- message : a type for this operator (write (a + b : Nat))`. The annotation chooses the operator's namespace, as in `(a + b : U32)` resolving to `U32.add(a, b)`; a def's `-> U32` cannot replace it (chapter 12, section 3).

A MATCH CANNOT INSPECT A COMPUTED VALUE

Chapter 01 shows the program and its fix. The message is `- message : a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)`.

A FUNCTION CALLS ONE DEFINED BELOW IT

Chapter 02 shows the program and its fix. The message is `- expected : a defined name`.

A RECURSIVE CALL DOES NOT DECREASE

Chapter 02 shows the program and its fix. The message begins `- expected : a decreasing self-call`.

A NAME IS NOT DEFINED

```
import Base

def main() -> Nat:
  missing()
```

```
- expected : a defined name
- observed : missing
```

Define `missing` above `main`, or import the file that defines it.

A LAW IS MISSING ITS PROOF

Chapter 07 shows the program. The message is `Error: 1 TODO found.`, and the same file with `{==}` as its proof prints `All terms check.`

DEFINITIONS RELY ON UNSAFE OR FOREIGN CODE

Chapter 08 shows the case that matters. The official program in this catalogue uses names that end in `?`, which bend 2.0.25 cannot parse:

```
import Base

def is_even?(n: Nat) -> Bool:
  match n:
    case 0n: True{}
    case 1n+p: is_odd(p)
```

```
- expected : '('
```

The fix in 2.0.36 is `@unsafe`, and 2.0.25 accepts that spelling too. The rule itself: a law that depends on an `@unsafe` definition is not verified. Chapter 08 shows the warning this version prints.

THE RESULT HAS THE WRONG TYPE

```
import Base

def main() -> U32:
  True{}
```

```
- expected : U32
- observed : Bool
```

Return a value of the declared type.

ERRORS THIS TUTORIAL FOUND BEYOND THE OFFICIAL LIST

These errors were produced by real programs while writing the chapters. They are not in the official index.

A superposition literal. `&{1, 2}` is rejected (chapter 06).

An `if` statement. Bend has no `if` (chapter 06).

A constructor without a type. A `let` with no context cannot infer the type of a constructor:

```
import Base

type Tok is Data:
  Tok{n: Nat}

def show(t: Tok) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  t = Tok{3n}
  show(t)
```

```
- expected : an annotated term (cannot infer)
```

Write `t : Tok = Tok{3n}`, as chapter 02 does.

A quantity argument left out. `List.length` takes its quantity first, so `List.length(String, args)` fails:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    args : List<String> <- IO.args()
    IO.print(Nat.show(List.length(String, args)))
```

```
- expected : Quant
```

Write `List.length(&1, String, args)`, as chapter 09 does.

A destructured computed value. `(a, b) = f(x)` inside a definition is a match on a computed value (chapter 03).

A hole inside a branch. `?name` stops the proof check and prints the goal (chapter 07).

A constructor field naming a later type. Chapter 02 shows the message.

VERSION DIFFERENCES

The index page is the table of contents. The table below compares the compiler installed for this tutorial, bend 2.0.25, with the 2.0.36 build the bend2.dev examples were checked against.

Topic	bend 2.0.25 (checked here)	bend 2.0.36 (site)
Proof success line	`All terms check.`	`ALL PROOFS CHECK and Use --verdict for mathematical validity.`
`--verdict` option	`unknown option --verdict`	supported
Proof that depends on `@unsafe`	prints `All terms check, but N defs rely on unsafe or foreign code` and exits 0	fails the proof check
Datatype fields naming a later type	rejected (`expected : a defined name`)	the guide says datatypes may appear in any order
`f?` names	parse error	accepted as shorthand for `@unsafe`

Topic	bend 2.0.25 (checked here)	bend 2.0.36 (site)
Forward reference message	`- expected : a defined name`	`- expected : a filled definition (an unfilled law is a dead claim: live code cannot use it)`

Two consequences for your workflow. First, the gate must read the output, not only the exit status, because 2.0.25 can exit 0 with an unsafe dependency. Second, `bend update` installs the latest compiler. The site's examples use 2.0.36, which has `--verdict`. The bend2.dev notes say that option needs Lean v4.34.0 exactly, or a prebuilt kernel named by `BENDTT`. Chapter 07 explains what the final check adds. This lab has neither the compiler option nor a Lean toolchain, so it has not run that check.

READING AN ERROR

Every message has the same shape: a headline, then `- expected` and `- observed` lines, and sometimes a `Location` with the source around it. Read `expected` first. It names what the checker needed at that point, such as `a defined name`, `Data`, or `cases for False`. Then find the same spot in your file. The `observed` line shows what it found instead.

NEXT

Chapter 12 is the syntax reference, and chapter 13 is the glossary.

C.12 · bend-gme/version-1/.opencode/skills/bend2-language/references/05-first-programs-concepts.md

--- title: "First programs" description: "Functions, bindings, conditionals, numbers, strings, tuples, lists, and imports in Bend2 syntax." updated: 2026-09-05 verified: 2026-09-05 status: docs-only canonical: /learn/first-programs/ sources:

- https://bend2.dev/learn/ `(https://bend2.dev/learn/)` ---

First programs

Functions, bindings, branching, numbers, strings, tuples, lists, and imports — the first ten Bend2 concepts, in the order a new programmer meets them.

FUNCTIONS

```
import Base

# Adds two numbers.
def add(a: U32, b: U32) -> U32:
  a + b

def main() -> U32:
  add(20, 22)
```

`def` names the function, parameters carry types, the return type follows the arrow, and the body is an expression — no `return`.

LET BINDINGS

Tuple assignment splits two calls at once (the fork from parallelism). A single binding is the same shape:

```
def main() -> U32:
  x = 40 + 2
  x
```

CONDITIONALS

Branching goes through `match` on numbers and constructors.

```
def describe(n: U32) -> U32:
 match n:
   case 0:
     0
   case 1+d:
     1
```

BOOLEANS

Boolean constructors and operators *(planned)*.

NUMBERS

Numbers are typed as `U32`, including successor-style matching with `case 0:` and `case 1+d:`.

```
def identity(n: U32) -> U32:
 n
```

STRINGS AND CHARACTERS

Strings concatenate with `++` and print through `IO::print` (see hello). Characters and escaping are planned.

TUPLES

Tuples destructure in assignment position: `a, b = f(x), g(x)`.

LISTS

List syntax *(planned)*.

IMPORTS

`import Base` brings in the standard library, including `IO`.

C.13 · bend-gme/version-1/.opencode/skills/bend2-language/references/06-data-matching-concepts.md

--- title: "Data and matching" description: "Algebraic data types, records, maps, pattern matching, recursion, higher-order functions, folds, and the bend construct." updated: 2026-09-05 verified: 2026-09-05 status: docs-only canonical: /learn/data-matching/ sources:

- https://bend2.dev/learn/ `(https://bend2.dev/learn/)` ---

Data and matching

Data definitions and the many shapes of `match`: algebraic data types, records, maps, pattern matching, recursion, and folds.

ALGEBRAIC DATA TYPES

Algebraic data types introduce constructor-shaped values such as `Nat`.

RECORDS

Records *(planned)*.

MAPS

Maps *(planned)*.

PATTERN MATCHING

The proofs sample matches `Nat` by constructors — brace-suffixed patterns, one arm per constructor:

```
def is_zero(n: Nat) -> Bool:
  match n:
    case Z{}:
      True
    case S{p}:
      False
```

Numeric matching works the same way (`case 0:` / `case 1+d:` , binding the predecessor).

LAMBDA MATCH

Lambda match *(planned)*.

RECURSION

Recursion is ordinary self-call, as in the parallel sum.

MUTUAL RECURSION

Mutual recursion *(planned)*.

HIGHER-ORDER FUNCTIONS

Higher-order functions *(planned)*.

CLOSURES

Closures *(planned)*.

FOLDS

Folds *(planned)*.

THE BEND CONSTRUCT

The `bend` construct that gives Bend its name awaits the public compiler.

C.14 · `bend-gme/version-1/.opencode/skills/bend2-proofs/SKILL.md`

Bend 2: laws and proofs

A law is a type; its proof is a program; the checker runs it. If the gate passes, the property holds for every input.

MODEL

- `law name: for x: T {lhs == rhs : T}` states a property. `def name(x): …` with the **same name** proves it.
- `{==}` closes a goal when both sides compute to the same value.
- `match` refines the goal per case; `%name(p) : {goal}` uses the law for a smaller `p` (induction) as a rewrite.
- Holes mark unfinished proofs; the checker reports what is left.
- Dependent types: types computed from values (`Unit` , `Empty` , length-indexed lists).
- **Totality**: every recursion must shrink; `@unsafe` turns that check off and with it the guarantee.
- A law proves what a program **computes**. It says nothing about cost, speed, the JS host, or the FFI.
- A law with **no same-named `def`** is refused (`expected: a filled definition … an unfilled law is a dead claim`), so a half-finished claim cannot silently ship.
- **No law may state a fact about `F32` .** Every `F32.*` operation in Base is a bodyless axiom; the checker cannot reduce it, so a float "proof" would either hang or prove nothing. Float behaviour belongs to JS-lane tests.

COST AND PARALLELISM (MEASURED, BEND 2.0.39, 12 CORES)

A law's cost is the **range it normalizes**, not the size of the file. Two consequences decide the layout:

- The wall clock of a proof run is the **longest law**, not the sum. Measured on this repo: five law files whose sequential times are 0.2 / 0.5 / 4 / 6.4 / 166 s sum to ~177 s, and running all five at once took **146 s** — a **1.21×** gain, not the 5× the core count suggests. The longest law is ~85% of the work, so that is the ceiling. Parallelism is not a substitute for finer laws.

- Split an enumeration by **range**: the 2002–2030 calendar as one law is 166 s; per year it is ~6 s each, so eight at a time finish in ~25–30 s, and editing one year re-checks only that year.

So: small laws in `LAWS*.bend`, exhaustive ones in `LAWS_TABLE*.bend` (per range), and the inner loop proves only the law files a change reaches: `bun run --cwd packages/core proof:changed` (`proof.mjs --changed --fast`). It prints each file's time and flags any file over 5 s as `slow`. The full `proof` (with the tables) runs in the background and at the commit. Measure before claiming a speedup.

Size decides where a law lives (measured on this repo, bend 2.0.39):

A law that…	Costs	Lives in	Runs
is stated `for x` and proved by cases/induction	ms	the layer's `LAWS.bend`	inner loop (`proof:changed`)
checks one fixed example (`{==}` on literals)	ms–s	the layer's `LAWS.bend`, **labelled a checked example**	inner loop
enumerates a range (calendar years, every expiry)	s–minutes	`LAWS_TABLE_<range>.bend`, one per range	background / commit only

An enumeration is the last resort: use it only when no inductive proof is practical, and split it so that one edit re-checks one range. If a law file is `slow`, split it before adding anything to it: every change to any module it imports pays its time.

RULES

1. Draft each law in plain words first; submit drafts to the user in batches; state approved ones in `LAWS.bend`. A new invariant in a slice means a new law draft in the same slice.
2. **Per layer**: `bend/<layer>/LAWS\*.bend` holds the laws **and** their same-named proofs, so `bend bend/<layer>/LAWS.bend` is a gate on its own (there is no per-layer `PROOF.bend`). **Inner loop**: `proof:changed` (or `node proof.mjs <changed file> --fast`, or `--list` to see the selection). **Gate**: `proof` (`proof.mjs`, all `LAWS\*.bend`, in parallel) in the background before the commit. The root `PROOF.bend` / `PROOF_FAST.bend` are hand-kept import lists; prefer `proof.mjs`, which globs every law file.
3. **Prove as much as possible.** Every invariant expressible over integers and ADTs gets a law (codec round-trips, calendar, money, settlement, orders, ledger, planners, VM causality, risk caps). Strings, F32 numerics and the JS host are painful to prove: property-test them and say so.
4. Never prove a property of an `@unsafe` function and present it as unconditional.
5. **Keep laws small and many, and know which command is which.**

Question	Command	Cost
is this law true?	`bend bend/<layer>/LAWS.bend` (a law file checks itself)	0.3 s leaf – 21 s `store/LAWS_COMPACT`
are the laws about my change true?	`bun run --cwd packages/core proof:changed` (no `LAWS_TABLE*`)	0.1 s when nothing reaches, else the selected files
every law, including enumerations?	`bun run --cwd packages/core proof` (parallel, 4 at a time)	~166 s
kernel-valid, not just checker-valid?	`bend <LAWS file> --verdict`	needs Lean v4.34.0 (R-29); today's gate is checker-level

One law per failure it rules out, each about one module, each checking in well under a second. Importing a big module into a law file makes that file selected (and paid for) on every change to that module, so a law file imports only what its laws are about. 5b. **Split by cost class.** A cheap law checks in milliseconds and belongs in the layer's `LAWS.bend` (selected by `proof:changed`). A law that enumerates a whole range (2002–2030 calendar, every expiry) is minutes of normalization; put it in `LAWS_TABLE.bend` (or per-range files) imported only by the full gate. Splitting one enumeration does not lower the total; it localizes it and lets it run **in parallel** — but only a split by *range* shrinks the wall clock,

because that is set by the longest law. 5c. **Say what a law does not cover.** A closed-shape law (`decode(encode(x)) = x` for six shapes) is useful and cheap, but it is not the general theorem. State the limitation in the file header and point at the test or the induction that covers the rest.

6. An unapproved or failing law does not gate the build; it is listed in NOTES.md as open.

PATTERNS (CHECKED WITH BEND 2.0.39)

```
import Base

law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> Nat:
  Nat.add(42n, 0n)
```

Project layout (this repo; real examples: minebender, bend-runtime `hvm5/version1/`):

```
bend/num/LAWS.bend        import Base; import ./money.bend as M
                          law money_neg_neg: for x: M.Money {M.neg(M.neg(x)) == x : M.Money}
                          def money_neg_neg(x): {==}          # statements AND proofs
bend/time/LAWS_TABLE_TIME.bend   the exhaustive range laws (minutes, full gate only)
bend/PROOF.bend           hand-kept import list (root gate); `proof.mjs` globs every LAWS*.bend instead
bend/PROOF_FAST.bend      hand-kept cheap subset (num + data); superseded by `proof:changed`
bend/<layer>/_probe.bend  scratch; `_`-prefixed files are never gates
```

PITFALLS

Symptom	Cause	Fix
goal does not reduce under `{==}`	the function is stuck on a variable	`match` on that variable first
induction step fails	missing rewrite with the smaller case	add `%law(p) : {...}` before `{==}`
`--verdict` fails: `no such file … bendtt.lean`	the install missed `bend/bend2/bendtt.lean`	copy it from the pinned tarball (it ships there)
`--verdict` fails: `lean: Executable not found`	the kernel needs Lean v4.34.0 (elan) or `$BENDTT`	install Lean, or stay on the checker gate and say so (R-29)
proof passes but the app is wrong	the law states the wrong property, or covers only the model	review the plain-words draft; add tests at the FFI
`bend PROOF.bend` takes minutes	one law enumerates a whole range	split it by range into `LAWS_TABLE_<range>.bend` ; measure again
`proof:changed` flags a file `slow`	big enumeration or heavy imports in a loop file	move the enumeration to a `LAWS_TABLE_*` file; import only the module under test
`expected: a filled definition … unfilled law`	the `def` is missing, or it is declared **above** its `law` (this also fires on a forward reference)	define the `law` first, or move the callee above the caller

VERIFY

```
BEND_NO_TELEMETRY=1 bend bend/<layer>/LAWS.bend         # one law file (it checks itself)
bun run --cwd packages/core proof:changed              # inner loop: law files the change reaches, no tables
bun run --cwd packages/core proof > /tmp/proof.log 2>&1 &   # full gate, in the background
```

REFERENCES

File	Covers	Open when
`references/01-laws-and-proofs.md`	first law, equality by computation, Boolean laws by cases, equality as a relation, existentials, holes, LAWS/PROOF layout	writing any proof
`references/02-dependent-types-and-totality.md`	types that compute, length in the type, totality, `@unsafe`	indexed types or termination
`references/03-proofs-concepts.md`, `references/04-types-and-proofs-concepts.md`	concept pages	a quick refresher
`references/05-proof-lineage.md`, `references/06-bend-vs-lean.md`	where Bend's proofs come from; Bend vs Lean	judging what Bend can and cannot prove

RELATED

`bend2-language` (termination errors) · `bend2-app` (which laws an app needs) · `research-to-release` (laws in the delivery pipeline).

C.15 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/01-laws-and-proofs.md

--- title: "Bend 2 tutorial 07: laws and proofs" description: "State a property as a law, prove it with a same-named def, read the proof goals in each match branch, use induction and rewrites, handle holes, and run the PROOF gate." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/07-laws-and-proofs/ sources:

- https://bend2.dev/learn/proofs/ (https://bend2.dev/learn/proofs/)
- https://bend2.dev/learn/induction/ (https://bend2.dev/learn/induction/)
- https://bend2.dev/learn/equality/ (https://bend2.dev/learn/equality/)
- https://bend2.dev/learn/holes/ (https://bend2.dev/learn/holes/)
- https://bend2.dev/learn/types-as-specifications/ (https://bend2.dev/learn/types-as-specifications/)
- https://bend2.dev/notes/ai-agents-bend/ (https://bend2.dev/notes/ai-agents-bend/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

07: laws and proofs

A **law** is a claim about a program, written as a type. Bend has no tactics and no proof search, so a proof is an ordinary definition with the same name as the law. The compiler checks that the definition has the claimed type. If you know Haskell, think of a QuickCheck property that the compiler must check for every input, not just sampled ones. If you know Lean, the table below maps the ideas, with one big difference: Bend has no tactics and no proof search, so every step is a term you write.

> **Core insight.** A law is a type, and a proof is a program of that type. This is the Curry-Howard idea: a claim is a type, and a term that inhabits it is its proof. `bend PROOF.bend` accepts a proof only if its def has the law's type, its recursive calls terminate, and each `{==}` holds because both sides compute to the same term.

Composed of: a law and its proof.

```
law add_zero                    the type: a claim
- for x: Nat                    a binder
- {Nat.add(x, 0n) == x : Nat}   the claim

def add_zero(x)                 the program, with the law's name
- one case per constructor of x
- each case: {==}, or a rewrite %e : P followed by {==}
```

Lean idea	Bend 2	Note
a theorem statement and its proof	`law name:` and a same-named `def name(...)`	the law is the claim; the def is the proof
`rfl`, closing a goal by computation	`{==}`	both sides must compute to the same term
`rw [h]`	`%h : P`	`P` is the goal, with `_` where the right-hand side goes
`induction n with`	`match n:` plus a recursive call	the recursive call is the induction hypothesis
`∃ w, P w`	`exs w: T` in the law	the proof returns `(w, proof)`
`sorry`	`?TODO`	the gate rejects any open hole
`simp`, `omega`, other tactics	nothing	write the rewrite or the case split yourself

YOUR FIRST LAW: INDUCTION

This law says that adding zero to any natural number does nothing:

```
import Base

law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> Nat:
  Nat.add(42n, 0n)
```

```
42n
```

Read the file in three parts.

- `law add_zero:` names the claim. `for x: Nat` quantifies over every natural. The equation `{Nat.add(x, 0n) == x : Nat}` is the claim itself: the two sides are equal at type `Nat`.
- `def add_zero(x):` is the proof. It has the same name as the law and one parameter for each `for`.
- `match x:` splits the proof into cases, and each branch refines the goal.

Look at the goal in each branch. In the case `0n`, the left side `Nat.add(0n, 0n)` computes to `0n`, which is the right side, so `{==}` closes the goal. In the case `1n+p`, the left side is `Nat.add(1n+p, 0n)`, and it computes to `1n+Nat.add(p, 0n)`, which is not yet `1n+p`. The recursive call `add_zero(p)` is the **induction hypothesis**: it proves the claim for the smaller number `p`. The line `%add_zero(p) : {...}` rewrites the goal with that equation. The `_` marks where the right-hand side of the equation goes. After the rewrite, both sides are the same, and `{==}` closes the goal.

The induction hypothesis is allowed only because `p` is smaller than `x`: it is a pattern-bound variable from `1n+p`, so the termination rule of chapter 02 holds.

EQUALITY BY COMPUTATION

When both sides compute to the same term, `{==}` alone is enough. The law below holds because `Nat.add` matches on its first argument, so `0n` on the left reduces without any induction:

```
import Base

law zero_left:
  for x: Nat
  {Nat.add(0n, x) == x : Nat}

def zero_left(x):
  {==}

def main() -> Nat:
  Nat.add(0n, 5n)
```

```
5n
```

The same reasoning fails when the computed argument is a variable. This law puts the variable where the computation needs it, and `{==}` cannot prove it:

```
import Base

law twice_is_double:
  for x: Nat
  {Nat.add(x, x) == Nat.double(x) : Nat}

def twice_is_double(x):
  {==}
```

```
- expected : Nat.add(x, x)
- observed : Nat.double(x)
```

Both sides are correct, but `{==}` needs them to reduce to the same term, and the variable `x` blocks the reduction. Split the variable with `match` and prove each case, or use induction as `add_zero` does.

BOOLEAN LAWS BY CASES

A law over a finite type needs one `{==}` per constructor. This law says negating twice gives back the original value:

```
import Base

law not_not:
  for b: Bool
  {Bool.not(Bool.not(b)) == b : Bool}

def not_not(b):
  match b:
    case True{}:
      {==}
    case False{}:
      {==}

def main() -> Bool:
  Bool.not(Bool.not(True{}))
```

```
True{}
```

EQUALITY AS A RELATION

Equality laws state that two computations give the same result. The bend2.dev example proves that cancelling a job twice has the same effect as cancelling it once, for every state. The job type and its cancel function live in another file:

job.bend

```
import Base

type State is Data:
  Queued{}
  Running{}
  Cancelled{}
  Finished{}

def cancel(state: State) -> State:
  match state:
    case Queued{}: Cancelled{}
    case Running{}: Cancelled{}
    case Cancelled{}: Cancelled{}
    case Finished{}: Finished{}
```

main.bend

```
import Base
import ./job.bend as Job

law cancel_idempotent:
  for state: Job.State
  {Job.cancel(Job.cancel(state)) == Job.cancel(state) : Job.State}

def cancel_idempotent(state):
  match state:
    case Job.Queued{}: {==}
    case Job.Running{}: {==}
    case Job.Cancelled{}: {==}
    case Job.Finished{}: {==}

def main() -> Job.State:
  Job.cancel(Job.cancel(Job.Running{}))
```

```
job.Cancelled{}
```

The law is checked as a type, so `main` does not need to call it. The output prints the constructor with its module prefix, `job.Cancelled{}`.

Two equality lemmas in Base help you rewrite equations: `Equal.sym` turns `e : {a == b : T}` into `{b == a : T}`, and `Equal.trans` chains two equations. The next example uses `Equal.sym`:

```
import Base

law flip:
  for +a: Nat
  for +b: Nat
  for e: {a == b : Nat}
  {b == a : Nat}

def flip(a, b, e):
  Equal.sym(Nat, a, b, e)
```

```
All terms check.
```

EXISTENTIAL LAWS

`exs y: T` in a law asks for a witness. The proof returns the witness together with the proof of the equation:

```
import Base

law has_zero:
  exs y: Nat
  {Nat.add(y, 0n) == 0n : Nat}

def has_zero():
  (0n, {==})
```

```
All terms check.
```

HOLES: UNFINISHED PROOFS

A `?TODO` in a proof leaves that branch open. The checker counts open holes, and a file with any open hole fails the proof check:

```
import Base

law cancel_all:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def cancel_all(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      ?TODO
```

```
Error: 1 TODO found.
The code is incomplete, and not a valid proof yet.
```

A named hole, `?name`, stops the check and prints the goal that the branch must prove. Here the goal is printed for the branch `1n+p`:

```
import Base

law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      ?inspect
```

```
- expected : {1n+Nat.add(p, 0n) == 1n+p : Nat}
- observed : ?inspect
```

Use `?name` to see the goal you must prove, then replace it with a proof. Leave `?TODO` only while you write the file. The proof gate rejects both.

LAWS AND PROOF: THE PROJECT LAYOUT

A project keeps its claims and its proofs apart. `LAWS.bend` imports the code and states the laws. `PROOF.bend` imports both and proves each law with a def of the same name, as `def Laws.name` for a law named `name`. The example below stores three files in one directory. `job.bend` is the code under test, and it is the same file as in the equality example. `LAWS.bend` states four laws about it:

job.bend

```
import Base

type State is Data:
  Queued{}
  Running{}
  Cancelled{}
  Finished{}

def cancel(state: State) -> State:
  match state:
    case Queued{}: Cancelled{}
    case Running{}: Cancelled{}
    case Cancelled{}: Cancelled{}
    case Finished{}: Finished{}
```

LAWS.bend

```
import Base
import ./job.bend as Job

law cancel_idempotent:
  for state: Job.State
  {Job.cancel(Job.cancel(state)) == Job.cancel(state) : Job.State}

law cancels_queued:
  {Job.cancel(Job.Queued{}) == Job.Cancelled{} : Job.State}

law cancels_running:
  {Job.cancel(Job.Running{}) == Job.Cancelled{} : Job.State}

law preserves_finished:
  {Job.cancel(Job.Finished{}) == Job.Finished{} : Job.State}
```

PROOF.bend

```
import Base
import ./job.bend as Job
import ./LAWS.bend as Laws

def Laws.cancel_idempotent(state):
  match state:
    case Job.Queued{}: {==}
    case Job.Running{}: {==}
    case Job.Cancelled{}: {==}
    case Job.Finished{}: {==}

def Laws.cancels_queued():
  {==}

def Laws.cancels_running():
  {==}

def Laws.preserves_finished():
  {==}
```

```
All terms check.
```

`bend PROOF.bend` is the gate. It checks every law and every proof, and it prints `All terms check.` only when none is open. Run `bend LAWS.bend` alone and every law is an open claim. The message counts them as TODOs:

```
Error: 4 TODOs found.
```

Two rules follow from the layout:

- **An accepted law is never weakened to pass.** Changing a law so that its proof becomes easy defeats its purpose. If a proof is hard, the law may be correct and the proof is what needs work.
- **A law with no def is an open claim.** It stays open until a same-named def proves it. A file that states a law and leaves it unproved fails the gate.

Chapter 11 covers the gate's other failure mode: a proof that depends on an `@unsafe` definition. Version 2.0.25 prints a warning and still exits with status 0, so do not treat exit status alone as the gate. Bend 2.0.36 adds `bend PROOF.bend --verdict`, which rechecks the proofs with a separate kernel. The bend2.dev notes say that option needs Lean v4.34.0 exactly, or a prebuilt kernel named by the `BENDTT` variable. Version 2.0.25 has no such option, and no Lean toolchain was installed on the lab machine, so the gate in this tutorial is `bend PROOF.bend` plus a reading of its output.

EXERCISES

1. Prove a right identity. Prove `Nat.add(x, 0n) == x` without copying `add_zero`, by naming the induction hypothesis and rewriting with it.

Solution. This is the proof above. Put the hypothesis `add_zero(p)` in a rewrite whose motive is `1n+_`, then close with `{==}`.

2. Find the failing proof. The law `cancels_running` states that cancelling a running job yields `Cancelled{}`. Change the `Running{}` branch of `cancel` to return `Queued{}` and predict what the gate prints.

Solution. The gate fails, because `cancels_running` no longer holds for the new `cancel`. The law is the specification, and the implementation has to meet it.

NEXT

Chapter 08 covers types that depend on values, and why they make preconditions checkable.

C.16 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/02-dependent-types-and-totality.md

--- title: "Bend 2 tutorial 08: dependent types and totality" description: "Types computed from values, Unit and Empty as checked preconditions, length-indexed lists, the termination rule in practice, and why @unsafe changes what a proof may rely on." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/08-dependent-types-and-totality/ sources:

- https://bend2.dev/learn/dependent-types/ (https://bend2.dev/learn/dependent-types/)
- https://bend2.dev/learn/sized-lists/ (https://bend2.dev/learn/sized-lists/)
- https://bend2.dev/learn/totality/ (https://bend2.dev/learn/totality/)
- https://bend2.dev/errors/unsafe-dependents/ (https://bend2.dev/errors/unsafe-dependents/)
- https://bend2.dev/notes/what-is-bend2/ (https://bend2.dev/notes/what-is-bend2/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

08: dependent types and totality

A **dependent type** is a type that a function computes from a value. In Haskell you reach for GADTs or type families. In Bend every type is an ordinary function call, so the same machinery that checks values checks types. This chapter shows three uses: a type that is `Unit` for one input and `Empty` for another, a list whose length is part of its type, and the rule that keeps every such definition terminating.

Three kinds of type appear in this chapter:

Kind	Example	What it depends on
normal	`U32`	nothing
function	`U32 -> U32`	nothing: it is the type of functions from values to values
dependent	`IsZero(n)`, `Vec(A, n)`	a value `n`, and for `Vec` also a type `A`

A dependent type is a function that returns a type. `IsZero` takes a `Nat` and returns `Unit` or `Empty`, and that result limits what a caller may pass: for `0n` there is one value to pass, and for `1n+p` there is none. The two sections below build those dependent types.

A TYPE THAT COMPUTES

`IsZero(n)` is `Unit` when `n` is zero and `Empty` otherwise. `Empty` has no values, so a function that demands an `IsZero` argument can only be called with zero:

```
import Base

def IsZero(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      Empty

def require_zero(n: Nat, evidence: IsZero(n)) -> Nat:
  n

def main() -> Nat:
  require_zero(0n, Unit{})
```

```
0n
```

`Unit{}` is the only value of `Unit`, so the caller must supply it. Calling `require_zero(1n, Unit{})` is rejected, because `IsZero(1n)` is `Empty` and `Unit{}` is not an `Empty`:

```
import Base

def IsZero(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      Empty

def require_zero(n: Nat, evidence: IsZero(n)) -> Nat:
  n

def main() -> Nat:
  require_zero(1n, Unit{})
```

```
- expected : Empty
```

A function whose argument has type `Empty` can be written with an empty `match`. It needs no case, because no value can exist:

```
import Base

def from_empty(-A: Type, e: Empty) -> A:
  match e:

def main() -> Nat:
  0n
```

```
0n
```

LENGTH IN THE TYPE

A `Vec` is a nested pair, and its type depends on a `Nat` length. `first` takes a list of length `1n+n`, which is one or more, so it has no empty case:

```
import Base

def Vec(-A: Type, n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      A & Vec(A, p)

def first(-A: Type, -n: Nat, xs: Vec(A, 1n+n)) -> A:
  (x, rest) = xs
  x

def total(n: Nat, xs: Vec(U32, n)) -> U32:
  match n:
    case 0n:
      0
    case 1n+p:
      (x, rest) = xs
      (x + total(p, rest) : U32)

def main() -> U32 & U32:
  x = first(U32, 2n, (10, (20, (30, Unit{}))))
  s = total(3n, (10, (20, (30, Unit{}))))
  (x, s)
```

```
(10, 60)
```

Two things make this work. The length `n` determines how many pairs `total` consumes, and its recursive call gets the smaller length `p`. The parameter `n` of `first` is erased: it appears only in the type, so it does not exist at run time (the `-` marks it).

Haskell would enforce this with a length-indexed GADT and a `Nat`-kinded phantom type. Bend uses the same idea, but the index is an ordinary function that the checker evaluates.

A PROOF THAT USES A TYPE

A law can take evidence as an argument. `IsEven(x)` below is `Unit` for zero, `Empty` for one, and `IsEven(p)` for `2n+p`. The law says halving an even number and doubling the result gives the number back. The evidence `e` is what makes the law true for only even inputs:

```
import Base

def IsEven(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+0n:
      Empty
    case 2n+p:
      IsEven(p)

def half(n: Nat) -> Nat:
  match n:
    case 0n:
      0n
    case 1n+0n:
      0n
    case 2n+p:
      1n+half(p)

law half_ok:
  for x: Nat
  for e: IsEven(x)
  {Nat.double(half(x)) == x : Nat}

def half_ok(x, e):
  match x:
    case 0n:
      {==}
    case 1n+0n:
      match e:
    case 2n+p:
      %half_ok(p, e) : {2n+Nat.double(half(p)) == 2n+_ : Nat}
      {==}

def main() -> Nat:
  half(8n)
```

```
4n
```

The branch `1n+0n` has no proof body of its own. Its evidence `e` has type `Empty`, so `match e:` with no cases closes the branch. The indentation of `case 2n+p:` is the one the checker accepts after an empty `match`, and it is the same as the bend2.dev example.

TOTALITY: EVERY RECURSION SHRINKS

Bend requires every safe recursive call to pass a structurally smaller input, as chapter 02 explains. The rule holds for proofs as well as for programs, because a proof is also a recursive definition. Two consequences matter in practice.

1. A recursion that does not shrink is rejected, so a loop bounded by the outside world takes a `Nat` fuel argument and matches its predecessor.
2. Mutual recursion is not allowed in a safe definition. Merge the functions into one with a selector argument.

WHEN @UNSAFE CHANGES THE RULES

`@unsafe` disables the termination check for one definition. The compiler still runs the function, but any proof that depends on it is outside the guarantee. This program runs and prints 3, with a warning that the proof check now depends on unsafe code:

```
import Base

@unsafe def idu(n: Nat) -> Nat:
  match n:
    case 0n:
      0n
    case 1n+p:
      1n+idu(p)

def main() -> Nat:
  idu(3n)
```

```
3n
All terms check, but 2 defs rely on unsafe or foreign code:
```

The same definition used by a law is the case that matters. Version 2.0.25 reports it and exits with status 0, so the gate must read the message, not just the status:

```
import Base

@unsafe def idu(n: Nat) -> Nat:
  match n:
    case 0n:
      0n
    case 1n+p:
      1n+idu(p)

law idu_two:
  {idu(2n) == 2n : Nat}

def idu_two():
  {==}
```

```
All terms check, but 2 defs rely on unsafe or foreign code:
```

Be careful with unsafe loops too. A law that requires the checker to evaluate a non-terminating `@unsafe` function does not finish. In one run on this machine, a law of that shape ran for more than 20 seconds before the run was stopped. Keep `@unsafe` out of every definition that a law depends on, and keep laws about terminating functions.

EXERCISES

1. Odd numbers. Write `IsOdd(n)` that is `Empty` for zero, `Unit` for one, and `IsOdd(p)` for `2n+p`. It is the complement of `IsEven`. Check that a function `pred_odd(n, e)` needs `IsOdd(n)`: prove `IsOdd(1n)` by building `Unit{}`, and check that `IsOdd(0n)` is empty.

Solution. `IsOdd` mirrors `IsEven`. The function `pred_odd` takes evidence that its argument is odd. `IsOdd(0n)` is `Empty`, so the zero case is closed by an empty `match`. Odd numbers such as `5n` pass the evidence `Unit{}`:

```
import Base

def IsOdd(n: Nat) -> Type:
  match n:
    case 0n:
      Empty
    case 1n+0n:
      Unit
    case 2n+p:
      IsOdd(p)

def pred_odd(n: Nat, e: IsOdd(n)) -> Nat:
  match n:
    case 0n:
      match e:
    case 1n+p:
      p

def main() -> Nat:
  pred_odd(5n, Unit{})
```

```
4n
```

To check that `IsOdd(0n)` is empty, match on evidence of that type. A `match` with no cases closes the branch:

```
import Base

def IsOdd(n: Nat) -> Type:
  match n:
    case 0n:
      Empty
    case 1n+0n:
      Unit
    case 2n+p:
      IsOdd(p)

def odd_zero_empty(e: IsOdd(0n)) -> Nat:
  match e:

def main() -> Nat:
  1n
```

```
1n
```

2. Fuel. Write `repeat(fuel, x)` that returns `x` plus `fuel`, using the fuel as the loop bound: each step adds 1 to `x`. Explain why it terminates.

Solution. It terminates because the first argument `fuel` is matched and the recursive call gets its predecessor. The following program adds 5 to 10:

```
import Base

def repeat(fuel: Nat, x: U32) -> U32:
  match fuel:
    case 0n:
      x
    case 1n+p:
      repeat(p, (x + 1 : U32))

def main() -> U32:
  repeat(5n, 10)
```

```
15
```

NEXT

Chapter 09 covers the IO monad, files, command-line arguments, and the compilation targets.

C.17 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/03-proofs-concepts.md

--- title: Proofs description: "A theorem is an assert block; its proof is an ordinary def of the same name." updated: 2026-08-21 verified: 2026-09-05 status: docs-only canonical: /learn/proofs/ sources:

- https://bend2.dev/learn/proofs/ `(https://bend2.dev/learn/proofs/)` ---

Proofs

This page proves one fact end to end: addition commutes. If the previous page was the theory, this is the lab session — one theorem, one proof, every step shown. A Bend2 theorem is an `assert` block, and its proof is an ordinary `def` carrying the same name: the claim, then the evidence. Here the claim is that addition commutes, and the argument is the standard Peano one, induction on the first argument.

```
import Base

# Theorem: "for all numbers a and b, a + b equals b + a".
assert add_comm:
  for all a : Nat
  for all b : Nat
  {add(a, b) = add(b, a) : Nat}

# Proof: case analysis, induction, rewrites (`%`).
def add_comm(a, b):
  match a:
    case Z{}:
      %add_zero(b)
      {=}
    case S{p}:
      %add_succ(b, p)
      %add_comm(p, b)
      {=}
```

THE STATEMENT

Propositions are types. `for all a : Nat` (two words) quantifies over the naturals; `{add(a, b) = add(b, a) : Nat}` is an equality between `Nat` expressions, braces around the proposition. The `assert` names the obligation; the `def` of the same name discharges it, parameter types supplied by the assert.

THE PROOF

Case analysis plus rewriting. `match a:` splits on `Nat`'s constructors — `Z{}` zero, `S{p}` successor, brace-suffixed patterns. Each `%` line rewrites the goal with a named equation: `%add_zero(b)` and `%add_succ(b, p)` apply lemmas from `Base`; `%add_comm(p, b)`, the recursive call, **is** the induction hypothesis. `{=}` closes a branch once rewriting made both sides identical.

FOLLOWING THE GOALS

Track each branch's goal and the `%` lines read plainly. Zero case: goal `add(Z, b) = add(b, Z)`; left reduces by definition, `%add_zero(b)` rewrites `add(b, Z)` to `b`, `{=}` closes `b = b`. Successor case: goal `add(S(p), b) = add(b, S(p))`; left reduces to `S(add(p, b))`, `%add_succ(b, p)` turns the right into `S(add(b, p))`, the induction hypothesis equates them for `{=}`.

NEXT TO LEAN

Term-level proofs, Agda register: recursion as induction, no tactic layer (`simp`, `omega`) in sight. Whether that trade pays off is the comparison.

C.18 · bend-gme/version-1/.opencode/skills/bend2-proofs/references/04-types-and-proofs-concepts.md

--- title: "Types and proofs, step by step" description: "Annotations, polymorphism, first dependent types, sized lists, equality, induction, and totality." updated: 2026-09-05 verified: 2026-09-05 status: docs-only canonical: /learn/types-proofs/ sources:

- https://bend2.dev/learn/ `(https://bend2.dev/learn/)` ---

Types and proofs, step by step

From annotations to proofs: how Bend2 types carry values, state equalities, and discharge obligations by recursion. The full worked proof is on the proofs page.

WHAT IS A TYPE-BASED PROVER?

Every compiler checks your code a little. A type-based prover checks it all the way down — not just "is this a number?" but "is this claim true for every possible input?"

Three analogies, from familiar to exact:

A blueprint inspector. A function signature is a promise about shape: takes two numbers, returns a number. A normal type checker verifies the shape. A prover verifies the *behavior written into the shape*: takes any two numbers, returns their sum regardless of order. The checker is an inspector who refuses to sign off until the building matches the blueprint — before anyone moves in, i.e. before the program ever runs.

A fact-checker, not a spell-checker. Testing runs examples: it tries a million pairs and finds no counterexample. Proving covers the cases you didn't try — including the one your test suite missed. A test says "it worked every time we looked"; a proof says "there is nowhere for a bug to hide."

Claims are types; evidence is code. This is the whole trick, called *propositions as types*. The type `{add(a, b) = add(b, a) : Nat}` is the claim "addition commutes." A program with exactly that type *is* the evidence — it can only exist if the claim holds, the way a key can only exist if it fits its lock. Writing the proof means constructing that program; checking it means the compiler trying the key. In Bend2 you state the claim with `assert` and deliver the evidence with an ordinary `def` of the same name. No separate proof language, no tactic scripts: the proof is a program, and the checker simply runs its type discipline over it.

TYPE ANNOTATIONS

Every posted `def` annotates parameters and results (`a: U32` , `-> U32` , `-> IO<Unit>`).

POLYMORPHISM

Polymorphism — functions generic over types *(planned)*.

DEPENDENT TYPES

Types can mention values: the proofs sample quantifies `for all a : Nat` and states equality `{add(a, b) = add(b, a) : Nat}` as a type. Sized lists ("a list of exactly this length") and equality types are the same idea pushed further *(planned)*.

SIZED LISTS

Sized lists *(planned)*.

EQUALITY

Equality appears in the public theorem sample as `{add(a, b) = add(b, a) : Nat}` .

INDUCTION

Induction in Bend2 is recursion: the `add_comm` proof calls itself as the induction hypothesis, and each `%` rewrite is checked.

TOTALITY

The guarantee that functions terminate, which is what makes proofs sound, is enforced by the Live linear mode *(planned)*.

C.19 · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/05-proof-lineage.md`

--- title: "Where the proofs come from" description: "Formality's productivity thesis, Kind's minimal theory, and why Bend2 treats proofs as a developer tool." updated: 2026-09-05 verified: 2026-09-05 status: stable canonical: /notes/proof-lineage/ sources:

- https://github.com/VictorTaelin/Formality `(https://github.com/VictorTaelin/Formality)`
- https://github.com/HigherOrderCO/Kind `(https://github.com/HigherOrderCO/Kind)`
- https://github.com/VictorTaelin/articles `(https://github.com/VictorTaelin/articles)`
- https://bend2.dev/notes/bend2-vs-lean/ `(https://bend2.dev/notes/bend2-vs-lean/)` ---

Where the proofs come from

Prerequisites

- What is Bend2? — the language this lineage leads to

Bend2's dependent types did not appear from nowhere. They are the third generation of one author's argument: **proofs are a developer-productivity tool**, an evolution of types the way TypeScript was an evolution of JavaScript. Understanding the lineage makes Bend2's strangest choices (linear everything, Dead vs Live, synthesis-first) read as conclusions rather than quirks.

PROOF-ORIENTED PROGRAMMING

Proofs are not only a way to certify mathematics. In HOC's tradition they are a programming tool: precise specifications make generated programs checkable, so synthesis can propose and the kernel can reject incorrect answers.

FORMALITY: PROOFS AS SUPERPOWERS

Formality (Taelin, ~2018, "a modern programming language featuring formal proofs — now written in itself") opens with the thesis outright: forget mathematics and security, the boring stuff — adding types to untyped languages already demonstrated the productivity jump, and formal proofs are the next step of that same curve. The language is there to make the developer faster, not the mathematician happier. Every Bend2 proof decision descends from this sentence: in the Lean comparison, the program is the product.

Taelin's essay on proving vs testing makes the mechanism explicit: auditing doesn't scale, tests sample behavior, proofs cover it — and the cheapest proof is one the machine helps write. That is the exact slot SupGen occupies in Bend2: synthesis proposes, the checker disposes.

KIND: THE MINIMAL THEORY

Kind ("a modern proof language") is the visible ancestor of Bend2's type system — and it is deliberately tiny. Its core grammar fits on one screen: dependent function `∀(x:T)U`, lambda, application, self-types, datatypes with constructors, pattern matching, holes (`?name`) for unfinished proofs, and `*` as the type of types. No tactic language, no standard library ceremony: just enough theory to state and check precise types.

Kind2's conversion checker in three lines To decide *A == B*: if textually identical, true; else reduce both to weak normal form and check similarity field-by-field. No "seen equations" map (Kind1's slower fix), no constructor flags — just identical → reduce → similar, in exactly that order. It covers everything Coq's checker covers, and the order-sensitivity is load-bearing: swap it and self-referential types (like *List Char* where *Char* aliases *#U60*) loop forever.

That design moral — the obvious algorithm works if reductions happen in precisely the right order — runs through all of HOC's work, from evaluators to checkers. Bend2 inherits the shape (quantifiers as types, programs as proofs) and adds the linearity discipline plus the Dead/Live mode split that lets it keep `Type : Type` without letting paradoxes near live code.

WHAT THIS PREDICTS ABOUT BEND2

Three predictions, all checkable at release: proofs will stay term-level (recursion as induction, rewrites as `%`, no tactic layer — tactics serve mathematicians, and Bend2 serves developers); the standard library will grow specifications first (precise types are SupGen's search targets); and the killer feature will read as productivity, not certainty — proofs replacing test suites rather than chasing foundations. If release-day Bend2 instead ships a tactic engine, this page is wrong in an interesting way, and will say so.

C.20 · `bend-gme/version-1/.opencode/skills/bend2-proofs/references/06-bend-vs-lean.md`

--- title: Bend2 vs Lean description: "Same dependent types, different jobs: in Lean the proof is the product, in Bend2 the program is." updated: 2026-08-30 verified: 2026-09-05 status: prerelease canonical: /notes/vs-lean/ sources:

- https://bend2.dev/notes/bend2-vs-lean/ `(https://bend2.dev/notes/bend2-vs-lean/)`
- https://github.com/HigherOrderCO/Kind `(https://github.com/HigherOrderCO/Kind)` ---

Bend2 vs Lean

Prerequisites

- Proofs — the `add_comm` sample both sides prove
- Where the proofs come from — Formality, Kind, and the productivity thesis

Both live on the propositions-as-types side of Curry–Howard: a specification is a type, a proof is a program, checking is type checking. What differs is which side pays the bills.

LEAN: THE PROOF IS THE PRODUCT

Lean 4 (de Moura, Lean FRO, Apache 2, self-hosted through C) exists to formalize: mathematics, verified algorithms, specifications. Its center of mass is mathlib — a half-million-item library built by hundreds of contributors, so new proofs import lemmas instead of rebuilding them. Automation means tactics: decision procedures, simp sets, `omega`, a decade of accumulated machinery. Execution is efficient sequential C with task-level concurrency; nobody chooses Lean because their proof needs a GPU.

BEND2: THE PROGRAM IS THE PRODUCT

Bend2's types exist to make machine-generated code checkable: synthesis (SupGen) proposes, the checker disposes. Its ancestor is Kind, Higher Order Company's proof language. The type theory is linear with no hidden exceptions: a plain binder allows at most one live use, duplication is written `+x`, only `Data` kinds may be duplicated. The checker splits in two — a Dead mode where specifications live unrestricted and may diverge, and a Live mode where code and proofs are linear and terminating — which keeps paradox-shaped terms on the Dead side despite `Type : Type` (what Girard's paradox makes inconsistent as a logic; Lean buys the same safety with a universe hierarchy plus termination checker).

The one public proof (addition commutes, August 2026) is term-level, Agda register: the function's own recursion is the induction, lemmas apply as `%` rewrites, `{=}` closes goals. No `simp`, no `omega`, no visible tactic layer at all.

THE SAME THEOREM IN LEAN 4

For comparison, here is how a Lean user proves the same fact today — tactics orchestrating library lemmas, in about four lines:

```
-- Illustrative: shows the Lean style, not typechecked for this page.
theorem add_comm' (a b : Nat) : a + b = b + a := by
  induction a with
  | zero => rw [Nat.zero_add, Nat.add_zero]
  | succ n ih => rw [Nat.succ_add, Nat.add_succ, ih]
```

Read the two side by side. Lean's version says *how to search*: `induction` picks the principle, `rw` fires named equations from the library, and the library (`Nat.zero_add`, `Nat.add_succ`, hundreds of thousands of siblings in mathlib) does the heavy lifting. Bend2's version *is* the search result already: the recursion is the induction, each `%` line is one rewrite step, `{=}` checks the endpoints match. No search language exists because, in Bend2's story, the machine (SupGen) performs the search and the human reads the trace. Which side of that trade pays off — tactic scripts over a giant library, or machine-found term proofs over precise types — is the open bet.

Watch it run in the infoview, goal by goal — input and output in one transcript, OCaml-toplevel style:

```
⊢ a + b = b + a
-- induction a with
-- ┌ case zero, ⊢ 0 + b = b + 0
-- │   rw [Nat.zero_add] → ⊢ b = b + 0
-- │   rw [Nat.add_zero] → no goals
-- └ case succ, ⊢ (n + 1) + b = b + (n + 1)
-- rw [Nat.succ_add] → ⊢ (n + b) + 1 = b + (n + 1)
-- rw [Nat.add_succ] → ⊢ (n + b) + 1 = (b + n) + 1
-- rw [ih] → no goals
```

A couple of remarks:

- Every rewrite names a lemma that must already exist — in mathlib, or proved just above. Bend2's `%` lines work the same way over `Base`.
- The `ih` in the last step *is* the induction hypothesis, exactly where Bend2 puts the recursive call `%add_comm(p, b)`. Same proof, two costumes: search script vs. found term.

THE DEFAULT TODAY

Lean, for anything needing a proof now — verified algorithms, formalized specs, mathematics. The library asymmetry is not close: 500,000+ items versus no public Bend2 library. Bend2 joins proofs to programs that run on HVM4 and to synthesis from precise types — a combination nobody else is selling, and nothing you can download.

C.21 · bend-gme/version-1/.opencode/skills/bend2-ownership/SKILL.md

Bend 2: ownership and copying

A binder is a resource. Use it once, or say in its type that it may be copied.

MODEL

- **Affine by default**: each binder is consumed at most once.
- `+x` marks a parameter whose value may be used many times (copyable kind). `-x` marks an **erased** parameter: it exists for the checker (types, proofs) and is gone at runtime.
- `Data` types (lists, records, ADTs) are values that can be copied when their kind allows. `Type` types (e.g. `Array<T>`) have exactly one owner and are updated in place; reads hand the array back.
- Closures capture their free variables; capturing an affine value consumes it.
- A copy is a **real copy**. Lazy, incremental sharing (a DUP node splitting a value on demand) exists only on an interaction-calculus runtime (HVM5), which is not available today; `references/02-sharing-concepts.md` describes that HVM-style model, so read it as background, not as how BendRT behaves.
- A copy is not free: its cost is the size of what is copied (`references/01-ownership-and-copying.md` §"What a copy costs").

RULES

1. Read `observed : x (consumed more than once)` literally: find the second use; either mark `+x` (if copying is cheap and intended) or restructure so the value flows once.
2. Thread single-owner values (`Array`, accumulators) through tail-recursive loops; return them with the result (`Array<T> & T`).
3. Seal big inputs into `Data` once (lists/records) when many functions must read them; keep `Array` for in-place builders.
4. Every index into an `Array` is bounds-checked against an explicit length `n`: arrays have power-of-two size and **wrap** out-of-range indexes silently (`../bend2-effects-targets/references/03-js-target-facts.md`).
5. Do not capture large affine values in closures you call more than once.
6. Use `-` for type-level/proof-only arguments so they cost nothing at runtime.
7. Budget every `+x` on large data as a full copy; never assume sharing makes it cheap (ADR-031).

PATTERNS (CHECKED WITH BEND 2.0.25)

```
import Base

def twice(+n: Nat) -> Nat:
  Nat.add(n, n)

def main() -> Nat:
  twice(3n)
```

Output `6n`. Without the `+` the checker rejects it:

```
Error:
- expected : n
- observed : n (consumed more than once)
Location: twice
```

PITFALLS

Symptom	Cause	Fix
`consumed more than once`	second use of an affine binder	`+x`, or restructure the data flow
silent wrong value from an array read	index past `n` wrapped around	bounds-check every index helper
quadratic time building a list	repeated append on the right	accumulate reversed, then `List.reverse` once
a closure "uses up" its capture	affine capture	pass the value as an argument instead

VERIFY

`BEND_NO_TELEMETRY=1 bend file.bend --check-only` (ownership is checked statically; there is no runtime borrow check).

REFERENCES

File	Covers	Open when
`references/01-ownership-and-copying.md`	affine binders, `Data` vs `Type`, `-` erased params, kinds, arrays, maps, templates, closures, folds, copy cost	any ownership question
`references/02-sharing-concepts.md`	sharing vs copying, concept page	explaining why a copy happens
`../bend2-superposition/references/01-duplication-superposition.md`	`dup` at the calculus level	relating copies to DUP nodes

RELATED

`bend2-language` · `bend2-effects-targets` (arrays across the JS FFI) · `bend2-superposition` (dup/sup) · `bend2-app` (data layout in a big core).

C.22 · bend-gme/version-1/.opencode/skills/bend2-ownership/references/01-ownership-and-copying.md

--- title: "Bend 2 tutorial 03: ownership and copying" description: "Affine binders, + and - quantities, Data versus Type, kinds, one-owner arrays, maps, templates, closures, and folds, with what each kind of copy costs." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/03-ownership-and-copying/ sources:

- https://bend2.dev/learn/duplication/ (https://bend2.dev/learn/duplication/)
- https://bend2.dev/learn/kinds/ (https://bend2.dev/learn/kinds/)
- https://bend2.dev/learn/arrays/ (https://bend2.dev/learn/arrays/)
- https://bend2.dev/learn/maps/ (https://bend2.dev/learn/maps/)
- https://bend2.dev/learn/templates/ (https://bend2.dev/learn/templates/)
- https://bend2.dev/learn/closures/ (https://bend2.dev/learn/closures/)
- https://bend2.dev/learn/folds/ (https://bend2.dev/learn/folds/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

03: ownership and copying

Haskell shares values by default, and a garbage collector frees them later. Bend does the opposite: every binder says how many times it may be used, and the checker enforces it. This chapter explains those quantities, the cost of each way to copy, and the containers that rely on them.

AFFINE BY DEFAULT

A plain binder may be used at most once. Dropping it is free:

```
import Base

def twice(n: Nat) -> Nat:
  Nat.add(n, n)

def main() -> Nat:
  twice(3n)
```

```
- expected : n
- observed : n (consumed more than once)
```

The fix is `+`, which marks the binder as reusable:

```
import Base

def twice(+n: Nat) -> Nat:
  Nat.add(n, n)

def main() -> Nat:
  twice(3n)
```

```
6n
```

`+` is allowed only when the binder's type is `Data`. `Nat`, `U32`, `Bool`, and records declared `is Data` qualify. Functions, arrays, and file handles never do.

DATA VERSUS TYPE

A type declared `is Data` may be copied with `+`. A type declared `is Type` may not:

```
import Base

type Token is Type:
  Tok{n: Nat}

def show(t: Token) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  t : Token = Tok{3n}
  +u = t
  show(u)
```

```
- expected : Data
- observed : Type
```

The same program works when `Token` is declared `is Data`, and it uses an annotation to build the value:

```
import Base

type Tok is Data:
  Tok{n: Nat}

def show(t: Tok) -> Nat:
  match t:
    case Tok{n}:
      n

def main() -> Nat:
  +t = {Tok{3n} : Tok}
  Nat.add(show(t), show(t))
```

```
6n
```

Think of a kind as a reuse budget. `Type` allows one use, and `Data` allows as many copies as the code asks for, marked with `+`. A type with arguments takes its kind from them, so `Maybe<&2, Nat>` is `Data` and `Maybe<&1, Nat>` is `Type`:

```
import Base

def dup(+m: Maybe<&2, Nat>) -> Maybe<&2, Nat> & Maybe<&2, Nat>:
  (m, m)

def main() -> Maybe<&2, Nat> & Maybe<&2, Nat>:
  dup(Some{4n})
```

```
(Some{4n}, Some{4n})
```

With `&1`, the same `+m` is rejected:

```
import Base

def dup(+m: Maybe<&1, Nat>) -> Maybe<&1, Nat> & Maybe<&1, Nat>:
  (m, m)

def main() -> Maybe<&1, Nat> & Maybe<&1, Nat>:
  dup(Some{4n})
```

```
- expected : Data
- observed : Type
```

THE MINUS SIGN: ERASED PARAMETERS

`-A` is a parameter that exists only at compile time. It appears in types and is deleted before the program runs. `keep_first` below ignores its second argument, which is dropped:

```
import Base

def twice(+n: Nat) -> Nat:
  Nat.add(n, n)

def keep_first(-A: Type, x: A, y: A) -> A:
  x

def main() -> Nat:
  keep_first(Nat, twice(3n), 9n)
```

```
6n
```

WHY MARK A PARAMETER ERASED

Without the minus sign, `A` is an ordinary parameter. The checker accepts both versions, but the compiled programs differ: the plain version receives the type as a value and drops it at the end. Write the program twice, once with `-A` and once with `A`, and compare the C each one emits (chapter 05 shows `-o file.c`):

```
bend erased.bend -o erased.c
bend plain.bend -o plain.c
grep -c _A_0 erased.c plain.c
```

```
erased.c:0
plain.c:2
```

The erased version has no variable for `A`. The plain version has one, `_A_0`, which it receives and then drops. The minus sign removes that cost.

WHAT A RUN-TIME USE IS

A binder is used at run time when the compiled body needs its value: it returns the value or computes with it. An erased binder cannot be used that way, because its only uses are in types and proofs:

```
import Base

def bad(-A: Type, x: A) -> A:
  A

def main() -> Nat:
  bad(Nat, 6n)
```

```
- expected : A
- observed : Type
```

TYPE ARGUMENTS ARE EXPLICIT

Bend infers almost nothing, so a type argument is passed like any other argument. Leaving it out is rejected:

```
import Base

def identity(-A: Type, x: A) -> A:
  x

def main() -> U32:
  identity(42)
```

```
- expected : Type
- observed : U32
```

With the type passed, the call checks. The type is erased, so it costs nothing at run time:

```
import Base

def identity(-A: Type, x: A) -> A:
  x

def main() -> U32:
  identity(U32, 42)
```

```
42
```

KINDS: HOW OFTEN A VALUE MAY BE USED

Every type has a kind that caps its uses. `Type` is `Kind(&1)` (affine) and `Data` is `Kind(&2)` (reusable). Base declares `List<a, A>` so that a list is exactly as reusable as its elements. A function can take the kind as an argument:

```
import Base

def count(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{head, tail}:
      1n+count(a, A, tail)

def main() -> Nat:
  count(&2, U32, [10, 20, 30])
```

```
3n
```

A parameter of type `+List<U32>` is a list whose elements are reusable. The type is `List<&2, U32>`, so the `+` on the parameter lets the function use the whole list twice:

```
import Base

def twice_len(+xs: +List<U32>) -> Nat:
  Nat.add(List.length(&2, U32, xs), List.length(&2, U32, xs))

def main() -> Nat:
  twice_len([1, 2, 3])
```

```
6n
```

ARRAYS: ONE OWNER, UPDATED IN PLACE

An array has exactly one owner. Writing to it changes the storage in place, and the array comes back to you. `[0 : U32*8n]` creates eight slots (the count must be a power of two). A read returns the array together with the element:

```
import Base

def value(result: Array<U32> & U32) -> U32:
  match result:
    case (array, item):
      item

def main() -> U32:
  a = [0 : U32*8n]
  a[5] <- 42
  value(a[5])
```

```
42
```

The bracket form is an array literal, and its type part says what the slots hold. `[fill : Type*slots]` asks for `slots` slots, which must be a power of two. `[fill : Type^depth]` names a depth, so `U32^3n` is 2^3 = 8 slots. The function form is `Array.new(-T: Data, +d: Nat, +v: T) -> Array<T>` in Base. `Array.new(U32, 3n, 0)` builds the same array as `[0 : U32^3n]`, because its second argument is the depth. Indexes wrap, so on an eight-slot array `a[9]` is slot 1. A slot count that is not a power of two is rejected:

```
import Base

def main() -> U32:
  a = [0 : U32*6n]
  a[5] <- 42
  0
```

```
- message  : a power of two count (^d takes a depth)
```

An array passed to a function is updated there and returned. Put the write in a def that takes the array as a parameter:

```
import Base

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item

def set_first(a: Array<U32>) -> Array<U32>:
  a[0] <- 7
  a

def main() -> U32:
  a = [1 : U32*4n]
  b = set_first(a)
  first_of(b[0])
```

```
7
```

A second independent array needs `Array.clone`, which copies every slot. Its result is a pair, and the pair cannot be destructured in `main`: a `match` may not inspect a computed value. Pass it to a helper that matches its parameter:

```
import Base

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item

def set_first(a: Array<U32>) -> Array<U32>:
  a[0] <- 7
  a

def use_clone(p: Array<U32> & Array<U32>) -> U32:
  match p:
    case (a1, a2):
      b = set_first(a1)
      x = first_of(b[0])
      y = first_of(a2[0])
      (x + y : U32)

def main() -> U32:
  a = [1 : U32*4n]
  use_clone(Array.clone(U32, a))
```

```
8
```

Here `x` is 7 and `y` is 1, so the copies stayed independent. Destructuring the result in `main` is rejected:

```
import Base

def main() -> U32:
  a = [1 : U32*4n]
  (a1, a2) = Array.clone(U32, a)
  first_of(a1[0])

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item
```

```
- message  : a parameter or field scrutinee (a match cannot scrutinize a computed value: give it its own def)
```

MAPS

A `Map` associates string keys with values. `Map.new` takes the kind and the value type. `Map.set` and `Map.get` return the map with their result, so the map keeps its owner. `Map.get` takes its default before the map:

```
import Base

def value(result: Map<&2, U32> & U32) -> U32:
  match result:
    case (map, item):
      item

def main() -> U32:
  m = Map.new(&2, U32)
  m = Map.set(&2, U32, m, "answer", 42)
  value(Map.get(U32, 0, m, "answer"))
```

```
42
```

A `Set` is a map with no values. `Set.add` returns the set, and `Set.size` counts its keys:

```
import Base

def main() -> Nat:
  s = Set.new()
  s = Set.add(s, "x")
  s = Set.add(s, "y")
  Set.size(s)
```

```
2n
```

TEMPLATES: COMPILE-TIME ARGUMENTS

A template parameter starts with `~`. Its argument is substituted at compile time and is not passed at run time. Each distinct template argument produces its own compiled copy, so a template may be called as often as you like:

```
import Base

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

```
42
```

A template may call itself and pass its template argument along. Two different steps give two compiled copies of the same loop:

```
import Base

def run(~step: U32 -> U32, n: Nat, x: U32) -> U32:
  match n:
    case 0n:
      x
    case 1n+p:
      run(~step, p, step(x))

def main() -> U32 & U32:
  (run(~(x => (x + 1 : U32)), 3n, 0), run(~(x => (x + 2 : U32)), 3n, 0))
```

```
(3, 6)
```

A template argument must be closed: it may mention top-level definitions, but not a local variable of the caller. Here `k` is a local variable, and the checker rejects the call:

```
import Base

def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def add_k(k: U32, x: U32) -> U32:
  twice(~(y => (y + k : U32)), x)

def main() -> U32:
  add_k(1, 40)
```

```
- expected : a template applied to closed ~ arguments (k is a variable here, not comptime: pass it at run time)
- observed : twice
```

WHY A TEMPLATE, NOT A CLOSURE

A parameter of type `U32 -> U32` is a closure, and a closure is affine. `f(f(x))` uses `f` twice, so the checker rejects this program:

```
import Base

def apply_twice(f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  apply_twice(x => (x + 1 : U32), 40)
```

```
- expected : f
- observed : f (consumed more than once)
```

A `~f` argument is substituted at compile time, so `f` is code rather than a value that can be used up. The `twice` above is that fix.

CLOSURES

A function may return a closure that captures a value. A closure is affine: it may be called once, even when everything it captures is `Data`:

```
import Base

def offset(k: U32) -> U32 -> U32:
  x => (x + k : U32)

def main() -> U32:
  add2 = offset(2)
  add5 = offset(5)
  add5(add2(35))
```

```
42
```

A top-level definition may be called many times. Use a template when you need a function called repeatedly.

FOLDS

`List.foldl` reduces a list. Its signature takes the quantity of the elements, their kind, the accumulator type, and the combining function as template arguments, then the list and the starting value. Here `add` is a top-level definition, so it costs nothing to pass:

```
import Base

def add(a: U32, b: U32) -> U32:
  (a + b : U32)

def main() -> U32:
  List.foldl(~&1, ~U32, ~U32, ~add, [10, 20, 12], 0)
```

```
42
```

`List.filter` and `List.sort` take the element type and a function as template arguments, and they return `List<&2, A>`, the reusable form. This `filter` keeps the elements greater than 5:

```
import Base

def is_big(x: U32) -> Bool:
  U32.is_gt(x, 5)

def main() -> List<&2, U32>:
  List.filter(~U32, ~is_big, [1, 6, 3, 8])
```

```
[6, 8]
```

WHAT A COPY COSTS

Operation	What happens	Cost
`+x` on a `Data` value	both uses share one value; a reference count tracks the copies	a count update per shared use
`Array.clone(T, a)`	every slot is copied into a second array	proportional to the array's size
`Array.fork`	two handles share one array; reads and writes race through atomics	cheap, but marked `@unsafe`, so no law may depend on it
a template argument	a separate compiled copy for each distinct argument	code size, no run-time cost
a closure	affine: called once	no sharing to pay for

The guide says `+` pays a reference count. The emitted C keeps that count in 24 bits (`RFC_CNT`). Chapter 05 shows how to find the counts. Chapter 10 explains why none of this gives you copy-on-write forks of running code.

EXERCISES

1. Cube. The parameter appears three times in `x * x * x`. Add the qualifier that makes it reusable, then check that `cube(3)` prints `27`.

```
import Base

def cube(+x: U32) -> U32:
  (x * x * x : U32)

def main() -> U32:
  cube(3)
```

```
27
```

2. Keep the original. Clone the array, change the first slot of one copy to 7, and return the first slot of the other copy. The answer is `1`.

```
import Base

def first_of(r: Array<U32> & U32) -> U32:
  match r:
    case (arr, item):
      item

def set_first(a: Array<U32>) -> Array<U32>:
  a[0] <- 7
  a

def other_copy(p: Array<U32> & Array<U32>) -> U32:
  match p:
    case (a1, a2):
      b = set_first(a1)
      first_of(a2[0])

def main() -> U32:
  a = [1 : U32*4n]
  other_copy(Array.clone(U32, a))
```

```
1
```

NEXT

Chapter 04 is the machine model: interaction nets, the rules that rewrite them, and why the model is the reason copying is explicit in Bend.

C.23 · bend-gme/version-1/.opencode/skills/bend2-ownership/references/02-sharing-concepts.md

--- title: "Sharing, duplication, superpositions" description: "What the fork shares, what duplication costs, and how superpositions branch — in Bend2 syntax." updated: 2026-09-05 verified: 2026-09-05 status: docs-only canonical: /learn/sharing/ sources:

- https://bend2.dev/learn/ `(https://bend2.dev/learn/)` ---

Sharing, duplication, superpositions

Sharing one value between two uses is where Bend2's cost model lives: the fork shares nothing, duplication is explicit, and superpositions branch. Runtime claims link to runnable HVM4.

The one idea The fork shares nothing (disjoint subgraphs), so it is free. Sharing one value between two uses needs explicit duplication — and duplicated functions meet superpositions, which is where HVM's power lives.

SHARING AND DUPLICATION

```
def main() -> U32:
  x = 40 + 2      # one value...
  x + x           # ...two uses: this is the duplication point
```

Bend2's types are linear by default (at most one live use per binder; duplication written `+x`, only `Data` kinds duplicable). Where the checker demands it, shared values duplicate explicitly rather than by accident — the language-level face of the runtime primitive.

UNSCOPED LAMBDAS

Unscoped lambdas exist in the runtime today (`λ$x` binders — run them in HVM4 hands-on). Their Bend2 surface syntax *(planned)*.

SUPERPOSITIONS

Superpositions are deliberately introduced later, in HVM4 Runtime 9.4. Their Bend2 surface syntax *(planned)*.

EVALUATION ORDER

Evaluation needs no annotations — order is the scheduler's business, and confluence keeps every schedule correct.

MEASURING SPEEDUP

Measuring speedup is a release-day exercise: same program, CPU call vs `!` call, with the runtime's interaction counters as the stopwatch *(planned)*.

C.24 · `bend-gme/version-1/.opencode/skills/bend2-superposition/SKILL.md`

Superposition, collapse, SupGen

In the Interaction Calculus a value can be two values at once; in released Bend 2 it cannot. Know both, and do not confuse them.

MODEL

- `dup` copies lazily: a DUP node meeting a value splits it incrementally.
- `sup ( &L{a, b} )` is a value that is both `a` and `b` under label `L`.
- **Same label** DUP meets SUP → annihilate (each copy takes one side). **Different labels** → commute (both duplicate), which is where the cost of search lives.
- **Collapse** turns a superposed result into the list of its branches; erasure drops failing branches early. This is search by superposition (`references/02-superposition-and-supgen.md` , `references/05-search-by-superposition.md`).
- **Superposition and sharing only work on HVM5, and HVM5 does not exist yet as a usable runtime.** Only an IC runtime gives SUP and DUP-based sharing; released Bend 2 runs on BendRT, and the HVM5 kernel in `bend-runtime/hvm5` is a research prototype whose end-to-end tests fail on 2.0.25 (`hvm5-ic`). So none of this can be used in an app today.
- **Status (bend 2.0.25)**: `&{1, 2}` is a parse error; there is no `sup` syntax, no SupGen, no copy-on-write fork of a running program, no homoiconic self-modification (`references/01-duplication-superposition.md` , `references/03-branching-and-self-change.md`). `ic` examples need Victor Taelin's Interaction Calculus binary (`IC_BIN`), not `bend` .

RULES

1. Treat any plan that needs `sup` , lazy sharing, SupGen or forked running code as **designed**, not buildable: it needs HVM5, which does not exist yet. Reject it in reviews unless it is labeled a research spike (ADR-031).
2. For search in an app: enumerate candidates explicitly (generators over an AST), prune with cheap checks first, and memoize shared sub-results; parallelize across workers.
3. For "test the new version against the old": run both on the same inputs and compare (an explicit A/B), and share precomputed features between them.
4. For branching state: keep immutable versions and content-address them; fork = new version id, merge = explicit three-way merge (see `hvm5-ic` for the IC-native design).
5. If you need real superposition, use the IC binary or the HVM5 kernel (`hvm5-ic`) in a research spike, and report it as such.

PATTERNS

```
# IC rules (../hvm5-ic/references/05-v2-calculus.md §2; text, not run here)
!&L{p,q} = &L{a,b}   →  p ← a; q ← b                          # same label: annihilate
!&L{p,q} = &M{a,b}   →  p ← &M{a0,b0}; q ← &M{a1,b1}; dup a; dup b     # L ≠ M: commute
```

```
# app-level replacement for superposed search (bend-gme generator, spec/09)
candidates = gen_batch(seed, budget)            # explicit enumeration over StratLang ASTs
survivors  = filter cheap_lints candidates      # erase early
scores     = screen(survivors) in N workers     # shared feature cache = shared prefix
```

PITFALLS

Symptom	Cause	Fix
`&{…}` parse error	`sup` is not in released Bend 2	explicit enumeration, or the IC binary for research
"free forks" assumed in a design	conflates IC theory with BendRT	label as designed; measure the explicit copy
exponential blow-up in a search spike	different-label commutation	share sub-results, prune earlier

VERIFY

```
BEND_NO_TELEMETRY=1 bend file.bend --check-only    # confirm what the released parser accepts
```

REFERENCES

File	Covers	Open when
`references/01-duplication-superposition.md`	`dup`, `sup`, labels, search, what Bend 2 lacks (checked parse errors)	first contact with sup/dup
`references/02-superposition-and-supgen.md`	IC behind sup/dup, labels, collapse, search with erasure, public facts about SupGen, cheap forks	designing search
`references/03-branching-and-self-change.md`	what SupGen, COW forks and homoiconic code would need; what Bend 2 offers instead	fork/test/merge ideas
`references/04-synthesis-concepts.md`, `references/05-search-by-superposition.md`	concept page and research note	background

RELATED

`hvm5-ic` (an IC kernel with real DUP/SUP, fork/test/merge) · `bend2-runtime` · `bend2-ownership` (copies at the language level).

C.25 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/01-duplication-superposition.md

--- title: "Bend 2 tutorial 06: duplication, superposition, and SupGen" description: "What dup and sup mean in the Interaction Calculus and HVM4, why search by superposition is documentation only, and which of SupGen, sup syntax, and copy-on-write forks the released Bend 2 compiler lacks." updated: 2026-10-18 verified: 2026-10-18 status: docs-only canonical: /tutorial/06-duplication-superposition/ sources:

- https://bend2.dev/notes/hvm4-explained/ (https://bend2.dev/notes/hvm4-explained/)
- https://bend2.dev/notes/search-by-superposition/ (https://bend2.dev/notes/search-by-superposition/)
- https://bend2.dev/notes/release-status/ (https://bend2.dev/notes/release-status/)
- https://bend2.dev/notes/bend2-vs-bend1/ (https://bend2.dev/notes/bend2-vs-bend1/)
- https://bend2.dev/learn/duplication/ (https://bend2.dev/learn/duplication/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

06: duplication, superposition, and SupGen

Your question mentions three things: copy-on-write with `sup` and `dup`, SupGen, and forks of running code. This chapter explains what the words mean in the lineage behind Bend, then shows what the released compiler accepts. Short version: the released Bend 2 has no `sup` syntax, no SupGen in its compiler or Base library, and no copy-on-write forks of running code. The rest of this chapter gives the evidence.

DUPLICATION AND SUPERPOSITION, IN THE CALCULUS

The Interaction Calculus, which HVM4 implements, has two kinds of term for sharing:

- A **duplication** `dup` takes one value and gives two copies, linked by a label. It is the term-level form of the DUP cell in chapter 04.
- A **superposition** `sup` holds two alternatives at once, also linked by a label. It is the term-level form of a choice that exists in several places.

The bend2.dev HVM4 page gives a worked example in its own syntax, where `&{1, 2}` is a superposition of two numbers. Adding 10 to it gives the two results 11 and 12. The page says the evaluator performs an addition for each alternative, and that duplication "still consumes work and memory". Sharing saves work only when the alternatives share their computation.

Labels decide how two terms meet. In the HVM literature, a `dup` and a `sup` with the same label route their parts to each other, and with different labels they commute, creating new `dup` and `sup` terms. Chapter 14 runs these rules on Victor Taelin's Interaction-Calculus reference implementation, which is a separate implementation from HVM4. HVM4 itself was not run.

> **Core insight.** `dup` and `sup` are two terms of one calculus. Their syntax differs, and they meet only through a label. Equal labels route, and different labels commute.

Composed of: a sharing term with label `L`.

```
sharing term, label L
- dup: !&L{x, y} = t; body     one value t, two uses x and y
- sup: &L{a, b}                two alternatives a and b, held in one place
```

Chapter 14 runs both rules. In its examples, equal labels take 2 interactions and different labels take 9.

SEARCH BY SUPERPOSITION

The bend2.dev note *Search by superposition* describes a technique built on this calculus. Apply a function to a superposition of its whole input domain, and the evaluator shares work across all inputs at once. The note reports results for a SAT instance and for program search. Those experiments used earlier HVM syntax. The page marks them **documentation only**: the numbers were reported by their author and were not re-run. Do not cite them as Bend 2 measurements.

WHAT THE RELEASED COMPILER ACCEPTS

These two programs are rejected by `bend 2.0.25`, and the messages are the real ones:

```
import Base

def main() -> U32:
  x = &{1, 2}
  (x + 10 : U32)
```

```
- expected : a name
- observed : '{'
```

`if` is not a keyword either. Bend branches with `match`, as chapter 01 shows:

```
import Base

def main() -> U32:
  if True{}:
    1
  else:
    2
```

```
- expected : '='
- observed : ':'
```

Adding a label does not help. The labelled form is rejected as well:

```
import Base

def main() -> U32:
  x = &0{1, 2}
  (x + 10 : U32)
```

```
- expected : ':'
- observed : ','
```

The released guide, Base, and the compiler say nothing about SupGen, and the bend2.dev pages document none. The site's own release-status note lists "Bend2–SupGen integration" among the launch blockers, and its HVM runtime note calls SupGen integration "still private". Treat SupGen as **not available** in the released compiler. Chapter 10 explains what you can do in its place.

WHAT BEND 2 HAS INSTEAD OF COPY-ON-WRITE

The released language has copying operations, but none of them is a copy-on-write fork of a running program:

- **`+` on a `Data` value** shares the value and counts the sharing. Nothing is copied, and the count is kept in 24 bits.
- **`Array.clone`** copies every slot into a second array. The cost grows with the array's size.
- **`Array.fork`** gives two handles to one array in constant time. Both handles read and write the same cells, so racing writes go through atomic operations. It is marked `@unsafe` in Base, which means the checker does not verify it, and no law may depend on it.
- **Templates** make one compiled copy for each distinct template argument. The copy happens at compile time, not while the program runs.

None of these forks a live program. A running Bend program is a value that the compiler has already compiled, and nothing in the language rewrites it.

EXERCISES

1. Read the error. Explain, in one sentence each, why `&{1, 2}` and `if` are rejected.

Solution. `&{` begins a superposition in HVM syntax, which the released parser does not accept, so it expects a name where it sees the brace. `if` is not a keyword, so the parser reads `if True{}` as the start of a binding and expects `=` where it finds `:`.

2. Two alternatives without superposition. Replace the idea of a superposed price with an ordinary choice. Write a `Choice` type with two constructors and a function that prices each one. Print both prices as a pair.

Solution. The alternatives are values, so they can be passed around, stored, and checked by laws. The output below is the pair of both prices.

```
import Base

type Choice is Data:
  Low{}
  High{}

def price(c: Choice) -> Nat:
  match c:
    case Low{}:
      10n
    case High{}:
      20n

def main() -> Nat & Nat:
  (price(Low{}), price(High{}))
```

```
(10n, 20n)
```

3. Why no law can use `Array.fork`. Explain what the `@unsafe` marker in Base changes for a law that depends on `Array.fork`.

Solution. The checker treats an `@unsafe` definition as outside its proof guarantees. A law whose proof depends on it is rejected in the proof check, so the law would be unverified. Chapter 11 shows the exact message.

NEXT

Chapter 07 covers laws and proofs, the part of Bend that checks properties of the values a program computes.

C.26 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/02-superposition-and-supgen.md

--- title: "Bend 2 tutorial 14: superposition, collapse, and SupGen" description: "The Interaction Calculus behind sup and dup, run on Victor Taelin's reference implementation: the rules and labels, collapse, search with erasure as the filter, what is public about SupGen, and whether a cheap fork is available." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/14-superposition-and-supgen/ sources:

- https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus)
- https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615 (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615)
- https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34 (https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34)
- https://gist.github.com/VictorTaelin/5d42e97684ff81f7e507aa5c149b2a64 (https://gist.github.com/VictorTaelin/5d42e97684ff81f7e507aa5c149b2a64)
- https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e (https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e)
- https://bend2.dev/notes/interaction-nets-explained/ (https://bend2.dev/notes/interaction-nets-explained/)
- https://bend2.dev/notes/hvm4-explained/ (https://bend2.dev/notes/hvm4-explained/) ---

14: superposition, collapse, and SupGen

This chapter is the one place in the tutorial where the code does not run on `bend`. It runs on the Interaction Calculus (IC), the calculus behind the `sup` and `dup` of chapter 06. Victor Taelin's reference implementation runs it on a CPU. Every IC program below was run with that binary, and the lines under each program are its real output.

WHAT IS VERIFIED HERE, AND WHAT IS QUOTED

- **Verified in this lab:** the rules, labels, and collapse results below, with their outputs and interaction counts (`WORK`). Reference build: `Interaction-Calculus` at commit `e9f0185`, default branch, CPU mode.
- **Quoted, not re-run:** the search results for SAT and program synthesis, the enumerator counts, and every SupGen claim. Each one names its gist or post.
- **Not verified:** wall-clock time and memory. The binary reports interaction counts and its own timing, and the timing does not compare across machines.

BUILD THE REFERENCE IMPLEMENTATION

```
git clone https://github.com/VictorTaelin/Interaction-Calculus.git
cd Interaction-Calculus && make
IC_BIN="$PWD/bin/ic" python3 bend-docs/scripts/check-tutorial.py bend-docs/src/content/tutorial
```

The checker runs `ic` blocks only when `IC_BIN` is set, and reports them as skipped otherwise.

THE CALCULUS IN SIX FORMS

Form	Syntax	Meaning
variable	`x`	used at most once (affine), but global: it may appear anywhere
erasure	`*`	a value that has been discarded
lambda	`λx.t`	a function
application	`(f a)`	apply `f` to `a`
superposition	`&L{a,b}`	two alternatives `a` and `b` with label `L`
duplication	`! &L{x,y} = t; body`	splits `t` into two uses `x` and `y` with label `L`

Labels are numbers. The grammar is the one in the calculus's README.

ONE PROGRAM PER RULE

Each program is followed by the lines the binary prints. `WORK` is the number of interactions.

APP-LAM: beta reduction is one interaction.

```
(λx.x λy.y)
```

```
λ$a.$a
WORK: 1 interactions
```

APP-SUP: applying a superposition runs both alternatives.

```
(&0{λx.x,λy.y} λz.z)
```

```
&0{λ$a.$a,λ$b.$b}
WORK: 5 interactions
```

The argument `λz.z` is duplicated, so each alternative gets its own copy.

DUP-SUP, same label: the duplication routes the parts.

```
!&0{x,y}=&0{λa.a,λb.b}; (x y)
```

```
λ$a.$a
WORK: 2 interactions
```

DUP-SUP, different labels: the two commute.

```
!&0{x,y}=&1{λa.a,λb.b}; (x y)
```

```
&1{λ$a.$a,λ$b.$b}
WORK: 9 interactions
```

DUP-LAM and the erasures.

```
!&0{x,y}=λz.z; (x y)
```

```
λ$a.$a
WORK: 3 interactions
```

```
(* λx.x)
```

```
*
WORK: 1 interactions
```

```
!&0{x,y}=*; (x y)
```

```
*
WORK: 2 interactions
```

LABELS: SAME ROUTES, DIFFERENT COMMUTE

The README says a superposition "works like a pair", and a duplication of a superposition "works exactly like a pair projection" when the labels match. The two DUP-SUP programs show both cases. With equal labels the duplication takes the pair apart, and the 2 interactions are routing and the final application. With different labels the duplication cannot route the alternatives, so it duplicates each one. The superposition comes out with label 1, at a cost of 9 interactions.

Labels also decide which alternatives are paired when two superpositions meet. The next program is the AND function applied to two boolean superpositions. True is `λt.λf.t`, False is `λt.λf.f`, and `(a b F)` computes AND. First, with different labels:

```
((λa.λb.(a b λt.λf.f)) &0{λt.λf.t,λt.λf.f} &1{λt.λf.t,λt.λf.f})
```

```
&0{&1{λa.λb.a,λc.λd.d},λe.λf.f}
WORK: 18 interactions
```

Next, with the same label on both inputs:

```
((λa.λb.(a b λt.λf.f)) &0{λt.λf.t,λt.λf.f} &0{λt.λf.t,λt.λf.f})
```

```
&0{λa.λb.a,λc.λd.d}
WORK: 12 interactions
```

Equal labels pair the alternatives: the first of one input meets the first of the other, so the result has two leaves, True and False. Different labels give every combination, and AND throws away the cases that do not matter. When the first input is False, AND never looks at the second, so the second superposition is erased in that branch. The result has three leaves. Labels and erasure, not just the function, decide the shape of the answer.

HOW THIS SITS ON INTERACTION NETS

The calculus uses the same three cells as chapter 04, and gives every DUP a label. Read the text as cells:

Text	Cell	Ports
`λx.t`	CON	principal: the lambda's output; aux: the variable `x` and the body `t`
`(f a)`	CON	principal: `f`; aux: `a` and the result
`!&L{x,y} = t; body`	DUP, label `L`	principal: `t`; aux: `x` and `y`; `body` is the rest of the net
`&L{a,b}`	DUP, label `L`, read from the other side	principal: the value; aux: the alternatives `a` and `b`
`*`	ERA	principal only

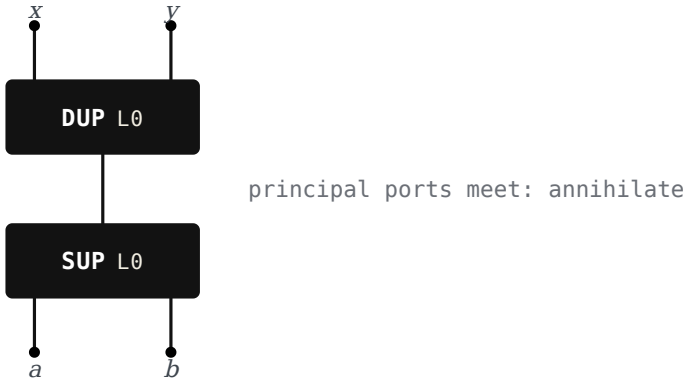
The rules are chapter 04's, with labels on the DUP cells. The counts are the ones the binary printed above:

Rule	In the net	Run	Interactions
APP-LAM	CON meets CON: β-reduction	`(λx.x λy.y)`	1
APP-SUP	CON meets a DUP: the application is copied	`(&0{λx.x,λy.y} λz.z)`	5
DUP-LAM	DUP meets CON: the lambda is copied, and its variable becomes a superposition	`!&0{x,y}=λz.z; (x y)`	3

Rule	In the net	Run	Interactions
DUP-SUP, same label	two DUPs with the same label annihilate; their aux wires join straight through	`!&0{x,y}=&0{λa.a,λb.b}; (x y)`	2
DUP-SUP, different labels	two DUPs with different labels commute	`!&0{x,y}=&1{λa.a,λb.b}; (x y)`	9
erasure	an ERA meets a CON or a DUP and erases it	`(* λx.x), !&0{x,y}=*; (x y)`	1 and 2

A superposition is not a new kind of cell. It obeys the rules of a DUP with its label: it commutes with an application, annihilates with a duplication of the same label, and commutes with a duplication of another label. The README says a superposition "works like a pair". That describes what the cell holds, two alternatives. The cell itself is a DUP read from the other side (chapter 04 draws both readings).

With equal labels, the two cells meet at their principal ports and annihilate. Each aux wire goes straight through, so `x` is wired to `a` and `y` to `b`:



A superposition is born in DUP-LAM. A duplication that meets a lambda copies the lambda, and the lambda's variable becomes `&L{X0,X1}` with the duplication's own label `L`. The two copies of the variable are the alternatives, and the body is duplicated behind them. The count fits this reading: `!&0{f,g}=λx.x; f` takes 2 interactions, one DUP-LAM and one annihilation of the body's copy with the same label. A different label would commute instead, and take more steps.

The README also notes that without labels the system cannot clone pairs, or clone a cloned lambda. The labels are what let the net copy a superposition. None of this runs in Bend 2; the reference binary runs it.

COLLAPSE: FROM SUPERPOSITIONS TO PLAIN RESULTS

The reference binary has a collapse mode. `ic run FILE -C` rewrites a result so that no superposition or duplication remains inside its alternatives. The outermost superposition stays, and each alternative is a plain term. The first program in the previous section already shows this: its output is a tree of plain λ-terms, with the labels kept only at the top.

SEARCH BY SUPERPOSITION, WITH ERASURE AS THE FILTER

A selector turns a result into a filter. The selector `λr.(r A *)` returns `A` when `r` is True, because True takes its first argument, and erases everything else when `r` is False. Applied to the AND of the two superpositions, it keeps exactly the input pairs for which AND is True:

```
((λa.λb.((λr.(r λx.λy.x *)) (a b λt.λf.f))) &0{λt.λf.t,λt.λf.f} &1{λt.λf.t,λt.λf.f})
```

```
&0{&1{λa.λb.a,*},*}
WORK: 37 interactions
```

One alternative survives: `A` for the single pair (True, True). Every other alternative is erased to `*`. This is search in its smallest form. The program is written once, the candidates are a superposition, and erasure keeps only the candidates that pass the test. The SAT note calls this "search without loops". The search does no enumeration in the program text, and the filter does the selection.

EXERCISE: KEEP THE FALSE CASES

The selector `λr.(r * A)` keeps `A` when `r` is False, because False takes its second argument. Predict how many alternatives survive, before you run the program below.

Solution: two, not three. The pairs (False, True) and (False, False) reach the same branch of AND, and that branch never uses the second input. So the second superposition is erased in that branch, and only one alternative is produced there. Counting alternatives counts the results that reach the output, not the assignments you might expect.

```
((λa.λb.((λr.(r * λx.λy.x)) (a b λt.λf.f))) &0{λt.λf.t,λt.λf.f} &1{λt.λf.t,λt.λf.f})
```

```
&0{&1{*,λa.λb.a},λc.λd.c}
WORK: 37 interactions
```

EXERCISE: PREDICT THE LABEL

Predict the label of the result of `!&0{x,y}=&1{λa.a,λb.b}; (x y)`, and the number of interactions. Then run it.

Solution: the label is 1 and the count is 9. The duplication has label 0 and the superposition has label 1, so they commute, and the result is a superposition with label 1.

```
!&0{x,y}=&1{λa.a,λb.b}; (x y)
```

```
&1{λ$a.$a,λ$b.$b}
WORK: 9 interactions
```

SEARCH IN THE PUBLISHED MATERIAL

The author's public work applies the same idea to larger problems. These results are quoted, not re-run here.

- **Program search, add-carry** (gist `fast_dps_add_carry`, updated July 2026). A template with 16 unknown bits is written once, its candidates are superposed, two test cases are applied, and the result is collapsed. The gist reports about 36 thousand interactions for the search, against about 262 million for 65,536 separate runs of a conventional loop. It is written in HVM1 syntax. The author says the full source will not be released.
- **SAT** (gist `sat.md`, 2024). Each boolean is a superposition of True and False, a formula is applied to all assignments at once, and the result is collapsed. The gist's text reports 3 minutes in Rust against 1 second in HVM for a 32-variable instance, but the code in the gist is a 16-variable instance. The author writes that he knows little about the asymptotic behaviour.
- **Enumerating λ-terms** (gist `superposed_lambda_calculus_evaluator.c`, 2025). The enumerator builds affine λ-terms directly as a superposition of constructors such as `#Lam` and `#App`, so the candidate programs are data. The gist reports about 76 thousand interactions for an untyped search, against about 32 million for a naive enumerator, and about 3 thousand for a dependently typed version. Its header says it mirrors a file in HVM3's book. That path returned HTTP 404 when this chapter was checked, so the mirror was not confirmed.

SUPGEN: WHAT IS PUBLIC

SupGen is the synthesizer these techniques were built for. Its source is not public. The evidence:

- The DPS gist says: "We'll not release the full source, but, meanwhile, I can talk about the theory." It also says HOC "will soon (EOY?) launch an API for our DPS solution".
- A 2025 HOC thesis gist says: "SupGen, our program synthesizer that outperformed existing solutions by up to 1000x." That is a claim, and no code accompanies it.
- A post surfaced by web search, which I did not open, says SupGen "could only discover functions when given their type signatures" until a breakthrough. A YouTube demo is titled "SupGen: an AI-free code synthesis tool". Neither was checked against source.

- Taelin's GitHub account lists 261 public repositories, and none is named SupGen. HigherOrderCO's public repositories are Kind, HVM1 to HVM4, Bend1, and Bend2, which now lives at `bendlang/bend`. Other GitHub repositories named `supgen` are unrelated, except `BabySupGen`, a browser playground that describes itself as inspired by the research.
- The tutorial's own notes agree. The release-status note lists "Bend2–SupGen integration" among the launch blockers, and the HVM runtime note calls the integration "still private".

So the mechanism is public: the calculus, its labelled rules, collapse, and the examples above. The synthesizer is not.

IS A CHEAP FORK OF RUNNING CODE AVAILABLE?

Partly, and only inside the calculus.

- **Yes, as alternatives inside one term.** A fork of a function can be written as a superposition of two versions. Applying it to an input runs both versions, and the input is shared through a duplication. In the program below the main version is the identity, and the branch is NOT. One run returns both answers:

```
(&0{λx.x,λx.(x λt.λf.f λt.λf.t)} λt.λf.t)
```

```
&0{λ$a.λ$b.$a,λ$c.λ$d.$d}
WORK: 8 interactions
```

The result `&0{True, False}` holds the main version's answer and the branch's answer side by side.

- **Limits.** The fork is an alternative inside one term. It is not a copy of a running process with its own memory. Its cost is counted in interactions, which is the author's metric. Memory and wall-clock time were not measured here, and the speedups quoted above are interaction counts.
- **No, in released Bend 2.** The compiler rejects `&{a, b}` (chapter 06), and BendRT does not reduce nets (chapter 04). A Bend program compiles to a fixed binary, and a branch is a second definition that you test with laws (chapter 10).
- **The homoiconic angle.** The enumerator builds programs as data, using constructors such as `#Lam`. That is the closest public thing to code that other code can build and inspect. Released Bend 2 has no such representation. Chapter 10 shows a small `Expr` interpreter in Bend, which is a limited form of it.

WHAT TO WATCH FOR

- A Bend release that adds `sup` syntax or the SupGen integration. The search note in the tutorial's notes says to re-read it when SupGen ships.
- A published SupGen source or API. Until then, every SupGen claim in this chapter is a quoted claim.
- An HVM4 or IC release that reports wall-clock results on the programs above. The interaction counts here do not give those numbers.

NEXT

Chapter 13 is the glossary. Chapter 10 applies the fork idea to Bend's laws-first workflow.

C.27 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/03-branching-and-self-change.md

--- title: "Bend 2 tutorial 10: branching and self-change" description: "What SupGen, copy-on-write forks, and homoiconic self-change would require, what the released Bend 2 compiler offers instead, and a laws-first workflow for testing a new branch of pure code." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/10-branching-and-self-change/ sources:

- https://bend2.dev/notes/hvm4-explained/ (https://bend2.dev/notes/hvm4-explained/)
- https://bend2.dev/notes/release-status/ (https://bend2.dev/notes/release-status/)
- https://bend2.dev/notes/ai-agents-bend/ (https://bend2.dev/notes/ai-agents-bend/)
- https://bend2.dev/learn/types-as-specifications/ (https://bend2.dev/learn/types-as-specifications/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

10: branching and self-change

The request behind this chapter asks for four things: SupGen, cheap copy-on-write forks of code (with `sup` and `dup`), homoiconic code that changes itself, and forking a running program with new code so that the main code keeps working and new branches can be tested cheaply. This chapter says which of these the released Bend 2 compiler offers, which it does not, and what to do instead. The short answer is that the part you can have is the laws-first workflow for pure code, and the rest is either absent from the compiler or cannot be stated as a law.

WHAT EACH REQUIREMENT NEEDS

Requirement	Status in bend 2.0.25	What you can do instead	Can a law pin it?
SupGen (type-directed program synthesis)	not in the compiler, Base, or guide; the site says it is private	write the program and its laws by hand	no
Copy-on-write fork with `sup` and `dup`	no `sup` syntax (chapter 06); no copy-on-write forks	values are immutable; `+` shares by count; `Array.clone` copies	value equality yes; sharing and cost no
Homoiconic, self-changing code	programs are fixed definitions; templates specialize at compile time	represent a program as `Data` and interpret it with `eval`	yes, for the interpreter's behaviour
Cheap fork of code	the calculus forks a term into labelled alternatives that share their input (chapter 14); released Bend 2 has no such construct	a new definition or constructor is a new branch; `main` is unchanged	the result yes; the cost no
Test new branches cheaply and fast	`bend --check-only` and `bend PROOF.bend` check the laws; native builds time the program	the laws-first gate below	the behaviour yes; the time no

Two lines in that table are absolute. A law states what a value computes, and the released compiler has no way to state the cost of sharing. Time and memory are measurements.

THE WORKFLOW THAT WORKS TODAY

For pure code, a branch is a new value or a new constructor, and the checker tests it against the same laws as `main`.

1. **State the requirement as laws.** Put the claims in `LAWS.bend`, and never weaken an accepted law to get a proof through.
2. **Implement the main behaviour as pure functions over `Data`.** Keep the input and output shell thin, so the laws can pin the pure part.
3. **Add a branch as a new constructor or a new definition.** Existing cases do not change, so the main laws keep their proofs.
4. **Write one law per branch,** plus laws that relate the branch to `main`. A *difference law* says where the branch may differ from `main`. A *fork law* says a branch that changes nothing behaves exactly like `main`.
5. **Prove in `PROOF.bend`.** The gate must print `All terms check.` with no line about `unsafe` or `foreign` code.
6. **Measure the branch separately,** with native builds, as chapter 05 describes.

The pattern works because the laws quantify over every constructor. A new constructor that is not covered by a proof makes the gate fail. The compiler enforces that every new branch carries its own proof.

PATTERN 1: A BRANCH AS A TEMPLATE

A template argument is a compile-time choice. Two different steps give two compiled copies of one loop, with no run-time selection:

```
import Base

def run(~step: U32 -> U32, n: Nat, x: U32) -> U32:
  match n:
    case 0n:
      x
    case 1n+p:
      run(~step, p, step(x))

def main_step(x: U32) -> U32:
  (x + 1 : U32)

def branch_step(x: U32) -> U32:
  (x + 2 : U32)

def main() -> U32 & U32:
  (run(~main_step, 3n, 0), run(~branch_step, 3n, 0))
```

```
(3, 6)
```

A law can pin the branch at one input, and the proof is computation. The checker evaluates the concrete U32 arithmetic, so `{==}` is enough:

```
import Base

def branch_step(x: U32) -> U32:
  (x + 2 : U32)

law branch_at_zero:
  {branch_step(0) == 2 : U32}

def branch_at_zero():
  {==}

def main() -> U32:
  branch_step(0)
```

```
2
```

A law over every input needs induction or a proof by cases, as chapter 07 shows. The concrete law is the smallest useful claim, and it runs in the same gate.

PATTERN 2: A PROGRAM AS DATA

Homoiconic code means that a program is also data that other code can build and inspect. Bend does not quote its own definitions, but you can represent an expression as a `Data` type and interpret it. A branch is then a different expression value, and the main expression is untouched. Bend is strict, so the data below is built and not computed: `eval` is the only step that computes a value. To keep work unevaluated, you can use a lambda (`x => ...`), a template (`~f`), or data like this. Chapter 12, section 11, shows all three:

```
import Base

type Expr is Data:
  Lit{n: Nat}
  Add{a: Expr, b: Expr}
  Mul{a: Expr, b: Expr}

def eval(e: Expr) -> Nat:
  match e:
    case Lit{n}:
      n
    case Add{a, b}:
      Nat.add(eval(a), eval(b))
    case Mul{a, b}:
      Nat.mul(eval(a), eval(b))

def main() -> Nat & Nat:
  main_expr : Expr = Add{Lit{2n}, Lit{3n}}
  branch_expr : Expr = Add{Lit{2n}, Mul{Lit{3n}, Lit{2n}}}
  (eval(main_expr), eval(branch_expr))
```

```
(5n, 8n)
```

Both values are pure, so evaluating the branch cannot change the main value. The interpreter `eval` is the place for laws: a law such as "`eval` of `Add{a, b}` is the sum of the evals" can be proved by a case split. This is homoiconic in a limited sense: the program is data you can build and test. It is not self-modification of the running binary, and it does not give you cheap sharing of two programs. Each `Expr` value is built and stored in memory, and the cost is what chapter 05 measures.

PATTERN 3: THE DIFFERENCE BETWEEN BRANCHES

The most useful law for a branch relates it to `main`. Suppose the branch should never give a larger answer than `main`. Write the claim, and let the gate check it:

```
law branch_never_more:
  for x: U32
  {U32.is_le(branch_step(x), main_step(x)) == True{} : Bool}
```

That law is false for the example above, because `branch_step` adds 2 and `main_step` adds 1. The gate would reject it, and that is the point: a difference law states exactly how the branch may differ from `main`, and a branch that breaks it is not accepted. Write the law you want, then change the branch until the gate accepts it, not the other way round.

WHAT IS NOT POSSIBLE, AND WHY

- **SupGen** is not in the released compiler. Nothing in this chapter depends on it, and no law can refer to it.
- **Copy-on-write forks with `sup` and `dup`** need a construct the parser rejects (chapter 06). The nearest things are `Array.fork`, which is `@unsafe` and so cannot support a law, and `Array.clone`, which copies.
- **Bend 2 does not fork a running program.** It compiles each program to a fixed binary, and a second program is a second file that you compile. The calculus forks terms, not processes: a superposition holds two alternatives, and applying it runs both on one shared input (chapter 14). The released compiler does not expose that.
- **Cost is not a law.** The checker proves that a branch computes the right values. It does not prove that the branch is cheap. Measure it, count the keeps in the emitted C (chapter 05), and compare builds.

EXERCISES

1. A third step. Add a template step `triple_step` that adds 3, and make `main` return the three results as a triple. Predict the output before you run it.

Solution. The result is `(3, 6, 9)` for three steps of three applications each. Triples are written `A & B & C`.

```
import Base

def run(~step: U32 -> U32, n: Nat, x: U32) -> U32:
  match n:
    case 0n:
      x
    case 1n+p:
      run(~step, p, step(x))

def main_step(x: U32) -> U32:
  (x + 1 : U32)

def branch_step(x: U32) -> U32:
  (x + 2 : U32)

def triple_step(x: U32) -> U32:
  (x + 3 : U32)

def main() -> U32 & U32 & U32:
  (run(~main_step, 3n, 0), run(~branch_step, 3n, 0), run(~triple_step, 3n, 0))
```

```
(3, 6, 9)
```

2. A law for a concrete input. Write the law that `triple_step(0)` is `3`, and prove it by computation.

Solution. The same shape as `branch_at_zero`. The concrete arithmetic reduces, so `{==}` proves it.

```
import Base

def triple_step(x: U32) -> U32:
  (x + 3 : U32)

law triple_at_zero:
  {triple_step(0) == 3 : U32}

def triple_at_zero():
  {==}

def main() -> U32:
  triple_step(0)
```

```
3
```

NEXT

Chapter 11 lists the checker's rules and every official diagnostic with the message the released compiler prints.

C.28 · bend-gme/version-1/.opencode/skills/bend2-superposition/references/04-synthesis-concepts.md

--- title: "Synthesis with SupGen" description: "Holes, types as specifications, and the synthesizer that proposes what the checker disposes." updated: 2026-09-05 verified: 2026-09-05 status: docs-only canonical: /learn/synthesis/ sources:

- https://bend2.dev/learn/ `(https://bend2.dev/learn/)` ---

Synthesis with SupGen

SupGen turns precise types into programs: holes mark the unfinished, specifications set the target, and search fills the gap.

The one idea Precise types give synthesis a search target and the checker an acceptance condition. SupGen proposes implementations; the type checker disposes. See search by superposition for the machinery that makes each guess cheap.

HOLES

A hole marks an unfinished program the synthesizer should fill — the `?name` of Kind's grammar, expected in Bend2's surface *(planned)*.

TYPES AS SPECIFICATIONS

"Sorted list", "parser round-tripping with its printer": types precise enough to pin behavior down. Writing them is already proving half the program; the rest is search.

SUPGEN

Named on the March 2026 launch-blocker list, demonstrated nowhere public. Interface, examples, and limits arrive with the release — this page will be rewritten from the compiler.

C.29 · `bend-gme/version-1/.opencode/skills/bend2-superposition/references/05-search-by-superposition.md`

--- title: "Search by superposition" description: "No loops, no enumeration: apply the function to every input at once and let sharing do the search. SAT in 1s, add-carry from 65,536 candidates." updated: 2026-09-05 verified: 2026-09-05 status: docs-only canonical: /notes/search-by-superposition/ sources:

- https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615 `(https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615)`
- https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34 `(https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34)`
- https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e `(https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e)` ---

Search by superposition

Superposition is usually introduced as "two values in one location" with a two-line arithmetic demo. Its serious use is different: **search without loops**. Apply a function to a superposition of its entire domain and you get a superposition of its entire image — with the evaluator sharing work across all guesses at once.

Status The experiments below ran on earlier HVM syntax and are **documentation-only** here: shapes and numbers as reported by Taelin, not re-run. The mechanism (first-class SUP + collapse) is unchanged in HVM4.

TRUTH TABLES FOR FREE

Evaluating `(And {True False} {True False})` yields the complete truth table — `{{True False} {False False}}` — because applying a function to a superposed domain produces the superposed image. So brute force stops being written as nested loops:

```
# instead of this:
for x0 in [T, F]:
    for x1 in [T, F]:
        ...
        print(F(x0, x1, ...))
# write this:
print(F({T,F}, {T,F}, ...))
```

The second form shares computation across guesses wherever the guesses agree — which, for structured problems, is nearly everywhere.

SAT: 3 MINUTES IN RUST, 1 SECOND IN HVM

Taelin's one-line-style SAT solver feeds all 2^{16} assignments of a random 16-variable 3-SAT instance as superposed booleans, lets the formula evaluate once across the whole superposition, then collapses per-label results back into the satisfying assignment. A 32-variable instance: ~3 minutes of Rust brute force vs ~1 second superposed. The honest caveat is his own: this was a new observation, asymptotics unknown — but the shape of the win (shared prefixes across guesses) is exactly what the machinery predicts.

PROGRAM SEARCH: 65,536 CANDIDATES, 36K INTERACTIONS

The Discrete Program Search series pushes the same idea up a level: synthesize the add-carry function from a template with 16 unknown bits. Conventionally that's 65,536 separate runs (~262M interactions at ~4k per call). Superposed — one template holding all candidates, two test cases, collapse — it returns the bit vector `[0,1,1,0,1,0,0,1,1,1,1,0,1,0,0,0]` in **36k interactions**: under one interaction per guess, a ~7200x win that grows with the search space. Enumeration and loops are fully replaced by superposition and collapse; the follow-ups extend it to DSL enumeration and superposed λ-terms.

WHY THIS PAGE EXISTS IN BEND2'S NOTES

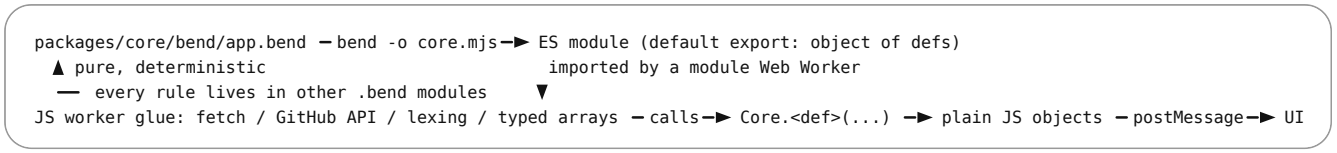
Bend2's SupGen — synthesis from precise types — is this technique aimed at programs instead of bits: types give the search a target, the checker an acceptance test, and superpositions make each guess cost less than one. The DPS posts even sketch the product shape (examples in, function out). When SupGen ships, re-read this page first.

C.30 · `bend-gme/version-1/.opencode/skills/bend2-effects-targets/SKILL.md`

Bend 2: effects, targets, and the Web Worker FFI

Bend computes; the host moves bytes. On the JS target the boundary is one file and plain JS objects.

MODEL



- Effects inside Bend (`do` , files, args, fork/join) exist for CLI programs and tests (`references/01-io-files-and-targets.md`). **The app core uses none of them.**
- On JS: constructors are `{$: "Name", field}` ; lists `{$:"Con", head, tail}` / `{$:"Nil"}` ; pairs `{$:"Tuple", fst, snd}` ; numbers are JS numbers; arrays are JS arrays passed **without copy**. Values survive structured clone, so `postMessage` and IndexedDB work (`references/03-js-target-facts.md` §FFI shape).
- Calls run through a trampoline (`run_lib`), so deep **tail** recursion is safe; non-tail recursion over ~5k frames overflows the JS stack.

RULES

1. One FFI surface file (`app.bend`), packing only; JS calls only its defs (imported-module defs appear as `"module:def"` ; do not call them).
2. No network, DOM, clock or randomness in Bend. JS passes `now` , seeds and bytes in; Bend returns decisions and view models.
3. Integers across the boundary for exact values (cents, day/minute ints, U32 words); F32 is `Math.fround` precision and loses integers past 2^{24}.
4. Columns enter as power-of-two padded arrays plus an explicit length `n` ; Bend bounds-checks every index (arrays wrap).
5. Long folds are tail-recursive with an accumulator; anything that could exceed ~5k frames is reviewed.
6. Version the surface: `api_version()` in `app.bend` ; JS refuses a mismatched `core.mjs` .
7. Ship `LICENSE-bend` next to the generated module (its runtime is Apache-2.0).
8. Library mode (a module with no `main`) needs bend ≥ 2.0.39; 2.0.25 can build JS only from a file with `main` .
9. **File IO exists, but not on the interpreter lane.** Base has `File.open/read/read_bytes/size/write/close` ; however `bend file.bend` (checker/interpreter lane) fails with `no effect registers File.open` — File effects are implemented only for compiled targets (`effs/file_*.c` / `file_*.js`). So a direct, no-compile IO test is **not** possible on 2.0.39: a file-reading test must be compiled (JS/native). Everything else (parsing, strategies) can still be tested directly with inline fixtures.
10. **Never read a `U32` scrutinee in the fallback arm of a literal `match`** (`case 7: … case _x: f(op)`): on the JS lane the match is compiled to a bit decision tree and the fallback rebuilds the value as a 32-cell `Word` (`u32_to_word` / `word_to_u32` , two allocations of 32 cells per call). It was 56 % of the VM's time (R-37). Use a comparison chain (`Bool.pick(U32, U32.is_le(op, 11), …)` , fine when every arm is a constant — `Bool.pick` is eager)

or, in a recursive function, a precomputed `Bool` parameter matched instead (`walk_go(…, last: Bool, …)`). A literal match whose fallback is a constant (`case _: 0`) is free. `packages/core/build.mjs` fails the build when any emitted function other than `lang/compile:f32_cmp` contains a Word conversion.

PATTERNS

```
BEND_NO_TELEMETRY=1 bend packages/core/bend/app.bend -o packages/core/dist/core.mjs   # JS module (this repo)
BEND_NO_TELEMETRY=1 bend file.bend -o out.js                                          # JS, single file

# The installed bend 2.0.39 binary cannot emit C/native here (`-o out.c`, `-o out`
# both die with `TypeError: ... 'FL.book.tlds[k].c'`, even for hello world).
# Re-check before using this (`bend hello.bend -o h && ./h`); if it builds, drop
# this note. While it fails, the machine is fine -- the source CLI of the same tag:
#   bun /tmp/bend-src/bend2/main.ts out.bend -o out     # bend-src = tag v2.0.39
# The JS target of the same file builds with the installed binary, so the app,
# its tests and the tools are unaffected. Native is also not automatically faster
# (bend2-runtime has the measured table: 2.3x slower for the cons-heavy VM).
```

```
// worker glue (TypeScript, in the UI package's worker entry; sketch)
import Core from "@bend-gme/core"        // bare specifier, not a file path
self.onmessage = ({ data }) => {          // data decoded with Effect Schema first
  const out = Core.step(data.state, data.input)   // opaque Bend value in, view model out
  self.postMessage(out)
}
```

MODULE SHAPE AND IMPORTS (`.MJS`, `.JS`, `.TS`)

This trips people up, so it is a rule, not a preference:

File	Language	How it is reached
`packages/core/dist/core.mjs`	**generated JS** by `bend -o`	via the package `exports` map, never by path from app code; **never edited**
`packages/core/index.d.ts`	hand-written TS declaration	`exports["."].types`; the contract the glue compiles against
`packages/core/conv.ts`	hand-written TS	subpath export `@bend-gme/core/conv` (source, not built)
app/worker/protocol `.ts`	TypeScript	normal TS imports

Why `.mjs` and not `.js` or `.ts`:

- The core **is not TypeScript**. `app.bend` is the source; Bend emits a JS ES module. Naming it `.mjs` marks it as generated ESM and keeps it working even if the package `type` changes. With `"type": "module"`, `dist/core.js` would behave identically; the extension is convention, and `build.mjs` + `exports` are the only places to change it.
- Application code imports the **bare specifier** (`@bend-gme/core`). TypeScript sees `index.d.ts`; Vite and the worker see `dist/core.mjs`. A relative `.mjs` import inside app code is a smell — the only legitimate one is the core's own test, which must exercise the exact emitted artifact (`import Core from "../dist/core.mjs"`).
- TypeScript files import each other with a **`.js` specifier** (`from "../src/index.js"`): with `verbatimModuleSyntax` + `moduleResolution: "Bundler"`, the specifier you write is what is emitted, and TS resolves `.js` back to the `.ts` source. Writing `.ts` would require `allowImportingTsExtensions`.
- Keep `index.d.ts` in sync with `app.bend` by a test that parses the `def`s out of the source and compares them with the module's exports (this repo: `packages/core/test/ffi-surface.test.ts`).

minebender layout we copy: `ui.bend` (≈50 lines, the whole FFI) → `ui.mjs` (generated, never edited) → `worker.js` (glue) → `index.html` (draws). Its notes: `references/04-minebender-notes.md`.

PITFALLS

Symptom	Cause	Fix
`RangeError: Maximum call stack size` in the worker	non-tail recursion on a long list	accumulator + `List.reverse`
prices off by one cent	money in F32	integer cents end to end

Symptom	Cause	Fix
a read returns a wrong but plausible value	array index wrapped	check against `n`
`bend -o x.mjs` refuses a file without `main`	2.0.25	upgrade to the pinned 2.0.39
UI freezes on a click	Bend running on the main thread	all Bend calls in workers

VERIFY

```
bend version                              # 2.0.39 in CI (packages/core/BEND_VERSION)
bend app.bend -o /tmp/app.js              # effectful entry: type, ownership, termination all checked here
# `--check-only` exits 1 for any program with a foreign effect ("defs rely on unsafe or foreign\n# code"), including the guide's own Clock.now — the verdict is not a signal about your file
bun run --cwd packages/core test:bend:changed   # tests the change reaches, compiled (the shipping lane) and cached
bun run --cwd packages/core test:bend:direct > /tmp/direct.log 2>&1 &   # parity: interpreter vs JS backend
```

REFERENCES

File	Covers	Open when
`references/01-io-files-and-targets.md`	`do`, fork/join, files, args, error results, targets	CLI tools, tests, building
`references/02-effects-targets-concepts.md`	concept page	quick refresher
`references/03-js-target-facts.md`	verified JS-target facts: numbers, wraps, stack, arrays, FFI shape, minebender lessons	any JS-target work
`references/04-minebender-notes.md`	the reference app's own notes (MIT, Luís Cleto)	copying the worker pattern

RELATED

`foldkit-verified-ui` (the UI side: ManagedResource + Subscription for workers) · `bend2-app` · `bend2-ownership` (arrays) · `bend2-runtime` (parallelism = many workers).

C.31 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/01-io-files-and-targets.md

--- title: "Bend 2 tutorial 09: IO, files, and targets" description: "The IO type and do blocks, binding effect results with a type, fork and join, channels, affine file handles, command-line arguments, and the C, JavaScript, and native targets." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/09-io-files-and-targets/ sources:

- https://bend2.dev/learn/io/ (https://bend2.dev/learn/io/)
- https://bend2.dev/learn/concurrency/ (https://bend2.dev/learn/concurrency/)
- https://bend2.dev/learn/files/ (https://bend2.dev/learn/files/)
- https://bend2.dev/learn/compile-to-c/ (https://bend2.dev/learn/compile-to-c/)
- https://bend2.dev/learn/compile-to-javascript/ (https://bend2.dev/learn/compile-to-javascript/)
- https://bend2.dev/notes/what-is-bend2/ (https://bend2.dev/notes/what-is-bend2/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

09: IO, files, and targets

Bend is pure. Effects live in the type `IO`, and a `do` block sequences them, much as Haskell's `IO` monad does. The differences are that every bind names its type, and that a file handle is affine: an operation on a handle gives the handle back with its result. This chapter covers the effect forms, files, arguments, and the three compilation targets.

SEQUENCING EFFECTS

A `do` block runs its lines in order. A bind `x : T <- action` runs an action and names its result with a type. A plain line runs an action whose result is `Unit`, and `return v` ends the block with a pure value:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("first")
    IO.print("second")
```

```
first
second
```

The last line of a `do` block must be an effect or a `return`. A pure term there is rejected:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    (6 * 7 : U32)
```

```
- observed : U32
```

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1)
    return "Hello, " ++ name

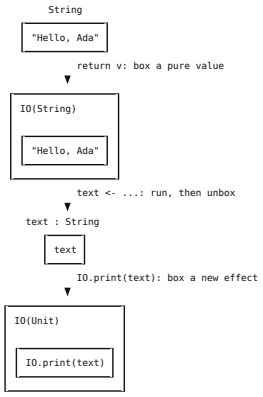
def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    IO.print(text)
```

```
Hello, Ada
```

In that example `IO.sleep(1)` waits one millisecond, and `return` wraps the pure string as an `IO` result. Every bind carries a type, as in `text : String <- greet("Ada")`. Without it, the checker cannot choose the monad's type arguments.

BOXES: PURE, BIND, AND EFFECTS

The picture is a model, not the runtime. An `IO(A)` is an effect: when it runs, it produces an `A`. Read the outer box as the effect and the inner box as the value, which exists only after the effect runs. A `do` block moves values between the boxes:



The same program, checked:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    IO.print(text)
```

```
Hello, Ada
```

- `return v` boxes a pure value. It is `IO.pure`.
- `text : String <- greet("Ada")` runs the effect and takes the value out. This is the only way out of a box.
- `IO.print(text)` makes a new effect: a box of type `IO(Unit)`.

WHAT A DO BLOCK MEANS

The guide defines `do` by two rules. `x : A <- v` becomes `M.bind(xs.., A, R, v, x => ...)`, and `return e` becomes `M.pure(xs.., R, e)`. For IO, Base defines `IO.pure(A, x)` and `IO.bind(A, B, m, f)`. The program above is the same as these calls:

```
import Base

def greet(name: String) -> IO(String):
  IO.pure(String, "Hello, " ++ name)

def main() -> IO(Unit):
  IO.bind(String, Unit, greet("Ada"), text => IO.print(text))
```

```
Hello, Ada
```

A plain line is a bind whose result is `Unit`, so two effects in a row are one bind with a lambda that ignores its argument:

```
import Base

def main() -> IO(Unit):
  IO.bind(Unit, Unit, IO.print("x"), u => IO.print("y"))
```

```
x
y
```

MATCHING ON IO

An `IO(A)` has no constructors, so a constructor pattern on it is rejected:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def show(io: IO(String)) -> IO(Unit):
  match io:
    case Some{x}:
      IO.print("some")
    case None{}:
      IO.print("none")

def main() -> IO(Unit):
  show(greet("Ada"))
```

```
- expected : a datatype
```

A `match` also cannot head a statement inside `do`, because a `match` heads a def body. Binding a value and then matching it in the same block is rejected:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("Ada")
    match text:
      case _:
        IO.print(text)
```

```
- expected : a term (a match heads a def body, not a term)
```

A wildcard `case _:` on an IO parameter is accepted, but its branch cannot see inside the effect. The pattern that works is to bind the value with `<-` and pass it to a def whose `match` inspects it:

```
import Base

def find(n: U32) -> IO(Maybe<&2, U32>):
  do IO<Maybe<&2, U32>>:
    return Some{n}

def show(m: Maybe<&2, U32>) -> IO(Unit):
  match m:
    case Some{x}:
      IO.print(U32.show(x))
    case None{}:
      IO.print("none")

def main() -> IO(Unit):
  do IO<Unit>:
    m : Maybe<&2, U32> <- find(7)
    show(m)
```

```
7
```

CONCURRENCY: FORK AND JOIN

`IO.fork` starts an effect in the background and returns a channel for its result. `IO.join` waits on that channel. Between the two, the main effect keeps running:

```
import Base

def delayed() -> IO(String):
  do IO<String>:
    IO.sleep(1)
    return "done"

def main() -> IO(Unit):
  do IO<Unit>:
    channel : Chan(String) <- IO.fork(String, delayed())
    IO.print("waiting")
    text : String <- IO.join(String, channel)
    IO.print(text)
```

```
waiting
done
```

The event loop interleaves the effects. It is not a parallel call: an effect that waits on a timer, a socket, or a channel steps aside for the others. The guide says each computation runs its pure code, possibly in parallel on every core, up to its next effect. Use `IO.fork` for work that waits, and a parallel call (chapter 05) for pure work that divides.

`Chan.new`, `Chan.send`, `Chan.recv`, and `Chan.close` are the low-level operations under `IO.fork` and `IO.join`. Their signatures are in Base, and `bend base Chan` prints them.

CHANNELS: VALUES BETWEEN COMPUTATIONS

`IO.fork` returns a channel for one result. For a stream of values, make a channel with `Chan.new`, give the sending computation the channel, and receive from the other side. `Chan.new(U32, 4)` makes a channel with room for four values. `Chan.recv` returns `Some{value}` for each value sent, and `None{}` once the channel is closed:

```
import Base

def send_numbers(+chan: Chan(U32)) -> IO(Unit):
  do IO<Unit>:
    sent1 : Bool <- Chan.send(U32, chan, 10)
    sent2 : Bool <- Chan.send(U32, chan, 20)
    Chan.close(U32, chan)

def show_message(m: Maybe<&1, U32>) -> IO(Unit):
  match m:
    case None{}:
      IO.print("closed")
    case Some{n}:
      IO.print(U32.show(n))

def main() -> IO(Unit):
  do IO<Unit>:
    +chan : Chan(U32) <- Chan.new(U32, 4)
    IO.spawn(Unit, send_numbers(chan))
    first : Maybe<&1, U32> <- Chan.recv(U32, chan)
    show_message(first)
    second : Maybe<&1, U32> <- Chan.recv(U32, chan)
    show_message(second)
    third : Maybe<&1, U32> <- Chan.recv(U32, chan)
    show_message(third)
```

```
10
20
closed
```

`IO.spawn` starts a computation whose result nobody waits for. Here the sender runs in the background, and the receiver reads the values in the order they were sent.

FILES: AFFINE HANDLES

A file is a handle, and a handle has one owner. Each operation on it returns the handle beside its result, so you must keep the handle and pass it on. `File.open` returns a `Result`, and `IO.try` unwraps it or ends the program with the error. This program writes a file and closes it:

```
import Base

def finish(result: File & Result<&1, &1, U32 & String, Unit>) -> IO(Unit):
  match result:
    case (file, status):
      do IO<Unit>:
        File.close(file)
        IO.pass(Unit, status)
        IO.print("written")

def main() -> IO(Unit):
  do IO<Unit>:
    file : File <- IO.try(File, File.open("message.txt", "w"))
    result : File & Result<&1, &1, U32 & String, Unit> <- File.write(file, "Hello from Bend")
    finish(result)
```

```
written
```

The program writes `message.txt` in the directory where you run it, and it replaces any earlier contents. The `finish` helper closes the handle on every path before it passes the status along. A handle that is not closed is a leak: the checker does not force you to close it, so you must do it yourself.

The guide's list of effect kinds includes TCP and UDP sockets, windows, and audio. Their handles follow the same affine rule. `bend base` prints their signatures.

THE NETWORK: TCP IN BASE, HTTP IN THE LIBRARY

Base has sockets, not the web: `Tcp.listen`, `Tcp.accept`, `Tcp.connect`, `Tcp.send`, `Tcp.recv`, `Tcp.poll`, `Socket.close`, `Listener.close`. There is **no HTTP client, no TLS and no DNS resolver** in Base 2.0.39; `Tcp.connect` takes an IPv4 literal.

But the ecosystem has them, and this is the default choice: `bend-kit-http`. (Apache-2.0, published on the Bend hub by name + hash, so there is nothing to vendor.) It ships HTTP/1.1 **and HTTP/2**, DNS, TLS through OpenSSL 3, proxies, redirects, retries, gzip/deflate/br/zstd, a connection pool, and laws and proofs per package. The TLS effect is `bend-kit-wire@0.4.7.0/effs/wire.js` — OpenSSL through `bun:ffi` — so it runs on the **JS lane**, not only native.

```
import Base
import bend-kit-http@0.38.2.0/http.bend as Http

def go(tok: String) -> IO(Unit):
  do IO<Unit>:
    got : Result<&1, &1, Http.Err, Http.Res> <- Http.fetch("GET",
      "https://api.marketdata.app/v1/options/expirations/GME/?date=2026-10-29",
      Http.set(Http.empty(), "authorization", "Bearer " ++ tok),
      Http.from_string(""))
    match got:
      case Done{res}: IO.print(Http.text(res))
      case Fail{err}: IO.print("fail")
```

Verified on 2026-10-31: that call returned the real 18-expiration JSON, from the repo root with the token in `.env`.

Five rules that cost a debugging round each if you skip them:

1. **Pin the compiler the package was published for.** `bend-kit-http@0.38.2.0` checks on **2.0.38** and fails on **2.0.39** (`Chan.send`'s result type changed, so `bend-kit-concurrency@0.2.0.0` no longer type-checks). The core stays on 2.0.39; a tool that uses the kit pins 2.0.38 in its own `bend-version` file.
2. **Compile, then run with Bun: `bend main.bend -o out.js && bun out.js`.** `bend file.bend` spawns Bun without the `.env` file, so `IO.get_env` finds nothing; and the interpreter lane overflows on a body over ~30 KB. The compiled JS lane has neither problem.
3. **`Http.Res` holds packed `Bytes`, which are linear.** `Http.text(res)` consumes the response and returns the body as a `String`; `Http.length` returns `body & U32`. You cannot read the status and then the body from one `Res` — take the text and parse it, or destructure once and use each part once.

4. **Match the error tags**: `ErrUrl`, `ErrDns`, `ErrConnect{code, why}`, `ErrTls{code, why}`, `ErrRead`, `ErrWrite`, `ErrTimeout`, `ErrRedirect`, `ErrBad`. There is no `Http.Err.show`.
5. **Parse JSON with `bend-kit-json`** (`@0.7.0.0`, already in the dependency graph) rather than hand-scanning a string. File effects come from `bend-kit-files`.

The rest of the field, for when the kit is not the right fit: **bendo** (MIT) is a full HTTP/1.1 server and proxy client over Base TCP, native lane, TLS deliberately behind a proxy — read it for structure and its `src/http/parse.bend`; **ezhttp** is a packaged client and server on the hub; **bender-http** (MIT) is a single HTTPS client with a proven core but targets Bend 2.0.32 and needs a sibling `bend-dns` clone; **Scrapanium** wraps curl-impersonate/BoringSSL for anti-bot work; **Stiff** and **Gralha** bundle HTTP among other things.

`naoeosavio/bend-net` is not `bend-kit`, despite the shared name (`bend-kit` was renamed *from* bend-net, and the hub's `bend-net-*` names resolve to it). It is an independent Apache-2.0 library with the widest protocol surface in the field — HTTP/1.1 + HTTP/2 client **and server**, TLS client and server, ALPN, WebSocket, ~134 test paths. But its published hub bundle is **pre-TLS**: `<hash>/http.bend` says *"https:// parses but fetch refuses with 501"* and `send_request` takes no headers. Use the repo as reading or vendor it at a commit; do not depend on the hub hash.

Search the directory before writing any of this: `https://builtwithbend.com/?q=http&category=libraries` returns eight libraries, and the plain GitHub search misses most of them.

Two traps: never put a token in `argv` (`ps` reads it — the kit never needs to, since the header goes in memory), and a host socket helper exists only when the JS runtime runs the effect loop (an `.mjs` library has no `bun:ffi`).

COMMAND-LINE ARGUMENTS

`IO.args()` returns the arguments after the program name, as a `List<String>`. The compiler's own options are removed, and `--` ends them. Here the program prints how many arguments it received:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    args : List<String> <- IO.args()
    IO.print(Nat.show(List.length(&1, String, args)))
```

```
0
```

Run it with arguments to see the count change:

```
bend main.bend one two
bend main.bend -- one two
```

Both commands print `2` in this lab. The `List.length` call takes its quantity first, `&1` for a list of affine elements, which is why the version without it is rejected (chapter 11).

ERROR RESULTS AND EXITS

Two Base effects end a program on purpose. `IO.die(-A, code, msg)` ends the program with the exit code and the message. `IO.pass(-A, r)` turns a `Result` into an effect that either returns the value or fails with its error. Prefer `IO.pass` and `IO.try` when the failure can be handled, and reserve `IO.die` for a program that cannot continue.

READING THE ENVIRONMENT

`IO.get_env` returns a `Result`. Matching on it handles the failure case. This variable is unset, so the program prints `missing`:

```
import Base

def describe(r: Result<&1, &1, U32 & String, String>) -> IO(Unit):
  match r:
    case Fail{e}:
      IO.print("missing")
    case Done{v}:
      IO.print(v)

def main() -> IO(Unit):
  do IO<Unit>:
    r : Result<&1, &1, U32 & String, String> <- IO.get_env("BEND_TUTORIAL_UNSET")
    describe(r)
```

```
missing
```

COMPILATION TARGETS

Bend has three targets. The same source compiles to each of them.

```
bend main.bend -o main.c
bend main.bend -o main
./main --threads 8
bend main.bend -o main.js
node main.js
```

- `-o main.c` writes one C file that contains the runtime and your program. Compile it with any C compiler, or inspect it, as chapter 05 does.
- `-o main` builds a native executable with clang 14 or newer. A binary that uses `!` also writes `main.gpu`, which must stay beside it. `--gpu off` runs bang calls on the CPU.
- `-o main.js` writes standalone JavaScript. The JavaScript target runs parallel bindings one after another.

The native and JavaScript targets print the same text, which this lab checked for the program below. It is the example from the compile-to-JavaScript page:

```
import Base

def main() -> IO(Unit):
  IO.print("Bend on JavaScript")
```

```
Bend on JavaScript
```

`bend main.bend` without `-o` runs the program directly. An `IO` main runs compiled, and a pure main is evaluated by the checker, so a pure main is slow for large work.

EXERCISES

1. Two greetings. Write `main` so that it runs `greet("Ada")` and then `greet("Lin")`, printing each greeting on its own line.

Solution. Two binds in one `do` block run in order.

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    return "Hello, " ++ name

def main() -> IO(Unit):
  do IO<Unit>:
    first : String <- greet("Ada")
    second : String <- greet("Lin")
    IO.print(first)
    IO.print(second)
```

```
Hello, Ada
Hello, Lin
```

2. Count the handles. Explain why `File.write(file, ...)` returns the handle again, and what would break if it did not.

Solution. A file handle is affine. If the write consumed the handle and returned only the result, the program could not close it afterwards. The handle would be lost, and the file would stay open. Returning the handle with the result keeps the single owner, so the program can close it on every path.

NEXT

Chapter 10 applies everything so far to your questions: forks, branches, and self-changing code.

C.32 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/02-effects-targets-concepts.md

--- title: "Effects and targets" description: "IO and files on the CPU, arguments, and compiling Bend2 to C, JavaScript, and Python." updated: 2026-09-05 verified: 2026-09-05 status: docs-only canonical: /learn/effects-targets/ sources:

- https://bend2.dev/learn/ `(https://bend2.dev/learn/)` ---

Effects and targets

Effects stay on the CPU in typed `do` blocks; pure code may fly to the GPU or compile to C, JavaScript, and Python.

IO

Effects run on the CPU in typed `do` blocks (hello) — `input`, `print`, sequenced with `<-`.

FILES

Files extend the same `IO` surface *(planned)*.

COMMAND-LINE ARGUMENTS

Command-line arguments extend the same `IO` surface *(planned)*.

COMPILING TO C

HVM4's ahead-of-time compiler targets native C *(current runtime tooling)*.

COMPILING TO JAVASCRIPT

Bend2's JavaScript target is planned.

COMPILING TO PYTHON

Bend2's Python target is planned. Flags, outputs, and the GPU story per target arrive with the toolchain — the one hard rule is already known: pure code may fly to the GPU, effects stay on the host.

C.33 · bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/03-js-target-facts.md

Bend 2 platform facts that shape this design

Everything here was checked on this machine on 2026-10-30 (the local `bend` is 2.0.25), against the minebender repo (built with 2.0.39), `bend guide`, `bend base`, the generated JS, and `WONTFIX.txt` plus issues in `github.com/bendlang/bend`. When this file and your memory disagree, trust this file, then re-check with the commands below.

TOOLCHAIN

Fact	Evidence	Consequence
Latest release is **2.0.39** (`bendlang/bend`, Apache-2.0).	`install.sh` served by bend-lang.com pins `VER="2.0.39"`.	Pin `2.0.39` in `packages/core/BEND_VERSION` (and the CI variable); CI fails on a mismatch.

Fact	Evidence	Consequence
2.0.25 cannot emit a library module: `bend x.bend -o x.mjs` without `main` fails with "no main to run". minebender's `bend ui.bend -o docs/ui.mjs` (no `main`) works on 2.0.39.	Tried locally.	**The first task is upgrading Bend.** `bend update` runs `curl | sh`; show the script to the user first (their stated preference).
`bend page.html -o dist` does not follow `new Worker(...)`.	minebender `NOTES.md`.	Bend does not bundle the app. Bend emits `core.mjs`; Vite bundles the Foldkit UI and the worker glue (ADR-027).
The JS lane runs sequentially, with no threads and no GPU. `!` calls run on the CPU.	`bend guide`: "The JavaScript target ignores all that and just runs sequentially."	Parallelism comes from several Web Workers, each loading the same `core.mjs`.
Library target for the browser (Wasm plus Web Workers) is an open proposal (#1352).	Issue list.	Do not plan on it. Keep the FFI surface small so a Wasm swap stays cheap later.

NUMBERS

Fact	Evidence	Consequence
Numeric types are `Nat`, `U32` and `F32`. There is **no F64, I32 or U64**. F64 and U64 are listed "SOON" in WONTFIX, and maintainers ask people not to build a soft-float.	`bend base --types`, issues #1120 and #1027.	All model math is in F32 (about 7.2 significant digits). Money is integer cents. See spec/03.
On the JS lane an F32 is a JS number rounded with `Math.fround` after **every** operation.	Generated code: `Math.fround(0.5 * _x_6)`.	Results are true IEEE single precision. Dekker/Kahan error-free transforms are valid; spec/03 uses Neumaier summation.
F32 has `add sub mul div mod pow atan2 neg abs sqrt exp log log2 log10 sin cos atan sinh cosh tanh floor ceil is_lt… show read bits`, plus `U32.to_f32` and `F32.to_u32`.	`bend base`.	There is no `erf` or `erfc`; we write our own normal CDF. F32 ops are axioms in the checker, so no proofs about float values.
`F32.to_u32` truncates toward zero and saturates at 0 for negatives (−2.9 → 0).	Tried.	Round explicitly. Never convert negative values through `to_u32`.
Superposition and sharing need HVM5, which does not exist yet. SUP values and lazy/optimal sharing (incremental DUP) are interaction-calculus runtime features. Released Bend 2 runs on BendRT (no interaction nets); a second use of a value is a real copy (`+x`, cost = size). The HVM5 kernel in `bend-runtime/hvm5` is a research prototype (sequential; `e2e_vault` / `e2e_cluster` fail on 2.0.25), not a usable runtime.	`&{1, 2}` is a parse error on 2.0.25; tutorial "Honest limits"; hvm5 v2 README + local test runs 2026-10-30.	No design, plan item or performance claim may rely on superposition, SupGen or lazy sharing. Search = explicit enumeration; A/B = run both.
`U32` arithmetic wraps mod 2^{32} (3 − 5 = 4294967294, then + 10 = 8).	Tried.	Signed cents can be two's complement on U32 with our own signed compare (spec/03).
F32 is exact for integers only up to 2^{24}: 16777217 → 16777216.	Tried.	**Never** store timestamps, volumes or share counts in F32. GME daily volume has exceeded 16.7M shares many times.
`Nat` crosses the FFI as a JS number up to 2^53 (2.0.39, `nat_in`). The 2.0.25 guide still says BigInt. The runtime caps Nat at 2^48 − 1.	minebender `ui.mjs`, WONTFIX.	Day counts and bar indexes are fine as Nat. Treat the BigInt-vs-number detail as version-pinned.

LANGUAGE RULES WITH DESIGN CONSEQUENCES

Rule	Consequence
Affine by default. `+x` lets Data be reused (refcounted on C, free on JS). Closures are affine (callable once). Only top-level defs and templates (`~f`) can be called many times.	**Strategies are data (an AST), not closures.** A per-bar signal can't be a captured closure that runs 6,000 times. Use an interpreter over a Data AST, or templates for fixed code paths.
Recursion must structurally decrease. No mutual recursion. Unbounded loops count down a `Nat` fuel. `@unsafe` opts out but loses proof guarantees.	Every loop is a fold over a list or a `Nat` count. Solvers (IV, fits) take fuel such as `40n` . Two-type interpreters become one def with a selector argument.
Tail calls become loops. **Non-tail recursion on the JS lane overflows the host stack** (List.length over ~200k elements; String.length is an intrinsic for this reason).	All long folds must be tail-recursive with an accumulator, then `List.reverse` if order matters. Treat anything over ~5k frames as long. Review rule in plan/04.
`match` works only on a parameter or a pattern-bound variable, never on a computed value. There is no `if` . Use `match b: case True{} or Bool.pick` .	Many small helper defs. Accept the verbosity and keep them in the same module.
No common-subexpression elimination.	Bind a value once and reuse it (`+x = …`).
Operators need a type annotation: `(a + b : F32)` . Nested calls need their own: `F32.tanh(u)` with `u` bound first.	Mechanical, but it causes most first-compile errors.
`Array<T>` is a `Type` (one owner), has power-of-two size, **wraps its index around**, and on JS is a plain JS `Array` mutated in place. Reads hand the array back (`Array<T> & T`). The `a[i]` sugar is `U32` -only; other element types use `Array.get` and `Array.set` . An array argument from JS is the caller's array, with no copy.	Raw columns enter as padded power-of-two arrays plus an explicit length `n` ; the shell pads them. Out-of-range indexes silently wrap, so every index helper must bounds-check against `n` . The sealed dataset is converted to Data (lists and records) so many calls can read it without threading ownership.
`Map` is string-keyed (`new set get has del keys`); `get` and `has` hand the map back.	Avoid it on hot paths. Use dense `U32` ids assigned at ingest.
No HTTP, TLS or DNS in Base (WONTFIX #846).	All network I/O stays in JS (page workers). Bend is the pure core, just like minebender.
Effects are JS/C host functions; only the event loop runs them.	The core exposes **no** IO. Pure defs only, so every core function is deterministic and testable.

FFI SHAPE (WHAT JS SEES)

- `import Core from "./core.mjs"` . The default export is an object. Defs of the entry file appear by bare name (`Core.run`), and defs from imported modules as `"module:def"` (for example `"game:step"` in minebender). Call only the entry file's defs.
- Constructors are `{$: "Name", field: value}` , lists are `{$:"Con", head, tail}` / `{$:"Nil"}` , pairs are `{$:"Tuple", fst, snd}` , `Maybe` is `{$:"Some", value}` / `{$:"None"}` , numbers are JS numbers, and arrays are JS arrays.
- Calls are wrapped by `run_lib(f, arity)` , which supports partial application and runs a trampoline (`run_loop`), so deep tail recursion is safe.
- Bend values are plain JS objects, so **structured clone works**. Workers can `postMessage` them and IndexedDB can store them. minebender's worker keeps the game as an opaque value and only reads it through `UI.*` calls. We do the same.

THE MINEBENDER PATTERN WE COPY

```
docs/index.html   hand-written: markup, CSS, input, drawing. Decides nothing about rules.
docs/worker.js    hand-written glue: holds opaque Bend values, converts lists <-> arrays, calls Core.*.
docs/ui.mjs       generated by `bend ui.bend -o docs/ui.mjs`; never hand-edited.
ui.bend           ~50 lines: the entire FFI surface; every rule lives in other .bend modules.
LAWS.bend         user-owned laws; PROOF.bend proves them; `bend PROOF.bend` is the gate.
```

Lessons from minebender that carry over:

1. Put heavy compute in a worker from day one. Their slow solver was the reason for the worker.

2. Make the click path trivial: immediate optimistic UI, small `postMessage` payloads, DOM diffing.
3. Ship `LICENSE-bend` next to the generated module, because the runtime code inside it is Apache-2.0.
4. Laws cost a lot. Theirs ran to about 12,900 lines of proof for about 1,330 lines of code. Our answer (ADR-032): prove as much as possible, but as many **small, per-layer** laws checked separately, not a few huge ones (spec/15 §7).

COMMANDS AN AGENT MUST RUN BEFORE WRITING BEND

```
export BEND_NO_TELEMETRY=1
bend version            # must print 2.0.39 or newer
bend guide              # whole guide; also: bend guide effects, bend guide shaders
bend base F32           # numeric API
bend base Array         # array API
bend base List          # list helpers (map/fold/reverse are templates)
```

C.34 · `bend-gme/version-1/.opencode/skills/bend2-effects-targets/references/04-minebender-notes.md`

minebender: notes

Running notes for the law-driven-development trial (see BRIEF.md for the questions to answer).

DECISIONS (CONFIRMED BY THE USER, 2026-10-25)

1. Flags are not knowledge. `consistent` and the knowledge ignore them. Flags do matter for actions: double-clicking (chording) a number reveals its hidden, unflagged neighbours.
2. The total mine count is knowledge: a consistent layout has exactly N mines.
3. An option is a hidden cell that is safe in every consistent layout. Provably mined cells are not moves.
4. The first click is always safe: mines are placed after it, avoiding that cell. Later (user's choice, after checking what later Windows versions did): when there is room (n + 9 <= w*h, true for every level and Custom cap) its neighbours are kept free too, so the first click always opens an area, as in Windows Vista and later. XP only guaranteed the cell itself (checked in the XPWinmine source: a first-click mine moves to the first free cell from the top-left).
5. Levels follow Windows XP (checked against a source restoration, github.com/12UE/XPWinmine): Beginner 9x9/10, Intermediate 16x16/40, Expert 30x16/99; Custom clamps width 9..30, height 9..25, mines 10..999, then caps mines at w*h - 10.
6. Chording only when flagged neighbours equal the number; clicking a flagged cell does nothing (both approved).
7. UI: leaning to a web page (`bend page.html -o dist` gives static HTML + plain JS with no Bend runtime needed, so it can be hosted on GitHub Pages; tested on a toy page).

APPROVED LAWS

Reviewed one at a time by the user; never weaken or delete these without their consent.

- 2026-10-25: level_beginner, level_intermediate, level_expert, level_custom, frozen, reveal_flagged, reveal_unflagged, flag_no_knowledge, flag_toggle, flag_others, flag_revealed
- 2026-10-25 definitions: at, mine, on_board, rows_wide, shaped, same, dist, within_one, adjacent, row_cells, cells.go, cells, touching, neighbours, count_mines, adj_mines, row_mines, total_mines, seen, is_hidden, hidden, revealed, agrees, agrees_all, consistent, ProvablySafe, Stuck, play, Reachable
- 2026-10-25: bridge, won_iff (+ defs is_won, all_safe_revealed.go, all_safe_revealed), monotone
- 2026-10-25 definitions: zero, zero_walk
- 2026-10-25: open_mine (now also shows Boom), open_safe (on-board premise dropped) (+ def is_lost), flood_complete (safe premise dropped), flood_exact, flags_hidden (user-proposed)
- 2026-10-25 definitions: flags_among, hidden_unflagged, chord_targets.pick, chord_targets.go, chord_targets (all_safe_revealed simplified afterwards, same meaning: passes +v on instead of rebuilding it)
- 2026-10-25: chord (+ def reveals), assist_stuck, assist_too_early, gen_shape, gen_count, gen_first_safe
- 2026-10-25 (later, user chose "option 1"): the assist action carries a random number, `Assist{k}` ; `assist_stuck` and `assist_too_early` gained `for k: Nat` and use `T.Assist{k}` . Same guarantees, for every k. The number picks which safe cell opens when stuck (frontier first, from a random point), which the user asked for; that preference itself is not a law.
- 2026-10-25 (after the UI went live): the display section: Tile type, tile, tile_of, is_covered, display_play, display_blast, display_hint, display_mines (approved as drafted; display_mines kept loose on purpose: every mine is visible after the game, not which XP tile it gets).

- 2026-10-25: gen_opening (first click opens an area when there is room); the generator now keeps a zone free (the first cell, or it and its neighbours), and its proofs gained a bound: at most 9 board cells are within one step of a cell.

CHOICES SETTLED DURING REVIEW

- The flood, the chord's own reveals and the assist open cells through `G.open`, which ignores flags; `flags_hidden` then makes the game clear any flag on a cell it reveals.
- Witness and stuck soundness of the solver are not laws: `assist_stuck` and `assist_too_early` state the button's behaviour directly against the brute-force spec, so the solver's own laws become lemmas in PROOF.bend (user had no preference; kept as proposed).
- Laws about play quantify over reachable games (any actions from a fresh game on a well-shaped layout). `frozen`, the flag laws (except `flags_hidden`), `reveal_*` and `chord` hold for every game state.
- No law about the randomness of generated boards (user agreed to skip it).
- `level_custom` covers the defs `clamp` and `custom`.

TRUST BOUNDARY (WHAT THE LAWS DO NOT COVER)

- The screen is `G.display(g)`, a grid of tiles; the display laws (end of LAWS.bend) tie it to the knowledge, flags and status. Not pinned by them: which tile a mine gets after the game (XP style in the code), and the crossed-out wrong flags after a loss. What is *not* proven is the JavaScript between that grid and the pixels: `docs/index.html` (draws the tile codes it is sent; turns clicks, taps and long-presses into Reveal/Chord/Flag/Assist) and `docs/worker.js` (holds the game, converts lists, feeds `crypto.getRandomValues` into `place_mines`). Neither decides anything about the rules; read them to check that. `ui.bend` is the whole interface they call. Also outside the laws: the timer, the mine counter (mines minus `UI.flags`, shown as 0 once won), the messages, and the statistics (localStorage, per board spec: five best times with dates, played/won/win %, current and longest streaks, Windows 7 style; a game abandoned after its first click counts as a loss, including a page closed mid-game, via a "pending" marker read on the next load).
- How the board is drawn on screen: on a portrait screen a board wider than tall is drawn turned a quarter turn (the player can rotate any non-square board, and zoom the cells from 16 to 48 px). Only the drawing and the mapping from a tapped cell back to the game's (x, y) change.
- Before the first click no game exists (mines are placed on the first reveal), so flags, chords and the assist do nothing until then; the board shown is `UI.idle`, Bend's display of an empty board.
- Generation laws cover shape, count and first-click safety, not the distribution.
- Which safe cell the assist reveals, and which provably safe cell an AssistTooEarly loss shows, are left free (the user does not mind, provided it is safe).

SPEC GAPS FOUND (Q2: A LAW PASSED WHILE BEHAVIOUR WAS WRONG)

No case of a law passing while observed behaviour was wrong. Found during review:

- (found in review, not yet in behaviour) "Games terminate" rests on monotone + bridge: an action that changes the knowledge must reveal a cell. On a board with zero rows, bridge cannot pin the knowledge's width (no row to measure), so the width field could change forever without a reveal. Harmless in practice: every level has at least 9 rows.

STATUS (2026-10-25)

All 30 approved laws (26 game laws and 4 display laws) are proven: `./check.sh` prints ALL PROOFS CHECK for `bend PROOF.bend` and for `--verdict` (the Lean-proven BendTT kernel; needs Lean v4.34.0, installed with elan v4.2.4 into `~/.elan`, checksum verified against the GitHub release). Self-play (`selfplay.bend`) wins every test game.

Assist speed, Expert self-play through the page's module in node (25 games, 5,451 calls): mean 20 ms, worst 267 ms. Before the solver rework: mean 185 ms, worst 12.8 s, ~1 s whenever stuck.

Web page: https://cleto.dev/minebender/ (`https://cleto.dev/minebender/`) (`docs/`, served by GitHub Pages; `./build.sh` regenerates `docs/ui.mjs` from `ui.bend` with `bend ui.bend -o docs/ui.mjs`). The upstream page bundler (`bend page.html -o dist`) was tried first, but it does not follow `new Worker(...)`, and the game runs in a worker so a slow assist can never freeze the page (that was seconds before the solver rework, now well under a second). So the page is three static files and no bundler.

Size: LAWS.bend ~660 lines (all approved) (user-owned); code ~1330 lines (types, board, game, solver); laws and proofs together ~12,900 lines across LAWS.bend, PROOF.bend and 16 `proof_\*.bend` lemma modules.

Q1: WHAT THE USER READ BEYOND LAWS.BEND

types.bend (the interface types), and nothing else: every definition and law in LAWS.bend was reviewed one by one (with plain-English and Bend for each). The code and proofs were not read; the gate is the checker. The user did need a Bend primer along the way (quantities `&1` / `&2` / `+`, templates, why loops become recursion over lists), and spotted several simplifications in the spec itself (`!=` instead of a helper, `not same in adjacent`, the `flags_hidden` law).

Later rounds kept the same pattern. The display laws were drafted and proven while the user was away, then reviewed one by one after the fact and approved unchanged; the Tile type was shown in that review. For the solver rework and the assist's random pick, the user read neither code nor proofs: they asked for the standard published method, approved the one law-text change (`Assist{k}`) in prose, and judged the result by the live page and the reported timings. Beyond LAWS.bend, their main check was playing the game (which is how the slow assist and the top-left pick were found: behaviour the laws allowed but the user did not want, rather than behaviour a law got wrong).

WHERE PROOFS GOT HARD (Q3)

- Rewrite direction (`%e`) was the most frequent friction: dozens of fixes. A fixed rule (`e : {new == old}`) helped; error messages show expected/observed terms, which made it fast.
- Function-typed hypotheses are affine (usable once, even erased), so "for every cell" invariants could not be passed around as functions. Every invariant became a Bool check over all cells plus a "read it back at one cell" lemma (equalities are copyable).
- No mutual recursion, and a proof helper cannot call the lemma it serves: induction hypotheses are passed into helpers as closures (`e2 => lemma(rest, ...)`).
- A law's `for x` without `+` cannot be reused in its proof, and a proof def cannot add `+`. The user's laws were left untouched: each `L.law` forwards to a restated lemma with `for +x`. (Correction to what I told the user during review: `for` lines *do* need `+` when a proof reuses the variable; the restated lemmas carry it.)
- `flood_exact` needs an actual path for every cell the flood reveals. The flood code was changed to carry a ghost path with each queued cell (used only by the proofs).
- The solver's completeness (no layout found ⇒ none exists) was the largest proof: pruning soundness needed grid sums = cell sums (generic templated lemma), inequality arithmetic, and completion/coverage invariants for the partial layout. The code was changed in one place for it: the to-do ordering used a pair (hard to take apart in proofs) and became two filters.
- The solver is proven against the brute-force definition of `ProvablySafe`. Its first version searched in reading order: easy to prove, but 1 s to 12.8 s on Expert. The rework follows the standard constraint approach to Minesweeper (connected components of the frontier, searched separately: Becerra, "Algorithmic Approaches to Playing Minesweeper", Harvard 2015, ch. 6; Tatham's Mines solver was read too, but it is heuristic and the assist law needs an exact answer). Two tricks kept the proof small:
  - The component order is a heuristic prefix put in front of the old, proven order. The search skips cells already assigned, so the proofs only need "every cell is still in the list" and "every entry is one of the solve's own": no proof that the components are right.
  - The search carries running counts instead of rescanning the grid; rather than re-prove the search, `proof_speed.bend` proves the counting search returns exactly what the old one returns (`dfs2_eq`), and the old lemmas do the rest. First prototypes were benchmarked unproven; one bad ordering ran for 10+ minutes and maxed the user's fan. Benchmarks now run under `nice` with a time limit.

BEND LESSONS

- `for x: A where P` turns `x` into a pair everywhere in the law; use a separate `for h: P` hypothesis.
- `match` only inspects parameters or pattern-bound variables; a computed value needs a helper def.
- Imported constructors are qualified like defs: `T.Pos{..}`, `case T.Pos{x, y}`.
- Termination reads arguments left to right; put the shrinking one first (`play(acts, g)`).
- A file importing a LAWS.bend with open laws will not run `main`. `examples.bend` checks the spec on concrete boards as `{==}` defs instead (`bend examples.bend --check-only`); a wrong value fails with expected/observed.
- A `+v` stays usable after `match v`: no need to rebuild a copyable value from its fields.
- `%e : P` with `e : {a == b}`: the goal must be P with b at `_`; it becomes P with a. Rule of thumb: `e : {new == old}`. Same for an inline `{%e : P; x : T}`: T is old (P with b), x has P with a. (use `Equal.sym` to go the other way).
- A refutation of `{1n+n == 0n}`: rewrite through `disc` (On to Empty, 1n+p to Unit), answer `Unit{}`.

C.35 · `bend-gme/version-1/.opencode/skills/bend2-performance/SKILL.md`

Bend 2: finding slow code, and why it is slow

`bend` has **no profiler, no trace flag, no counter**. What it has is a checker and an emitter. So the loop is always: *time it → bisect it → name the cost → fix one thing → re-time the same input*.

1. MEASURE BEFORE YOU THINK

```
export BEND_NO_TELEMETRY=1
bend x.bend -o /tmp/x.js                  # emit once (2–5 s for a tool)
time timeout 60 bun /tmp/x.js             # wall clock, and a hard stop
```

Read three numbers, not one:

observation	meaning
`user` ≈ `sys`, both large, RSS flat	CPU spinning — a recursion or an eager evaluation, not I/O
`sys` dominates	I/O: files, sockets, or page faults from huge allocation
RSS climbing to the cap	materialising something you never needed (a year of rows as a list)
finishes, but 10× the arithmetic	constant factors, JSON of the wrong shape, or `Word` rebuilds

Built-in signals, cheap to use:

- `node test-bend.mjs` prints **per-file seconds** and flags anything over `BEND_SLOW_TEST_S` (10 s) as `slow`. A test that grows past it is a design smell.
- `bend <file> --check-only` prints nothing about speed; its own 0.2–4 s is the checker walking the import closure. Small closures are the fast inner loop.
- **No incremental build**: every `bend` call re-checks the whole closure. If the loop feels slow, the fix is fewer imports, not a cache flag.

2. BISECT THE INPUT, NOT THE CODE

Halve the input twice before reading any code:

```
250 items -> >30 min, never returns
 60 items -> >20 min, never returns
  5 items -> >5 min,  never returns
  1 item  -> >2 min,  never returns      <-- the diagnosis
```

One item cannot take minutes for per-item work. So the cost is **per element inside one item**, or the item count is not what drives it at all. That single line eliminates the parser, the codec and the engine (you can prove that by timing the same work through `core.mjs`: 20 days in 34 ms). What is left is your plumbing.

Always keep a known-good reference: **measure `core.mjs` first**. It separates "the core is slow" from "my tool is slow" in one command.

3. WHEN YOU NEED SAMPLES

```
bun  --cpu-prof build/rank.js                     # writes a .cpuprofile in cwd
node --cpu-prof --cpu-prof-dir=/tmp/prof build/rank.js
bun  --inspect build/rank.js                      # then chrome://inspect
```

A `.cpuprofile` is JSON: `nodes[].callFrame` plus `hitCount`; sort by `hitCount` for self-time. A 0.8 s run gives ~16 samples at 1 ms — too few to read, so profile something that runs ≥ 10 s, or instrument with `performance.now()` in the effect JS around a loop.

The repo's own bench style (`packages/core/bench/*.bend`, `docs/05-benchmarks/`):

- loop the hot function **N times inside one compiled call**, not N process launches;

- take the timing in the JS that calls the module, never inside Bend;
- **divide by the step count the function returns**, not by the number you think it did (one published 4× error came from exactly that);
- report the **min of 7** warm runs, and note the load average.

4. THE COST MODEL (PREDICT IT BEFORE YOU RUN IT)

construct	cost	consequence
`Bool.pick(c, a, b)`	**both** `a` and `b` evaluated	a recursion in each arm is 2^n — see §5
`Bool.and(a, b)`	both evaluated (no short-circuit)	guard with `match` when the second arm is expensive
`match` on `U32` literals, fallback reads the scrutinee	rebuilds a 32-cell `Word`	13.9× on the VM's hot path; `build.mjs` greps for it (R-37)
non-tail recursion	grows the JS stack	overflows near 200k frames; fold with an accumulator, `List.reverse` at the end
`acc <> x` inside a loop	copies the left side	$O(n^2)$; cons and reverse once
`List.set` per element	walks the list	same, quadratic; build the list in the right order
closures	affine, callable once	a captured lambda cannot be a per-bar function; use templates or data
no CSE	repeated subexpression recomputed	bind once with `+x`

5. THE BUG THIS REPO HIT, END TO END

`tools/bend-blobs/rank.bend` filtered 250 file names:

```
Bool.pick(List<&2, String>, is_bgb(name),
  pick_go(rest, Con{name, acc}),     -- builds the rest
  pick_go(rest, acc))                -- builds it again
```

`Bool.pick` is an ordinary def: both arguments are evaluated before it returns one. Two recursive calls per element, each spawning two more — 2^{250}. It does not get slower; it never finishes. The same shape in `dump.bend` survived because its arms were `IO` values whose recursion sat inside a `do` continuation: built, not run.

Fix, one line, no semantics lost:

```
List.take(&2, String, List.filter(~String, ~(n => bgb_only(n)), names), max_days())
```

`List.filter` matches internally, so it recurses once. Measured: **>30 min → 0.8 s** for 250 days, identical output.

The second factor was separate: `File.read_bytes` returns one `U32` per **byte** (26,620 cells per day) and then a byte→word walk in Bend; `read_words` packs four bytes per word inside the effect (6,655 cells, no walk). Fixing the constant factor alone would still have hung — the order matters.

6. CHECKLIST

1. Time it with `timeout`. Never run an unknown Bend program without one.
2. Halve the input twice. Flat time = not the loop you think.
3. Compare against `core.mjs` on the same data. One of the two is innocent.
4. Name the cost from the table in §4 before editing.
5. Fix **one** thing, re-time the **same** input, and keep the output byte-identical (`cmp` the result file) — a speedup that changes the answer is a different bug.
6. Lock it with a test that would have caught it: the arithmetic (n × unit), a golden file, or a law. This repo's `api_blob` and `blob-parity` tests came out of this loop.

WHERE THE NUMBERS ALREADY ARE

- `docs/05-benchmarks/01-ffi-and-array.md` — boundary shapes, text vs lists.
- `docs/05-benchmarks/02-vm-interpreter.md` — the 9.35 → 0.67 µs/frame work, the JIT/GC/AOT measurements, the native reversal.
- `packages/core/bench/vm.bend` — the benchmark harness to copy.
- `packages/core/build.mjs` — the static guard (Word rebuilds) and the size budget.

C.36 · `bend-gme/version-1/.opencode/skills/bend2-app/SKILL.md`

Building a complex app in Bend 2

Make the core pure, small at the boundary, and layered; make user programs data; prove the few laws that buy trust; measure the rest.

MODEL

```
packages/core/bend/
  app.bend        FFI surface: packing only, api_version()
  view/  val/  gen/  bt/  lang/  chain/  ind/  pricing/  data/  store/  sync/  time/  num/
  LAWS.bend  PROOF.bend  tests/*.bend
```

Imports go only downward (`num` at the bottom, `view` at the top); no cycles. The worked example is this repo: `../../../docs/03-spec/01-system-architecture.md §3`.

RULES

1. **Pure core.** No I/O in Bend. JS (workers) fetches, lexes, stores and draws; Bend validates, decides and builds view models.
2. **One boundary file.** `app.bend` packs and unpacks; every rule lives below it (`../../../docs/03-spec/12-ffi-and-workers.md`).
3. **Exact by construction.** Money in signed cents, time as day/minute integers, persisted data as U32 words; F32 only behind `Real` (`../../../docs/03-spec/03-numerics-and-time.md`).
4. **Single tagged types** instead of mutually recursive ones (Bend has no mutual recursion).
5. **User programs are data.** Strategies are an AST built by constructors, checked by one checker, run by one VM, so causality and risk laws are proven once for all programs (`../../../docs/03-spec/07-strategy-language.md`).
6. **Deterministic encodings** for anything persisted: same input → same bytes → same content hash (`../../../docs/03-spec/19-blob-store.md`).
7. **As many laws as possible**: every integer/ADT invariant of every layer is a law; property tests only where laws don't fit (strings, F32, host) (`../../../docs/03-spec/15-testing-and-laws.md` §7). 7b. **Every layer is testable alone**: `bend/<layer>/LAWS\*.bend` (a law file checks itself) and `tests/` . Fast loop, selected by the import graph, never the whole suite: module `--check-only` → `check:changed` → `proof:changed` → `test:bend:changed` (compiled, cached). The direct-lane parity run (`test:bend:direct`) and the full gates run in the background (`research-to-release` "Fast iteration").
8. **Lane parity**: the same fixtures through native and JS give equal outputs. **Golden vectors** come from independent references (Rust, QuantLib), never from the code under test.
9. **Spike before budgets**: measure the hot paths (decode, fold, pricing) on the real target before promising numbers.
10. Tail-recursive folds; chunk long work; workers for anything over a few ms.
11. No design may depend on superposition, SupGen or lazy sharing: those need HVM5, which does not exist yet (ADR-031).

FIRST FIVE FILES

1. `packages/core/BEND_VERSION` (pin) and the build script emitting `dist/core.mjs` .
2. `bend/num/real.bend` (F32 facade) and `bend/num/money.bend` (signed cents on U32).
3. `bend/time/day.bend` (trading days, calendar).
4. `bend/app.bend` with `api_version()` and one real call.
5. `bend/LAWS.bend` + `bend/PROOF.bend` with one approved law, so the gate exists from day one.

READING A REAL BEND REPO (REFERENCES/02-READING-REAL-REPOS.MD)

Find: the entry point (`main` / FFI file), `LAWS.bend` (what is claimed), `PROOF.bend` (what is proved), tests (what is only measured), and the effect boundary (where I/O happens). Say what it proves and what it only measures.

PATTERNS

- Capstone, a checked command parser with constructors, `Maybe` , two proved laws and CLI args: `references/01-capstone.md` .
- minebender (MIT, Luís Cleto): `ui.bend` FFI + worker + laws; notes in `../bend2-effects-targets/references/04-minebender-notes.md` .

PITFALLS

Symptom	Cause	Fix
import cycle / mutual recursion error	layering violated	move the shared type down a layer; one tagged type
UI and core disagree on a rule	logic duplicated in TypeScript	delete the TS copy; call the core
results differ between runs	clock, randomness or iteration order leaked in	pass `now` /seed in; sort with an explicit order
a proof gets huge	the law states too much at once	split it into small laws; move string/F32 parts to property tests

VERIFY

```
BEND_NO_TELEMETRY=1 bend packages/core/bend/app.bend --check-only
bun run --cwd packages/core proof:changed && bun run --cwd packages/core test:bend:changed   # the loop
bun run --cwd packages/core proof > /tmp/proof.log 2>&1 &                                   # the gate, background
bun run check && bun run test                                                               # before the commit
```

REFERENCES

File	Covers	Open when
`references/01-capstone.md`	a complete checked program with laws and args	a template for a module + laws
`references/02-reading-real-repos.md`	curated real Bend repos, which files to open, patterns to copy	studying prior art
`../../../docs/03-spec/01-system-architecture.md` , `spec/12` , `spec/07` , `spec/19` , `spec/15`	this app's architecture, FFI, AST+VM, blob codec, tests	working on bend-gme
`../../../docs/02-plan/04-agent-playbook.md`	implementing-agent rules and DoD	before writing code here

RELATED

`bend2` · `bend2-effects-targets` · `bend2-proofs` · `bend2-runtime` · `research-to-release` .

C.37· bend-gme/version-1/.opencode/skills/bend2-app/references/01-capstone.md

--- title: "Bend 2 tutorial 15: capstone, a checked command parser" description: "One small program built from the tutorial's parts: types, a codec, a transition function, laws that the gate proves, and a command-line entry point that the laws cover." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/15-capstone/ sources:

- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

15: capstone, a checked command parser

This chapter builds one small program from the parts of the tutorial. It reads commands from the command line, moves a job through states, and proves properties of the transitions. The laws are the specification, and the gate checks that the code meets them.

The project has three files in one directory:

- `job.bend` : the program. Pure code comes first: types, a codec, and the transition function. The IO shell `main` comes last.
- `LAWS.bend` : the claims. A person writes them, and nobody edits them to make a proof pass (chapter 07).
- `PROOF.bend` : one proof per law. `bend PROOF.bend` is the gate.

1. THE THREE FILES

job.bend

```
import Base

type Cmd is Data:
  Start{}
  Cancel{}
  Finish{}

type State is Data:
  Queued{}
  Running{}
  Cancelled{}
  Finished{}

def encode(c: Cmd) -> String:
  match c:
    case Start{}:
      "start"
    case Cancel{}:
      "cancel"
    case Finish{}:
      "finish"

def only(hit: Bool, c: Cmd) -> Maybe<&2, Cmd>:
  match hit:
    case True{}:
      Some{c}
    case False{}:
      None{}

def decode(+s: String) -> Maybe<&2, Cmd>:
  Maybe.or(&2, Cmd, only(String.eq(s, "start"), Start{}), Maybe.or(&2, Cmd, only(String.eq(s, "cancel"), Cancel{}), only(String.eq(s, "finish"),
Finish{})))

def step(+s: State, c: Cmd) -> State:
  match s c:
    case Queued{} Start{}:
      Running{}
    case Queued{} Cancel{}:
      Cancelled{}
    case Running{} Cancel{}:
      Cancelled{}
    case Running{} Finish{}:
      Finished{}
    case _ _:
      s

def state_name(s: State) -> String:
  match s:
    case Queued{}:
      "queued"
    case Running{}:
      "running"
    case Cancelled{}:
      "cancelled"
    case Finished{}:
      "finished"

def run1(m: Maybe<&2, Cmd>) -> IO(Unit):
  match m:
    case Some{c}:
      IO.print(state_name(step(Queued{}, c)))
    case None{}:
      IO.die(Unit, 2, "unknown command")

def run2(m: Maybe<&2, Cmd>, n: Maybe<&2, Cmd>) -> IO(Unit):
  match m n:
    case Some{c} Some{d}:
      IO.print(state_name(step(step(Queued{}, c), d)))
    case _ _:
      IO.die(Unit, 2, "unknown command")

def dispatch(args: List<String>) -> IO(Unit):
  match args:
    case Nil{}:
      IO.print("usage: job CMD [CMD]")
    case Con{a, Nil{}}:
      run1(decode(a))
    case Con{a, Con{b, Nil{}}}:
      run2(decode(a), decode(b))
    case _:
      IO.die(Unit, 2, "too many commands")
```

```
def main() -> IO(Unit):
  do IO<Unit>:
    args : List<String> <- IO.args()
    dispatch(args)
```

Read the file from the top. `encode` and `decode` are inverses: `encode` writes a command as a string, and `decode` reads it back and returns a `Maybe`. `step` is the transition function, and every pair it does not list leaves the state unchanged. `dispatch` takes one or two commands, and `main` passes it the arguments. Each helper is defined above the function that uses it, so the file has no forward references (chapter 02).

LAWS.bend

```
import Base
import ./job.bend as J

law round_trip:
  for c: J.Cmd
  {J.decode(J.encode(c)) == Some{c} : Maybe<&2, J.Cmd>}

law finished_stays:
  for c: J.Cmd
  {J.step(J.Finished{}, c) == J.Finished{} : J.State}

law start_then_cancel:
  {J.step(J.step(J.Queued{}, J.Start{}), J.Cancel{}) == J.Cancelled{} : J.State}
```

`round_trip` says that every command survives encoding and decoding. `finished_stays` says that a finished job stays finished, whatever command arrives. `start_then_cancel` is a concrete claim that needs no case split: starting and then cancelling a job cancels it.

PROOF.bend

```
import Base
import ./job.bend as J
import ./LAWS.bend as Laws

def Laws.round_trip(c):
  match c:
    case J.Start{}:
      {==}
    case J.Cancel{}:
      {==}
    case J.Finish{}:
      {==}

def Laws.finished_stays(c):
  match c:
    case J.Start{}:
      {==}
    case J.Cancel{}:
      {==}
    case J.Finish{}:
      {==}

def Laws.start_then_cancel():
  {==}
```

```
All terms check.
```

`PROOF.bend` imports `job.bend`, so the gate also type-checks the program's `main`. Every proof here is a case split with `{==}` in each case, because each case computes to the same term on both sides.

4. RUN IT

The program is a command-line tool. These are the results of running it:

```
bend job.bend start
bend job.bend start cancel
bend job.bend start finish
bend job.bend
bend job.bend bogus
bend job.bend start bogus cancel
```

```
running
cancelled
finished
usage: job CMD [CMD]
unknown command        (exit status 2)
too many commands      (exit status 2)
```

5. BREAK IT ON PURPOSE

Each exercise has an answer that was checked by running the gate.

1. **A broken encoder.** Change the `"cancel"` string in `encode` to `"start"`. Predict the gate's first message.

   Answer: the gate fails in `LAWS.round_trip`:

   ```
   - expected : Some{job.Start{}}
   - observed : Some{job.Cancel{}}
   ```

   `round_trip` catches the bug because the claim names the behaviour that broke.

2. **A gap the laws do not cover.** Restore the encoder. Change `step` so that `Queued` plus `Cancel` stays `Queued`. Run the gate.

   Answer: the gate still prints `All terms check.` No law mentions that pair, so the broken transition passes. The laws are the specification, and the gate proves only what they say.

3. **Close the gap.** Add this law to `LAWS.bend` and this proof to `PROOF.bend`:

   ```
   law cancel_queued:
     {J.step(J.Queued{}, J.Cancel{}) == J.Cancelled{} : J.State}

   def Laws.cancel_queued():
     {==}
   ```

   Answer: the gate fails until `step` is fixed. It reports `expected : job.Queued{}` and `observed : job.Cancelled{}` at `LAWS.cancel_queued`. Restore `Cancelled{}` in `step`, and the gate passes again.

WHAT TO TAKE AWAY

A law names a behaviour, and the gate checks it. Passing the gate says nothing about behaviour that no law names. Write the claim first, keep it fixed, and let the proof change to match the code, not the reverse (chapter 07).

NEXT

Chapter 16 maps the repositories that assemble these pieces into real programs, and says which files to read first.

Continue to chapter 16.

C.38 · bend-gme/version-1/.opencode/skills/bend2-app/references/02-reading-real-repos.md

--- title: "Bend 2 tutorial 16: reading real Bend repositories" description: "A curated map of Bend 2 repositories to read after the tutorial: small verified examples, proved libraries, games, compilers, and large applications, with the files that show how each one is composed." updated: 2026-10-20 verified: 2026-10-20 status: docs-only canonical: /tutorial/16-references/ sources:

- https://github.com/bendlang/bend (https://github.com/bendlang/bend)
- https://github.com/AdrielSantana/metal-bending (https://github.com/AdrielSantana/metal-bending)
- https://adrielsantana.github.io/metal-bending/ (https://adrielsantana.github.io/metal-bending/)
- https://github.com/aivv73/bend-voxel (https://github.com/aivv73/bend-voxel)
- https://github.com/costamatheus97/bend-craft (https://github.com/costamatheus97/bend-craft)

- https://github.com/eliesgalvira/bendoom (https://github.com/eliesgalvira/bendoom)
- https://github.com/gildrb/elpis-fast (https://github.com/gildrb/elpis-fast)
- https://github.com/Giulio2002/bend-collections (https://github.com/Giulio2002/bend-collections)
- https://github.com/i2cjak/Backplane (https://github.com/i2cjak/Backplane)
- https://github.com/ind-igo/bend-frontend (https://github.com/ind-igo/bend-frontend)
- https://github.com/lilalittle/bend-packages (https://github.com/lilalittle/bend-packages)
- https://github.com/Lulzx/bendc (https://github.com/Lulzx/bendc)
- https://github.com/nuxyel/bend-ml (https://github.com/nuxyel/bend-ml)
- https://github.com/paymog/bend-kit (https://github.com/paymog/bend-kit)
- https://github.com/angerman/birc (https://github.com/angerman/birc)
- https://github.com/FabianVegaA/mylsm (https://github.com/FabianVegaA/mylsm)
- https://github.com/bendlib/bendlib (https://github.com/bendlib/bendlib)
- https://github.com/777genius/awesome-bend (https://github.com/777genius/awesome-bend)
- https://hub.bend-lang.com (https://hub.bend-lang.com)
- https://github.com/luiscleto/minebender (https://github.com/luiscleto/minebender) ---

16: reading real Bend repositories

The tutorial teaches one idea per page. A repository teaches the next thing: how the pieces are arranged when the program is larger than one screen. The useful question is not which repository is biggest. It is where the boundary between pure code, effects, proofs, and measurement sits, and whether that boundary is visible in the file tree.

This chapter maps repositories worth reading. The list is not a ranking and it is not official. Each entry says what the repository demonstrates and which files to open first.

Core insight. A Bend repository is readable when its claims, its pure code, its effects, and its evidence live in separate places. Most of the repositories below differ in size, not in that shape.

THE SHAPE OF A BEND REPOSITORY

A small repository usually looks like this:

```
repo/
  README.md          the pinned compiler, how to run it, what it proves
  LAWS.bend          the claims, written by a human
  PROOF.bend         one same-named def per law
  main.bend          the entry point
  src/               pure domain modules
  tests/             golden files, unit tests, differential tests
  bench/             measurements, with the machine and version recorded
  host/ or *.c/*.js  foreign effects, kept at the edge
```

Larger repositories repeat the pattern per package or per layer:

```
repo/
  package-a/
    LAWS.bend
    PROOF.bend
    check.bend       runtime checks for the parts the checker cannot reach
    VERSION          the hub version, independent of its neighbours
    package-a.bend
    bench/
  package-b/
    ...
```

Path	What it holds	Tutorial
`LAWS.bend`	`law` declarations: the specification	07
`PROOF.bend`	`def` with the same name as each law	07
`check.bend`	ordinary tests, often for native arrays or effects	09
`src/`	pure functions, one concern per module	02

Path	What it holds	Tutorial
`tests/`	golden output and differential comparisons	01
`bench/`	timing, memory, and throughput runs	05
`*.c` / `*.js`	foreign effects behind a named signature	09

Not every repository has every row. A package with no effects has no `host/`. A teaching example may put everything in one file. What matters is that you can tell which row a file belongs to.

HOW TO READ ONE

Read in this order, and stop when you can predict the next file:

1. **README.** Find the compiler version. Bend 2 changes between releases, and a repository that pins 2.0.37 may not check on 2.0.25. Find the command that runs the proof gate.
2. **`LAWS.bend`.** Read the claims before the implementation. They tell you what the author thinks the program means.
3. **`PROOF.bend`.** Look at the shape of a proof, not every line: `{==}`, a rewrite, an induction hypothesis, or a call to a lemma.
4. **The entry point.** Find `main`, the CLI, or the test driver. Trace one path from it into one leaf module.
5. **One leaf module and its test.** This is where the writing style lives: how types are annotated, how matches are ordered, how effects are avoided.
6. **The build and the benchmark.** Check whether the numbers name the compiler, the thread count, and the machine. A number without those is an anecdote.

Useful questions to carry from repository to repository:

- Does `PROOF.bend` import the adjacent `LAWS.bend`? Without that import, the laws you read are not the ones the proof file proves.
- Is `check.bend` testing the same behaviour as the laws, or something the checker cannot express?
- Are foreign effects behind a small named module, or scattered through the domain code?
- Does the benchmark compare against another implementation, or only against itself?
- Does the README say what the repository does **not** prove?

A SHORT READING LIST

If you want to see	Read
the smallest complete laws-and-proofs project	metal-bending
a proved, benchmarked library	bend-collections
the boundary between a checker and a backend	bend-frontend
a pure game core with a thin driver	bend-craft
a large, deeply modular application	backplane
performance work with a proof obligation	elpis-fast
the package ecosystem	bend-packages

THE COMMUNITY REPOSITORIES

METAL-BENDING: THE SMALLEST COMPLETE PROJECT

Repository: AdrielSantana/metal-bending `(https://github.com/AdrielSantana/metal-bending)` · **Browser walkthrough:** adrielsantana.github.io/metal-bending `(https://adrielsantana.github.io/metal-bending/)`

A playground on Apple Silicon with seven single-concept examples, six graphics programs, and a working proof gate. The README is in Portuguese; the code is the common language.

Read it for: the smallest repository that still has the full shape. Each example is one file, and each file is one idea: `01_hello.bend` for `do`, `02_types.bend` for `type ... is Data`, `03_recursion.bend` for terminating recursion, `04_parallel.bend` for the fork shape, `05_arrays.bend` for one-owner arrays, `06_io.bend` for `IO.fork` and channels, `07_monads.bend` for `do` over `Maybe`.

Start at: `examples/02_types.bend`, then `LAWS.bend`. The file opens with `add_zero` and `append_nil` from Base, then states `total_cons` about the repository's own `src/math.bend`. That is the progression the tutorial describes: prove a Base fact, then prove a fact about your code. `gfx/01_mandelbrot.bend` is the same language doing a GPU-shaped loop in a window.

Composition lesson: one concept per file, one claim per law, and a gate the README shows failing as well as passing. The `gfx/` directory is separate from `examples/` because rendering is a different concern with a different driver.

BEND-VOXEL: AN ENGINE, AN APP, AND ACCEPTANCE SCRIPTS

Repository: aivv73/bend-voxel `(https://github.com/aivv73/bend-voxel)`

A voxel engine with destructible environments and a native window. The repository separates the editable body (`voxel.bend`), the renderer (`render.bend`), lighting (`lighting.bend`), materials (`material.bend`), and the DDA traversal (`dda.bend`) from the application shell (`atelier.bend`, `main.bend`).

Read it for: acceptance checks written in Bend rather than in a separate test framework. `scripts/reference_check.bend`, `scripts/lighting_check.bend`, `scripts/material_check.bend`, `scripts/image_check.bend`, and `scripts/ui_check.bend` are programs that compare complete outputs: cell sets, sun-depth texels, material exports, and CPU/GPU images. The `docs/` directory keeps the evidence, including the raw TSV and log files behind the DDA comparison.

Start at: `LAWS.bend` for the material and geometry invariants, then `voxel.bend` and `dda.bend`. Then read `scripts/reference_check.bend` to see how a test is just another Bend program with an exit condition.

Composition lesson: the engine is a library of pure modules, and every executable is a thin driver. The `Makefile` exposes `make test`, `make geometry`, `make lighting`, `make materials`, and `make images` so each layer of evidence can be run alone.

BEND-CRAFT: A DETERMINISTIC GAME CORE

Repository: costamatheus97/bend-craft `(https://github.com/costamatheus97/bend-craft)`

A block world with terrain, a first-person player, and a per-pixel voxel ray tracer. The whole game is U32 fixed point, so a run is a function of its seed and input log, and every frame is the same bytes on JavaScript, native C at one or many threads, and the GPU.

Read it for: the cleanest small example of the pure-core pattern. `world.bend` is the world as one flat `Array<U32>` and its terrain generator. `render.bend` is the Amanatides-Woo DDA and four shading tiers. `game.bend` is a pure 60 Hz `step` plus `render(state)`. `play.bend` is the only game file that reads a clock, a window, or a device; the benchmarks read them too.

Start at: the README's file list, then `game.bend` and `play.bend` side by side. The gap between them is the architecture. `tests/world.bend`, `tests/render.bend`, and `tests/game.bend` test the pure side; `run_tests.sh` runs them. `bench/` holds the scripts that produced the README's numbers, and `LOG.md` and `OPTIMIZATIONS.md` record what was tried and rejected.

Composition lesson: determinism is a design constraint, not a test added later. Keeping time and I/O in `play.bend` is what lets the same `step` run in a benchmark, a replay, and the window.

BENDOOM: A LARGE PORT, DECOMPOSED

Repository: eliesgalvira/bendoom `(https://github.com/eliesgalvira/bendoom)`

Doom's first level, E1M1, written in Bend 2. The README carries a "SLOP WARNING" and says there was no adult supervision of the code. Treat that as a warning about style, not about usefulness: the decomposition is the lesson.

Read it for: how to port a large C program without turning it into one file. `src/` has one module per concern: `wad.bend` for the archive, `bsp.bend` for the level tree, `render.bend`, `sim.bend`, `things.bend`, `sprites.bend`, `sfx.bend`, `opl.bend`, and `music.bend`. `doom.bend` is the game driver; `film.bend` replays a demo to video; `tests/` mirrors the module list with one test file per subject.

Start at: `LAWS.bend`, which states byte, WAD, key, sound, and simulation invariants, then `src/bytes.bend` and `src/sim.bend`. Read `nix/proof.nix` and `nix/tests.nix` to see how the proof gate and the test matrix are wired into CI.

Composition lesson: a port is a set of named modules plus a reference oracle. `tools/oracle.nu` compares frames against Chocolate Doom, so the port has an external definition of correct, not just an internal one.

ELPIS-FAST: SPEC, LAWS, PROOF, DIFF

Repository: gildrb/elpis-fast (https://github.com/gildrb/elpis-fast)

A 4-bit quantized Qwen inference engine with decode and prefill kernels, measured on one RTX 3090. It is dense and specialized; read it for the unit of work, not for a general application layout.

Read it for: the spec/laws/proof/diff discipline around each optimized kernel. A kernel such as `attn_chunk` has `attn_chunk.bend`, `attn_chunk_spec.bend`, `attn_chunk_laws.bend`, `attn_chunk_proof.bend`, and `attn_chunk_diff.py`. The `.bend` file is the fast implementation, `_spec` is the reference definition, `_laws` states their relationship, `_proof` proves it, and `_diff` compares against a trusted implementation. The `*_TABLE.bend` files are decision tables for scheduling and attention shapes.

Start at: `LAWS.bend` and `PROOF.bend` at the root, then one kernel family from the README table. The repository's "What fast means" section is as important as the code: a speedup is kept only if the output is unchanged or passes a named quality gate, and each gate records thresholds set before the runs.

Composition lesson: performance work is not exempt from specification. The spec stays readable, the optimized kernel stays fast, and the proof plus the differential test connect them.

BEND-COLLECTIONS: A LIBRARY WITH SPECS AND PROOFS

Repository: Giulio2002/bend-collections (https://github.com/Giulio2002/bend-collections)

Containers and cryptographic primitives in stock Bend, benchmarked against optimized C implementations of the same algorithms. The README has a table of containers and another for SHA-256, Keccak, BLAKE, AES-GCM, Argon2id, and the other crypto modules.

Read it for: how a real library separates the public module, the specification, and the proof. `src/containers/` holds the implementations (`dynamic_array.bend`, `deque.bend`, `balanced_search_tree.bend`, `hash_table.bend`, `lru.bend`). `spec/` holds reference definitions, `proofs/` holds the proof modules, and the root `laws.bend`, `laws_containers.bend`, `laws_crypto.bend`, and `laws_math.bend` group the claims by area. `benchmarks/bend/` runs the same workloads as the native C baselines.

Start at: `src/containers/deque.bend` and its proof under `proofs/containers/`, then `laws.bend`. The cryptographic modules show the same shape at a larger scale: a specification, a proved equality, and published test vectors.

Composition lesson: a container is not "done" when it works. It is done when its invariants have a specification, a proof, and a measurement against the implementation it replaces.

BACKPLANE: A LARGE APPLICATION IN FLAT MODULES

Repository: i2cjak/Backplane (https://github.com/i2cjak/Backplane)

A large workspace for agentic hardware development, with deterministic KiCad and STEP viewers. It is a real product with a release process, not a sample.

Read it for: how a large Bend codebase stays navigable. `src/core/` is a flat set of roughly 140 small modules named after one concept each: `json.bend`, `codec.bend`, `semver.bend`, `sha256.bend`, `schematic.bend`, `pcb2d.bend`, `pcb3d.bend`, `model.bend`, `store.bend`. `src/app/` holds the view layer: `main.bend`, `layout.bend`, `render.bend`, `viewer.bend`, `theme.bend`, and one module per surface.

Start at: `src/app/main.bend` to see the entry point, then `src/core/json.bend` or `src/core/codec.bend` as a representative leaf. `LAWS.bend` and `PROOF.bend` sit at the root, so the proof gate covers the whole product with one command.

Composition lesson: flat names beat deep hierarchies when the names are specific. `src/core` is a vocabulary; `src/app` is a presentation of that vocabulary. Neither imports sideways for convenience.

BEND-FRONTEND: A CHECKER WITH A TYPED OUTPUT

Repository: ind-igo/bend-frontend (https://github.com/ind-igo/bend-frontend)

A composable frontend for full upstream Bend 2. It checks a program once and returns a typed, serializable `Core.Program` for a backend.

Read it for: the boundary between a language and its consumers. `src/core.bend` is the authoritative schema for the output. `src/decode.bend` turns upstream terms, patterns, declarations, and templates into that schema and rejects proof holes and missing `LAWS` imports. `src/frontend.bend` exposes `Frontend.check` and `Frontend.or_die`.

`src/wire.bend` handles metadata numbers. `src/cli.bend` is a small driver. `host/adapter.js` and `host/run.js` attach the checker and I/O to the Bend process.

Start at: `src/core.bend`, then `src/decode.bend`, then `src/LAWS.bend`. The laws are about the output: `bare_idempotent`, `missing_laws_rejected`, `complete_accepted`, and `metadata_overflow_rejected`. That is a useful example of laws stated at an API boundary rather than about arithmetic.

Composition lesson: the IR is a schema, decoding is a pure step, the host adapter is the only JavaScript in the runtime path, and every backend consumes the same serializable value. `ind-igo/bend-evm` is a separate repository that does exactly that.

BEND-PACKAGES: A MAP OF THE ECOSYSTEM

Repository: lilalittle/bend-packages `(https://github.com/lilalittle/bend-packages)`

A curated, regularly refreshed overview of BendHub `(https://hub.bend-lang.com)`: top packages, source repositories, publishers, and an A–Z list. The snapshot in this chapter's sources lists 703 packages on 2026-10-19.

Read it for: finding the next repository. The README groups picks by purpose: verified math and proofs, tensors and autodiff, web and networking, tools, games. Each row names the hub package, the version, the source repository when one is public, and the import line.

Start at: the "Top picks" tables, then `ALL_PACKAGES.md` when you know the name you want. The README also explains the two import forms: a name and version such as `bend-mathlib@0.7.3.0/nat.bend`, which resolves to a content hash, and the raw hash form, which always resolves to the same code.

Composition lesson: packages are the unit of reuse. A repository that publishes several packages, such as `bend-kit` or `bend-ml`, shows how to keep each one independently versioned and independently proved.

EIGHT MORE REPOSITORIES

These are the repositories I would open next. They are grouped by what they teach, not by stars.

BENDLANG/BEND: THE OFFICIAL SOURCE

Repository: bendlang/bend `(https://github.com/bendlang/bend)`

The compiler, the `Base` library, the guide, the official demos, and the test suite. This is the authority the rest of the tutorial cites.

Read it for: `guide/GUIDE.md`, which `bend guide` prints; `bend2/base.bend`, which `bend base` prints; `demos/` for complete programs; and `tests/` for the exact syntax and error behavior the checker accepts. Read the official tests when a tutorial chapter and a community repository disagree.

Start at: `guide/GUIDE.md`, then `bend2/base.bend`, then one demo and the tests for the feature you are using. Do not copy from `Base` without checking the version: the standard library changes between 2.0.x releases.

LULZX/BENDC: A SELF-HOSTING COMPILER

Repository: Lulzx/bendc `(https://github.com/Lulzx/bendc)`

A compiler for Bend 2, written in Bend 2, that compiles itself and reproduces its own output byte for byte. The README reports a bootstrap fixpoint and 80 tests passing under both stages.

Read it for: how to structure a compiler as passes. `bendc.bend` is the driver, `check.bend` the type checker, `core.bend` the shared representation, and `boot.bend` the bootstrap path. Each optimization or transformation has a matching proof file: `LIT.bend` with `LITPROOF.bend`, `SUBST.bend` with `SUBSTPROOF.bend`, `INLINE.bend` with `INLINEPROOF.bend`, `DEAD.bend` with `DEADPROOF.bend`. `rt/` is the runtime, split by concern (`bendrt.h`, `native.c`, `cc.c`, `gpu.h`, `hub.c`).

Start at: `bootstrap.sh`, then `bendc.bend` and `check.bend`. Then read one small pass and its proof together. `tests/` pairs its test programs with `.out` golden files, and `tests/check/` is a numbered suite of checker errors.

Composition lesson: a compiler pass is a module with an input language, an output language, and a proof obligation. The proof file next to the pass makes that obligation visible.

PAYMOG/BEND-KIT: PACKAGES WITH THEIR OWN GATES

Repository: paymog/bend-kit `(https://github.com/paymog/bend-kit)`

A general-purpose library for Bend 2, split into packages such as `bytes`, `bignum`, `collections`, `concurrency`, `crypto`, `json`, `http`, and `cache`. Each package is published separately and has its own version.

Read it for: the package layout. The mature package directories contain `LAWS.bend`, `PROOF.bend`, `check.bend`, `VERSION`, the implementation (`bytes.bend`), and a `bench/` directory. `bytes/check.bend` is especially useful: it says in a comment that the laws run in the checker, where `Array` is a tree of defs, while `check.bend` runs the same operations on the runtime's native array at sizes the checker would not reach. That is a precise statement of what each gate covers.

Start at: `bytes/LAWS.bend`, `bytes/PROOF.bend`, and `bytes/check.bend`, then the README's "Same type" section. It explains why a caller must import the same package version the callee imports: a different version is a different type.

Composition lesson: one package equals one name, one version, one proof gate, and one benchmark. Dependencies are pinned by name and hash, not floating.

NUXYEL/BEND-ML: DEPENDENT TYPES IN A LIBRARY

Repository: nuxyel/bend-ml `(https://github.com/nuxyel/bend-ml)`

Machine learning in Bend 2 with tensor shapes in the type. A mismatched matrix product does not compile; a `reshape` needs a proof that the element count is preserved. Six packages are published on BendHub, with an MNIST demo and a GPT-2 demo.

Read it for: how dependent types shape an API. `tensor/` uses lists with `Vec<n>` and `Mat<r,c>`; `tensor-array/` uses a flat `Array<F32>` with the same type-level sizes and a `Bands<r,c>` type for parallel matrix-vector work; `autograd/` has a typed backward pass; `bpe/` proves the tokenizer round trip; `binary-read/` keeps foreign code out of the proved packages. `demos/gpt2` and `demos/mnist` are the integration tests.

Start at: the README's two shape-error examples, then `tensor-array/main.bend`, then `autograd/main.bend`. Read the package README before the code: it names the package version, its laws, and its dependencies.

Composition lesson: put the invariant in the type when the type checker can carry it, and in a law when it cannot. The shape is checked at the boundary, and every layer after that is checked for any shape.

ANGERMAN/BIRC: A PURE CORE WITH A THIN FOREIGN EDGE

Repository: angerman/birc `(https://github.com/angerman/birc)`

A terminal IRC client with a pure Bend core and a small C FFI for terminal rendering and POSIX sockets. The README reports more than 91% of the code in Bend, a zero-poll event loop, and a native DNS client.

Read it for: the actor-and-frame architecture. `src/bend/irc.bend` parses the protocol, `frame.bend` handles CRLF framing, `client.bend` and `buffer.bend` hold state, `submit.bend` builds outbound lines, `view.bend` lays out the screen, `session.bend` ties the pieces together, and `actor.bend` and `net.bend` do the I/O. `src/ffi/` is the thin C layer, and `src/ui/timui.h` is the rendering library.

Start at: `LAWS.bend` for the protocol and client invariants, then `src/bend/irc.bend` and `src/bend/frame.bend`. The `tests/pty/` directory is the part most projects skip: it drives the real terminal through a pseudo-terminal, including resize, paste, EOF, and UTF-8 split across reads.

Composition lesson: foreign code is acceptable when it is small, named, and at the edge. The domain stays testable because the protocol and state transitions are pure.

FABIANVEGAA/MYLSM: A SYSTEMS PROGRAM WITH PER-MODULE LAWS

Repository: FabianVegaA/mylsm `(https://github.com/FabianVegaA/mylsm)`

An experimental durable LSM key-value store. It writes to a log, keeps sorted tables, recovers after a restart, and exposes byte and UTF-8 views of the same database.

Read it for: law files per module. `src/` holds the storage modules (`ByteWal.bend`, `Wal.bend`, `ByteMemTable.bend`, `SstFile.bend`, and the rest), `laws/` holds a matching file for each (`laws/ByteWal.bend`, `laws/ByteMemTable.bend`, and so on), and `PROOF.bend` imports them into one gate. `app/cli/` and `app/repl/` are separate applications over the same library. `bench/smoke/`, `bench/crash/`, and `bench/workload/` cover ordinary paths, fault injection, and sustained runs.

Start at: `laws/ByteWal.bend` and `src/ByteWal.bend` together, then `app/cli/main.bend`. The README is unusually explicit about the limits: the laws cover pure decisions, not filesystem behavior, process locking, or the C and JavaScript host effects.

Composition lesson: a large systems program can keep its proof surface small by stating laws per module and importing them at the root. The `app/` layer is a client of the library, not part of it.

BENDLIB/BENDLIB: LEMMAS AND A LAW CHECKER

Repository: bendlib/bendlib (https://github.com/bendlib/bendlib)

A foundation library with `bend-mathlib` (published lemmas about `Nat`, `Bool`, `List`, `String`, `Maybe`, equality, and order) and `lawcheck`, a tool that looks for counterexamples before you try to prove a law.

Read it for: how to publish proofs as a library. `packages/bend-mathlib/` has one module per subject (`nat.bend`, `list.bend`, `bool.bend`, `maybe.bend`, `order.bend`, `sort.bend`, `perm.bend`), an `all.bend` entry point, and a README with a lemma index. `tools/lawcheck` is the complement of the proof gate: it tests the law against the implementation and shrinks a failing input.

Start at: `packages/bend-mathlib/README.md` for the lemma index, then `nat.bend` for the style. Read the `lawcheck` section of the README before writing a new law; a false law is cheaper to find by testing than by attempting a proof.

Composition lesson: proofs are reusable when they are packaged like code. The `all.bend` facade keeps imports stable while the individual modules stay focused.

777GENIUS/AWESOME-BEND: THE CURRENT INDEX

Repository: 777genius/awesome-bend (https://github.com/777genius/awesome-bend)

A curated list for current Bend 2, explicitly not Bend 1. It tracks the latest release, the hub, the official compiler, packages, tools, demos, benchmarks, papers, and community links.

Read it for: discovery and release context. The list records the changes between 2.0.x releases and warns which packages may need a newer compiler. The catalog section links to generated indexes with search and filters, which are better than a static list when you are looking for a specific package.

Start at: the "Packages", "Demos", and "Benchmarks" sections. Then check the repository's "Last checked" date; a list is only as current as its snapshot.

Composition lesson: an index is part of the ecosystem. It tells you which repositories are maintained, which are experiments, and which compiler version each one expects.

MORE TO EXPLORE

Repository	What to look at
nuxyel/llama.bend (https://github.com/nuxyel/llama.bend)	A Llama-architecture implementation checked token for token against PyTorch. `reference/` holds the Python oracle; `tests/bad_weight.bend` and `tests/bad_cap.bend` are negative tests.
rootagi/bend-json (https://github.com/rootagi/bend-json)	A JSON parser and serializer with `LAWS.bend`, `PROOF.bend`, an independent Python test suite, and four runnable examples.
ind-igo/bend-evm (https://github.com/ind-igo/bend-evm)	A backend that consumes `bend-frontend`'s `Core.Program`, lowers it to Yul, and carries a certificate plus per-contract laws.
CrocSwap/bendSVM (https://github.com/CrocSwap/bendSVM)	A Bend 2 to Solana SBF compiler with a shared SDK and several example contracts, each with its own `LAWS.bend` and `PROOF.bend`.
MatheusBBarni/bendJVM (https://github.com/MatheusBBarni/bendJVM)	A JVM interpreter organized as parser, verifier, linker, and runtime, with a differential test harness against the reference JVM.
nicolas-abril/bend2-libs (https://github.com/nicolas-abril/bend2-libs)	Many small packages in one repository (`bytes`, `deflate`, `png-decode`, `jpeg-decode`, `truetype`, `http-server`), each with its own entry and tests.
OkeyAmy/bend-voxel-bench (https://github.com/OkeyAmy/bend-voxel-bench)	An honest benchmark: a Luanti terrain port, a live race against the C++ original, and a `docs/bend-notes.md` list of the traps hit along the way.
jonathanperis/jonlib (https://github.com/jonathanperis/jonlib)	A raylib-compatible graphics and math library that replays upstream examples frame by frame against the native reference.

WHAT TO COPY, AND WHAT NOT TO

Copy these:

- **The gate.** `LAWS.bend` plus `PROOF.bend` at the root of the project, and a README command that runs it.
- **The pure core.** A domain module that takes values and returns values, with the clock, the window, the filesystem, and the network in a separate driver.
- **The named effect.** A `.c` or `.js` file behind one Bend signature, with the host boundary documented.
- **The matching test.** A `tests/` file for each domain module, and a `check.bend` for the parts the checker cannot reach.
- **The pinned version.** The compiler version in the README, and the package version or hash in the import.

Do not copy these:

- **A law that restates the implementation.** A claim such as "for every `x`, `f(x) == f(x)`" passes and says nothing. A law should name a behaviour a caller relies on.
- **A proof that hides a hole.** A `?TODO` or an `@unsafe` dependency makes the gate weaker. The tutorial's chapter 07 shows the messages.
- **A benchmark without a baseline.** A time in milliseconds with no machine, thread count, compiler version, or comparison is not evidence.
- **Foreign code as a shortcut.** If a `.c` file does the work, the Bend code is a wrapper, and the laws prove the wrapper, not the work.
- **A repository wholesale.** Bend 2 is moving. A repository pinned to 2.0.22 may use syntax that 2.0.37 rejects, and the reverse. Read its version first.

VERSIONS AND FRESHNESS

All repository links in this chapter were checked on 2026-10-20. The latest stable release named by `awesome-bend` on that date was Bend 2.0.37. The tutorial's own runnable blocks were checked with Bend 2.0.25, so a repository that requires a newer compiler may reject code that this tutorial accepts, and a repository that uses an older feature may need `bend update` or a pinned checkout.

When a repository and this tutorial disagree, the compiler in front of you decides. Run `bend version`, then run the repository's proof command. If the proof gate passes, read the code; if it fails, read the version notes before changing anything.

NEXT

Go back to the table of contents, or pick one repository and read it in the order above. The most useful next step is to take `metal-bending`'s shape, add one law about your own code, and run the gate.

C.39 · `bend-gme/version-1/.opencode/skills/bend2-runtime/SKILL.md`

Bend 2: the runtime and parallelism

The theory is interaction nets; the shipped compiler is BendRT. Design for the second, think with the first, and measure.

MODEL

- **Interaction nets**: a graph of cells with ports; an *active pair* (two principal ports joined) rewrites by a local rule. Rules are local and the system is confluent, so independent active pairs can fire in any order or in parallel. β-reduction is one interaction (`references/01-interaction-nets.md`, `references/07-four-interactions.md`).
- Optimal (lazy) sharing and superposition are properties of an IC runtime. **Superposition and sharing only work on HVM5, and HVM5 does not exist yet as a usable runtime.** BendRT copies.
- **Released Bend 2 does not run on nets.** It emits BendRT, a flat C evaluator with no active pairs at runtime (`../bend2/references/01-tutorial-index.md`, Honest limits).
- Parallelism in BendRT comes from the **fork shape**: independent recursive calls over a balanced tree of leaves; `--threads` sets workers; `!` calls target the GPU (`references/02-parallelism-and-runtime.md`, `references/04-gpu-concepts.md`).
- **JS target**: one Bend call runs on one thread. A browser app gets parallelism from N module Web Workers each running `core.mjs` on disjoint batches (spec/12).

RULES

1. Write divide-and-conquer over balanced structures (split in halves, combine) instead of a sequential accumulator, when the work is large and the combine is associative.
2. Keep leaves big enough to amortize the fork; tune the leaf size by measurement.

3. Never claim a speedup without: machine, version, thread count, input size, wall time, and the sequential baseline. On this devcontainer only the JS target builds (see Patterns): say "JS lane" and do not extrapolate to native.
4. Count what the compiler emits when performance surprises you (`references/02-parallelism-and-runtime.md` §"Count what the compiler emits").
5. On the JS target, batch work across workers and transfer columns once per session; do not expect `--threads` to apply.
6. Laws never prove cost; cost is measured (`bend2-proofs`).
7. Native is not "the fast target": measure both lanes on the actual workload. A cons-heavy interpreter can be 2.3× *slower* natively (measured; table in Patterns). Report machine, version, thread count and both lanes.

PATTERNS

```
sum(tree):                      # parallel-friendly shape (pseudocode)
  match tree:
    Leaf{xs}:      sum_list(xs)            # sequential, tail-recursive leaf
    Node{l, r}:    sum(l) + sum(r)         # two independent calls: the fork
```

```
 BEND_NO_TELEMETRY=1 bend prog.bend -o prog.js  # JS target -- works with the installed binary
time node prog.js                              # measure; repeat for a stable number

# Native. FIRST check whether the installed binary still fails -- this workaround
# is meant to expire (R-41):
#   BEND_NO_TELEMETRY=1 bend hello.bend -o /tmp/hello && /tmp/hello
# If that builds and runs, use the plain `bend ... -o out` everywhere and DELETE
# this block: the failure is a property of one bad build, not of the machine.
# While it does fail:
#   bend prog.bend -o prog    ->  TypeError: undefined is not an object
#                                 (evaluating 'FL.book.tlds[k].c')   (bend 2.0.39)
# The source CLI of the same tag builds the same file natively (verified
# 2026-11-01 with hello world, the site's pow2 example and bench/vm.bend):
git clone --depth 1 --branch v2.0.39 https://github.com/bendlang/bend.git /tmp/bend-src
BEND_NO_TELEMETRY=1 bun /tmp/bend-src/bend2/main.ts prog.bend -o prog
./prog +RTS --threads 1          # native options: --threads N, --gpu off, closed by -RTS
```

Do not assume native is faster. Measured on the same 200,000-frame loop (`bench/vm.bend` , WSL2 x86-64, bend 2.0.39, min of 5 native / 7 JS runs; full method and the corrected estimate in `docs/05-benchmarks/02-vm-interpreter.md`):

Same logic	JS lane (V8)	native (BendRT)
AST interpreter (cons-heavy, allocation per node)	**735 ns**	**1,678 ns** — 2.3× *slower*
compiled template def (no interpreter)	25 ns	17 ns — 1.4× faster
hand-written predicate	3 ns	4 ns

BendRT has no GC, but it allocates and reference-counts a cell per cons; V8 JITs the same cons lists with hidden classes, escape analysis and a cheap nursery. So the native win is workload-dependent — allocation-heavy code can be *slower* — and what actually removes the 30–100× is **compiling the strategy**, not changing the runtime.

PITFALLS

Symptom	Cause	Fix
no speedup with more threads	sequential dependency chain (accumulator)	restructure into a tree reduce
speedup on native, none in the browser	JS target is single-threaded per call	shard across Web Workers
"optimal reduction" claims in a design doc	theory vs BendRT	label as designed; measure BendRT
slower with tiny leaves	fork overhead dominates	larger leaves

VERIFY

```
bend --help                    # confirm flags for the installed version before using them
bend prog.bend -o prog.js && time node prog.js   # the target that works here
```

REFERENCES

External references (bend2.dev serves each page as Markdown with `Accept: text/markdown`):

Page	Use
https://bend2.dev/notes/install-bend2-from-source/ (https://bend2.dev/notes/install-bend2-from-source/)	install, build from source, clang 14+ (19+ for GPU), `+RTS`, the exact `-o` commands
https://bend2.dev/notes/bend-runtime-options/ (https://bend2.dev/notes/bend-runtime-options/)	`+RTS --threads N --gpu off`, `-RTS`, `BENDRTS`, `IO.args`
https://bend2.dev/notes/bendrt-explained/ (https://bend2.dev/notes/bendrt-explained/)	what BendRT does at run time
https://bend2.dev/learn/compile-to-c/ (https://bend2.dev/learn/compile-to-c/)	the C target
https://bend2.dev/llms.txt (https://bend2.dev/llms.txt)	retrieval index for examples and diagnostics

File	Covers	Open when
`references/01-interaction-nets.md`	cells, ports, active pairs, rules, confluence, symmetric interaction combinators, labels, β as one step	reasoning about the model
`references/02-parallelism-and-runtime.md`	parallel call, fork shape, native build, measuring, thread count, counting emitted code	making code faster
`references/03-parallelism-concepts.md`, `references/04-gpu-concepts.md`, `references/05-hvm-hands-on.md`	concept pages, GPU calls, hands-on HVM	quick refresher
`references/06-interaction-nets-notes.md`, `references/07-four-interactions.md`, `references/08-hvm-runtime-notes.md`, `references/09-optimal-computer.md`, `references/10-bend-vs-mojo.md`	research notes	deep background, comparisons

RELATED

`bend2-superposition` (DUP/SUP) · `hvm5-ic` (an IC kernel written in Bend) · `bend2-effects-targets` (workers) · `bend2-app`.

C.40 · `bend-gme/version-1/.opencode/skills/bend2-runtime/references/01-interaction-nets.md`

--- title: ”Bend 2 tutorial 04: interaction nets” description: ”Cells, ports, active pairs, local rewrite rules, confluence, Lafont’s symmetric interaction combinators, and beta-reduction as one interaction, with what the released Bend 2 runtime does instead.” updated: 2026-10-18 verified: 2026-10-18 status: docs-only canonical: /tutorial/04-interaction-nets/ sources:

- https://bend2.dev/notes/interaction-nets-explained/ (https://bend2.dev/notes/interaction-nets-explained/)
- https://bend2.dev/notes/bendrt-explained/ (https://bend2.dev/notes/bendrt-explained/)
- https://bend2.dev/notes/hvm4-explained/ (https://bend2.dev/notes/hvm4-explained/)
- https://bend2.dev/learn/duplication/ (https://bend2.dev/learn/duplication/)
- https://dl.acm.org/doi/10.1145/96709.96718 (https://dl.acm.org/doi/10.1145/96709.96718)
- https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432) ---

04: interaction nets

This chapter is about a computation model, not a Bend program. You will not run any code here. Its purpose is to explain three words you will see in Bend's documentation, *active pair*, *confluence*, and *duplication*, and why the designers started from this model.

Status: **docs-only**. The examples are pen-and-paper exercises. Chapter 05 shows what the released compiler does.

THE PROBLEM: COPYING IS THE WHOLE GAME

Lambda calculus has one operation that does the work: β-reduction, which substitutes an argument into a function body. A variable used twice in the body forces you to copy the argument, and if the argument is itself an expensive computation, the copy repeats the work. Tree-shaped rewriting has no way to share that work.

Interaction nets solve this by representing a term as a graph. A shared piece of the graph is one node with several wires leaving it, so it is computed once. The price is that copying becomes an explicit operation with its own rules.

CELLS, PORTS, AND NETS

A **cell** has a symbol (its name), exactly one **principal port**, and a fixed number of **auxiliary ports**. A **net** is a collection of cells whose ports are joined pairwise by wires. A port that has no wire is free, and the free ports form the net's *interface*.



Core insight. A net is a graph, and its rules rewrite the graph. Released Bend 2 does not build that graph. Its runtime runs definitions, as the BendRT block below shows.

Composed of: a net.

```
net
- cells: CON | DUP | ERA, each with one principal port
- auxiliary ports: CON 2, DUP 2, ERA 0
- wires: each joins two ports; a port with no wire is part of the interface
```

ACTIVE PAIRS AND RULES

Two cells joined by a wire between their principal ports form an **active pair**. Only an active pair can be reduced. A **rule** replaces an active pair with another net that has the same free ports. The rule is local: it reads two cells and rewrites them, and nothing else.

Each unordered pair of symbols has at most one rule. A net with k symbols therefore has at most $k(k+1)/2$ rules. A net is in **normal form** when no active pair remains.

Two active pairs never share a cell. A cell has one principal port, so it can be part of at most one active pair. This is the property that makes local, parallel reduction safe.

CONFLUENCE: ORDER DOES NOT MATTER

Take two distinct active pairs in one net. Reducing either one leaves the other still reducible, and reducing both in either order produces the same net. This is **one-step confluence**, the diamond property:



The bend2.dev page on interaction nets states a stronger property for the same systems: if a net has a normal form, reduction order does not change that result, or the number of interaction steps needed to reach it. The same page is clear about what this does not give you. It does not guarantee that every program terminates, that every

implementation of the rules is correct, or that the implementation is free of synchronization, allocation, and scheduling costs.

Read those two claims as the model's claims. Chapter 05 shows what a real runtime must still pay for.

THE SYMMETRIC INTERACTION COMBINATORS

Lafont (1997) showed that three symbols are enough. The system is *symmetric* because its two binary symbols have the same shape, one principal and two auxiliary ports, so no special primitive is needed for lambda or copy. The bend2.dev page uses the Greek names γ, δ, ε; this chapter uses the names common in later literature:

Symbol	Name	Auxiliary ports
CON	constructor (γ)	2
DUP	duplicator (δ)	2
ERA	eraser (ε)	0

There are six rules, one for each unordered pair of symbols:

1. **CON–CON annihilates.** The two cells disappear, and their auxiliary ports are reconnected pairwise: first with first, second with second.
2. **DUP–DUP annihilates.** The same crosswise reconnection as rule 1.
3. **ERA–ERA annihilates.** Both cells disappear and nothing replaces them.
4. **CON–DUP commutes.** Four new cells appear: two DUP copies attached to the CON's auxiliary wires, and two CON copies attached to the DUP's auxiliary wires. Each new CON is wired to each new DUP, so the four new cells form a 2×2 crossing. The number of cells goes from two to four.
5. **CON–ERA erases.** The CON cell disappears, and two new ERA cells are attached to its two auxiliary wires.
6. **DUP–ERA erases.** The same as rule 5, for a DUP cell.

Rule 4 is the one that copies. A DUP that meets a CON is pushed through it, so each of the CON's two results gets its own DUP. Repeated application copies a structure one layer at a time, and each step is one local interaction.

Lafont proved that these six rules are universal: a translation maps any other interaction system into them and preserves its computation and its parallelism. The bend2.dev page describes this as "universal", and it does not claim that the six rules alone make programs fast.

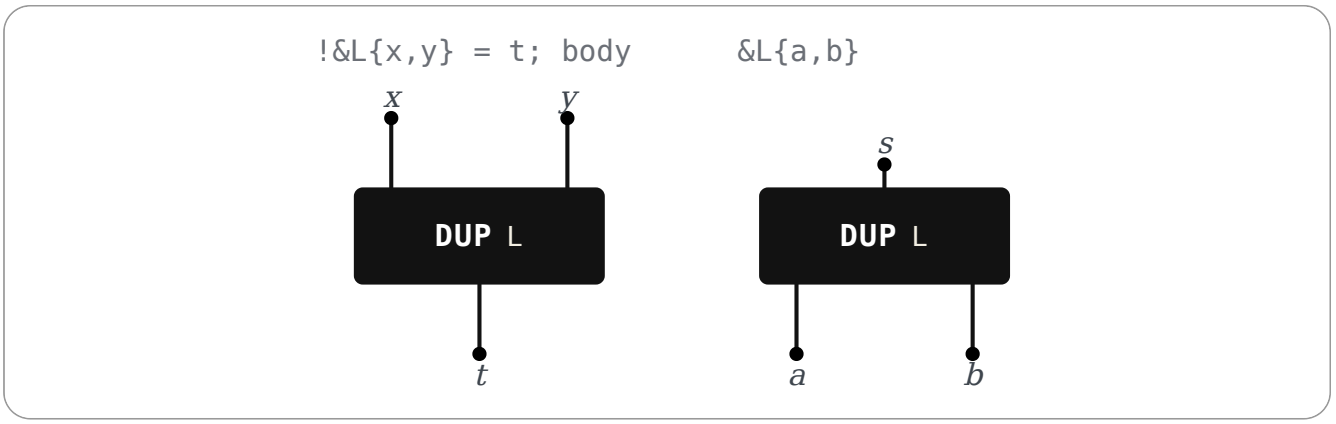
LABELS: THE INTERACTION CALCULUS

The three combinators above have one kind of DUP. Victor Taelin's Interaction Calculus (IC) keeps the same three kinds of cell, gives every DUP a **label** (a number), and adds a text notation for nets. The text is not a drawing: `!&L{x,y} = t; body` names cells and wires, the way `λx.t` names a CON. Read the text as cells:

Text	Cell	Ports
`λx.t` and `(f a)`	CON	for `λx.t`, the output is principal and the aux ports hold the variable and the body; for `(f a)`, the function is principal and the aux ports hold the argument and the result
`!&L{x,y} = t; body`	DUP, label `L`	principal: `t`; aux: `x` and `y`; `body` is the rest of the net
`&L{a,b}`	DUP, label `L`, read from the other side	principal: the value; aux: the alternatives `a` and `b`
`*`	ERA	principal only

Give every DUP one label, and the rule that two DUPs annihilate (rule 2 above) is the case where their labels match. Different labels commute instead. So the symmetric combinators are the special case with a single label.

So a superposition is not a new kind of cell. `&L{a,b}` is a DUP labelled `L`, read from the other side: its principal port faces the value, and its aux ports hold the alternatives. Both readings are drawn side by side:



Chapter 14 runs these rules on the reference implementation, prints the interaction counts, and shows where a superposition is born.

B-REDUCTION AS ONE INTERACTION

Interaction nets encode a lambda abstraction and an application as two CON cells:

- **λx. M** is a CON whose principal port is the lambda's output, whose first auxiliary port is the variable x (a wire to every use of x), and whose second auxiliary port is the body M.
- **(F A)** is a CON whose principal port connects to the function F, whose first auxiliary port carries the argument A, and whose second auxiliary port carries the result.

When a lambda meets an application through their principal ports, that is an active pair of two CONs, and rule 1 applies. The first auxiliary wires join: the variable's wire now connects to the argument A. The second auxiliary wires join: the body M now connects to the result. β-reduction is one interaction.

If the variable is used twice, its wire must be duplicated, and each copy is a separate CON–DUP commutation. That is the duplication cost. The model does not hide it, and Chapter 04's exercises count it.

WHY THIS MATTERS FOR BEND, AND WHAT IT DOES NOT

The bend2.dev notes describe interaction nets as the model behind Bend's earlier HVM runtimes. Released Bend 2 does not run them. Its compiler emits a flat C evaluator, called BendRT, that keeps pending work explicitly and does not reduce graphs. The notes list what BendRT does instead:

- **Affine ownership.** Each ordinary value has at most one live use, so the compiler knows when memory can be freed. No tracing garbage collector is needed.
- **Counted sharing.** A `+` value keeps a reference count. A value used through `+` is shared by count, not by graph.
- **Balanced forks.** `a b = f(x) g(y)` hands each call to a core once. The programmer promises the two calls take about the same time.

> **Core insight.** BendRT is a flat evaluator of definitions. Its primitives are words, definitions, jumps and tasks, not cells and rewrites.

Composed of: BendRT, the released runtime.

```
BendRT
- term: one 64-bit word; a small value is stored inline, anything else is a pointer into one heap shared by every core and the GPU
- definition: a segment of a flat state machine; a call is a jump
- parallel call: one join task, plus one task per call, dealt to CPU or GPU lanes
- memory: no garbage collector; a match frees the node it opens; only `+` values carry a reference count, which the emitted C keeps in 24 bits
```

The guide's Under the Hood section gives the first three parts and the memory rule. The 24-bit width comes from the emitted C.

So the model explains *why* Bend is affine and why copying is spelled out. It does not predict how fast a program runs. The notes quote the Asperti and Mairson result that the implementation cost of optimal reduction cannot be bounded by an elementary function of the number of reduction steps. Counting interactions is not the same as measuring time.

SUMMARY OF PROPERTIES

Property	What it says	Where the page states it
Locality	a rule reads and rewrites only its two cells	interaction-nets page
One-step confluence	two distinct active pairs commute	interaction-nets page, citing Lafont 1990
Step-count invariance	any order to a normal form takes the same number of steps	interaction-nets page
No termination guarantee	a net can reduce forever	interaction-nets page
Universality	the six combinator rules can encode other systems	interaction-nets page, citing Lafont 1997
Cost is not steps	allocation, scheduling, and memory still cost time	bendrt-explained and interaction-nets pages

EXERCISES (PEN AND PAPER)

1. Erase a cell. A CON has its principal port wired to an ERA. Its two auxiliary ports are wired to two free ports, `p` and `q`. How many interactions happen, and what is left?

Solution. One interaction: rule 5. The CON disappears and two new ERA cells attach to `p` and `q`. Before the step there were two cells (CON, ERA). After it there are two cells (ERA, ERA). No active pair remains, so the net is in normal form.

2. Count the cells of a commutation. A CON and a DUP are joined through their principal ports. Their four auxiliary ports are wired to four free ports. How many cells and how many internal wires exist after rule 4?

Solution. Four cells: two DUP copies and two CON copies. The copies' principal ports take the four free wires, and their auxiliary ports form four internal wires, one for each (CON copy, DUP copy) pair. The number of cells rises from two to four.

3. Two active pairs. Net N has two active pairs, A and B, that share no cell. Show that reducing A then B gives the same net as reducing B then A.

Solution. Rule A rewrites only the cells of A and creates wires only to A's free ports, which are cells or interface ports that B does not touch. The same holds for B. Both orders therefore produce the same set of cells and wires, and the diamond closes.

4. Beta in one step. Encode `(λx. x) y` with two CONs, where `y` is a free port. Reduce it.

Solution. The lambda CON's first auxiliary wire (the variable x) meets the application's first auxiliary wire (the argument y), and the lambda's body wire meets the result wire. The body of the lambda is `x`, which is the variable's wire, so the result is directly connected to `y`. One interaction; no DUP was needed because `x` is used once.

NEXT

Chapter 05 shows what the released compiler does for a parallel call, and what a balanced fork looks like in code.

C.41 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/02-parallelism-and-runtime.md

--- title: "Bend 2 tutorial 05: parallelism and the runtime" description: "Parallel calls, the balanced fork shape with leaves, GPU calls with bang, --threads, how to measure a speedup honestly, and how to count keeps in the emitted C." updated: 2026-10-18 verified: 2026-10-18 status: runnable canonical: /tutorial/05-parallelism-and-runtime/ sources:

- https://bend2.dev/learn/parallelism/ (https://bend2.dev/learn/parallelism/)
- https://bend2.dev/learn/measuring-speedup/ (https://bend2.dev/learn/measuring-speedup/)
- https://bend2.dev/learn/gpu/ (https://bend2.dev/learn/gpu/)
- https://bend2.dev/notes/bendrt-explained/ (https://bend2.dev/notes/bendrt-explained/)
- https://bend2.dev/notes/ai-agents-bend/ (https://bend2.dev/notes/ai-agents-bend/)
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md) ---

05: parallelism and the runtime

Bend runs independent recursive calls in parallel without threads, locks, or annotations. You write a *parallel call*, and the compiler makes it a fork. Two promises come with it. The calls must be independent, which purity and affine ownership guarantee. They must also take about the same time, which is your job. This chapter shows the shape that meets the second promise, and how to check the result honestly.

THE PARALLEL CALL

```
import Base

def pow2(+d: Nat) -> U32:
  match d:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)

def main() -> U32:
  pow2(12n)
```

```
4096
```

`a b = pow2(p) pow2(p)` is one line with two calls. Both start together, and the line continues when both have finished. `+d` lets both calls use the depth. The checker runs this program and prints 4096. A native binary runs the same calls on several cores. Compile it with `bend pow2.bend -o pow2` and run `./pow2 --threads 8`.

Bang runs the calls on a GPU. `pow2!(12n)` sends that call and every parallel call it makes to the GPU. The lab machine for this tutorial has no GPU device (there is no `/dev/dri` and no NVIDIA device), so the bang form was not timed here. On a machine with no usable GPU, `--gpu off` runs bang calls on the CPU threads.

HOW A CALL RUNS

A top-level definition is code, not a value, so calling it does not use it up. The checker counts uses of values: a closure may be called once, and a `+` value costs a reference count when it is shared (chapter 03). A definition may call itself as often as its argument shrinks (chapter 02).

The guide describes the run-time shape of those calls:

- A tail call compiles to a loop. `leaf` below calls itself in tail position, so each leaf becomes one loop.
- Bend has no C stack. Each definition compiles to a segment of a flat state machine, and a call is a jump.

THE FORK SHAPE: A TREE OF LEAVES

A parallel call that splits one call into two equal halves is a *fork*. Forking at every level down to single numbers costs more in scheduling than the work itself. The shape that works has two parts:

- **Tree.** Above a fixed depth, split the range in two equal halves with a parallel call.
- **Leaf.** Below the tree, a tail loop walks 8 to 64 consecutive items with scalar accumulators. A loop of this kind compiles to one native loop.

This program sums the numbers from 0 to 511 with eight leaves of 64 items each:

```
import Base

def leaf(+n: Nat, +i: U32, acc: U32) -> U32:
  match n:
    case 0n:
      acc
    case 1n+p:
      leaf(p, (i + 1 : U32), (acc + i : U32))

def tree(+d: Nat, +i: U32) -> U32:
  match d:
    case 0n:
      leaf(64n, i, 0)
    case 1n+p:
      a b = tree(p, i) tree(p, (i + (64 << p) : U32))
      (a + b : U32)

def main() -> U32:
  tree(3n, 0)
```

```
130816
```

The check is arithmetic: the numbers 0 to 511 sum to 511 × 512 / 2 = 130816. Two details carry the design:

- The second half starts at `i + (64 << p)`. Each leaf covers 64 indices, so a tree of depth `p` covers `64 << p` indices, and the second half begins right after the first. The general formula is `i + (L << p)`, where `L` is the leaf size. With leaves of one item it is `i + (1 << p)`, which is why that form appears in other examples. With leaves of 64 it must use 64.
- The tree's halves are equal in work, because every leaf does the same 64 steps. A fork over unequal halves would leave one core idle.

Use a `tree` for data a fork splits, and a `List` for data a leaf walks once. Head-and-tail recursion over a list runs on one core, so use it only inside a leaf. Use an `Array` for data you index or update in place (chapter 03).

EXERCISES

1. Four levels. Change the depth to `4n`. How many numbers are summed, and what is the output? Check your answer by running the program.

Solution. The tree covers 64 × 2^4 = 1024 numbers, so the sum is 1023 × 1024 / 2 = 523776.

```
import Base

def leaf(+n: Nat, +i: U32, acc: U32) -> U32:
  match n:
    case 0n:
      acc
    case 1n+p:
      leaf(p, (i + 1 : U32), (acc + i : U32))

def tree(+d: Nat, +i: U32) -> U32:
  match d:
    case 0n:
      leaf(64n, i, 0)
    case 1n+p:
      a b = tree(p, i) tree(p, (i + (64 << p) : U32))
      (a + b : U32)

def main() -> U32:
  tree(4n, 0)
```

```
523776
```

2. Break the midpoint. Replace `64 << p` with `1 << p` in the program above and run it. Explain the output before you run it.

Solution. The halves overlap. The second half starts only one index after the first, so most indices are summed several times and the total is wrong. Expect the program to print 17920, and that the number is not 130816.

```
import Base

def leaf(+n: Nat, +i: U32, acc: U32) -> U32:
  match n:
    case 0n:
      acc
    case 1n+p:
      leaf(p, (i + 1 : U32), (acc + i : U32))

def tree(+d: Nat, +i: U32) -> U32:
  match d:
    case 0n:
      leaf(64n, i, 0)
    case 1n+p:
      a b = tree(p, i) tree(p, (i + (1 << p) : U32))
      (a + b : U32)

def main() -> U32:
  tree(3n, 0)
```

```
17920
```

Each leaf still sums 64 consecutive values from `i`. With the midpoint `i + (1 << p)`, the second half of a level-one tree starts one index after the first, so every leaf overlaps its neighbour by 63 values. The total counts many values several times, and 17920 is not the sum 0 to 511. The check is simple: whenever a parallel sum disagrees with the sequential sum, recompute the midpoints first.

BUILD A NATIVE BINARY AND RUN IT

The checker runs a pure `main` slowly, so measure with a native build:

```
bend main.bend -o main
./main --threads 1
./main --threads 8
```

Both runs must print the same output. If they do not, the program has a bug, not a scheduling problem. The speedup is the one-thread time divided by the multi-thread time, measured on the same input, with checks that the outputs match.

MEASURE HONESTLY

A speedup claim needs five things:

1. The same input for every run, and the same output checked against the one-thread result.
2. Several runs per mode, reported as a median, with the order rotated between modes.
3. The machine, thread count, GPU setting, and compiler version written down with the numbers.
4. A named baseline. "Twelve times faster" means nothing without the two times.
5. Build time separate from run time. Compile first, then time only the binary.

Small inputs mislead: for work this small, process startup dominates the time. Measure with inputs big enough that the run takes long enough to time reliably, and state how long that is.

CHOOSING THE THREAD COUNT

Bend has no automatic setting. `--threads` takes a number from 1 to 128, and without the flag a binary uses the CPU count, which `./main --help` prints. `--threads auto` is rejected:

```
$ ./main --threads auto
bend: expected a thread count of 1 or more after --threads
```

Definition. The thread count is the `--threads N` setting of a native binary: the number of CPU threads it runs on, from 1 to 128. Without the flag it is the CPU count.

Core insight. The fastest count belongs to one program, one input, and one machine. The number of leaves does not give it. `tree(d)` has 2^d leaves, far more than any thread count, and the scheduler hands each task to a core once and never moves it (the guide). Measure.

A script can do the measuring. It runs each count after a warm-up, requires every run to print the output of the one-thread run, and reports the median and the range:

```
#!/usr/bin/env bash
# sweep.sh PROGRAM [THREADS...]
# Runs PROGRAM at each thread count after one warm-up run. Every timed run must
# print the same output as the one-thread run, or the sweep stops. Prints the
# median and range of RUNS runs per count, then the fastest count.
set -euo pipefail
prog=$1
shift
[ "$#" -gt 0 ] || set -- 1 2 4 8 12
runs=5
reference=$("$prog" --threads 1)
best_t=""
best_ms=""
for t in "$@"; do
  "$prog" --threads "$t" >/dev/null
  times=""
  for _ in $(seq "$runs"); do
    start=$(date +%s%N)
    out=$("$prog" --threads "$t")
    end=$(date +%s%N)
    if [ "$out" != "$reference" ]; then
      echo "threads=$t printed a different result; stopping" >&2
      exit 1
    fi
    times="$times $(( (end - start) / 1000000 ))"
  done
  sorted=$(printf '%s\n' $times | sort -n)
  median=$(printf '%s\n' "$sorted" | sed -n "$(( (runs + 1) / 2 ))p")
  low=$(printf '%s\n' "$sorted" | head -1)
  high=$(printf '%s\n' "$sorted" | tail -1)
  echo "threads=$t median=${median} ms (range ${low}-${high} ms)"
  if [ -z "$best_ms" ] || [ "$median" -lt "$best_ms" ]; then
    best_t=$t
    best_ms=$median
  fi
done
echo "fastest: --threads $best_t (${best_ms} ms)"
```

Save it as `sweep.sh` and run `bash sweep.sh ./main 1 2 4 8 12 16 24`, with `main` set to `tree(26n, 0)`. On the lab machine (12 CPUs) it printed:

```
threads=1 median=366 ms (range 343-372 ms)
threads=2 median=214 ms (range 208-237 ms)
threads=4 median=150 ms (range 143-151 ms)
threads=8 median=108 ms (range 102-118 ms)
threads=12 median=112 ms (range 107-115 ms)
threads=16 median=108 ms (range 100-113 ms)
threads=24 median=101 ms (range 98-112 ms)
fastest: --threads 24 (101 ms)
```

Three things stand out:

- Most of the gain comes by 8 threads: 366 ms falls to 108 ms.
- The default of 12 was not the fastest. Sixteen and 24 were a few milliseconds faster, but their ranges overlap with the range of 12.
- On `tree(22n, 0)`, one thread took 22 ms, and each of 4, 8, 12, 16, and 24 threads took 14 to 17 ms. Those differences are about the size of the run-to-run spread, so the sweep cannot choose a count there. Use an input large enough to time, as the section Measure honestly says.

The script has limits. It times wall-clock time on the machine it runs on, so other load changes the numbers. It needs bash and GNU `date`. Its answer holds for this program, this input, and this machine. The guide says the GPU shines on uniform numeric work, such as mandelbrot or nbody, and that divergent work, such as n-queens, stays faster on the CPU. Measure those too, with `!` and without it.

COUNT WHAT THE COMPILER EMITS

`bend main.bend -o main.c` writes the C source. Three names mark runtime work that copies or releases a shared value: `term_keep`, `ctr_take`, and `rfc_seal`. A reference count is a `term_keep`. A program that uses `+` on values pays for them, and a program that does not should have none in its own functions.

For the `pow2` program above, the compiler emitted `term_keep` 2 times, `ctr_take` 3 times, `rfc_seal` 2 times, and `term_peek` 6 times. A program with no sharing at all, `def main() -> U32: 1`, emits exactly the same counts. So those occurrences come from the runtime, and `pow2` adds none of its own. `term_peek` is a borrow, which costs no count. When you audit a program, compare its counts with that baseline, and treat any new keep in your own code as a regression to remove or explain.

NEXT

Chapter 06 covers duplication and superposition, the terms behind labels and search. It also says plainly what the released compiler does not have.

C.42 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/03-parallelism-concepts.md

--- title: Automatic parallelism description: "The fork is where data flow splits — ordinary recursion the runtime runs concurrently." updated: 2026-08-13 verified: 2026-09-05 status: docs-only canonical: /learn/parallelism/ sources:

- https://bend2.dev/learn/parallelism/ `(https://bend2.dev/learn/parallelism/)` ---

Automatic parallelism

Bend2 parallelizes ordinary recursion: split the work in two, and the runtime evaluates the halves concurrently. No pragmas, no threads, no scheduler in the program. The program below sums 2^{24} numbers this way.

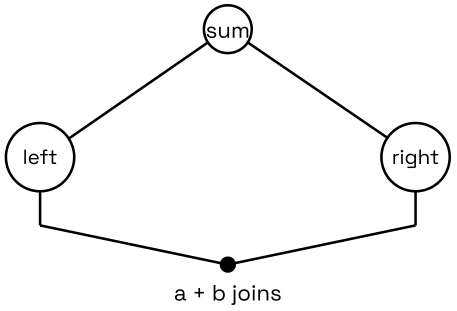
```
import Base

# Sums a range of numbers in parallel.
def sum(d: U32, i: U32) -> U32:
  match d:
    case 0:
      i
    case 1+d:
      a, b = sum(d, i*2), sum(d, i*2+1) # fork
      a + b

# Runs sum on the GPU, via `!`.
def main() -> U32:
  sum!(24, 0)
```

THE FORK

Bend2 forks where the data flow splits. `a, b = sum(d, i*2), sum(d, i*2+1)` starts both recursive calls, and nothing joins them until `a + b` needs both results. The calls share no state, so on the interaction-net runtime they are disjoint subgraphs — recall the model: disjoint pairs cannot conflict — and any idle core reduces either one without locks.



The fork: one call becomes two disjoint subgraphs; *a + b* is the only join.

THE MATCH

Bend2 pattern-matches on numbers with a successor case. `case 0:` handles zero; `case 1+d:` matches anything ≥ 1, binding `d` to one less (shadowing the argument). A branch body is an expression — no `return` anywhere. The arguments are `U32`, 32-bit unsigned integers — full 32-bit range.

WHAT IT COMPUTES

Each call owns a block of 2^d consecutive integers and splits it in half; at depth 0 the block is the single number `i`. So `sum(24, 0)` covers 0 through 16,777,215, for an exact total of 140,737,479,966,720 — which notably does not fit in a `U32`. Device selection (`!`) is a separate decision from parallelism; that is the GPU page.

C.43 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/04-gpu-concepts.md

--- title: Running on GPUs description: "One character picks the device. The exclamation mark chooses where, never how." updated: 2026-08-13 verified: 2026-09-05 status: docs-only canonical: /learn/gpu/ sources:

- https://bend2.dev/learn/gpu/ (https://bend2.dev/learn/gpu/) ---

Running on GPUs

Bend2 runs a function on the GPU when the call carries `!`: `sum!(24, 0)` evaluates on the GPU where `sum(24, 0)` would use the CPU. The function is the parallel sum, unchanged — only `main` differs. That is the entire GPU programming model.

```
def main() -> U32:
  sum!(24, 0)
```

WHAT THE MARK DOES

`!` chooses **where** a call evaluates, not **how** it parallelizes. The fork inside `sum` is ordinary divide-and-conquer recursion, identical on either device; the GPU just points its thousands of threads at the same graph. Kernels, grids, blocks, and device memory do not appear in the language — the one-line contrast with Mojo from the comparison.

WHAT STAYS ON THE CPU

Pure code goes through `!`; effects stay on the host. Hello world runs its input and print in `IO` on the CPU — what flies to the GPU here is arithmetic with no `IO` in its type.

THE BACKENDS

Metal and CUDA, built on HVM4. No public toolchain ships them yet, so this page carries no measured numbers; they arrive when a compiler does.

C.44 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/05-hvm-hands-on.md

--- title: "HVM4 hands-on" description: "Build the actual runtime with gcc and run superpositions, duplication, and fib — every output below was produced this way." updated: 2026-09-05 verified: 2026-09-05 status: runnable canonical: /learn/hvm-hands-on/ sources:

- https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)
- https://github.com/HigherOrderCO/HVM4/blob/main/docs/primer.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/primer.md)
- https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/syntax.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/syntax.md) ---

HVM4 hands-on

This page is different from the Bend2 pages: everything here runs today. HVM4 is public, one C file, and the outputs below were produced by building it and running each snippet (gcc works; the README uses clang).

```
curl -sSL -o hvm.c https://raw.githubusercontent.com/HigherOrderCO/HVM4/main/src/hvm.c
gcc -O2 -o hvm hvm.c   # a few warnings, builds fine
./hvm file.hvm -s      # -s shows interaction/heap stats
./hvm file.hvm -s -C10 # -C collapses superpositions into separate results
```

FIRST RUN: A FUNCTION OF TWO ARGUMENTS

Books are `@name = term` lines; application is `f(a, b)` — never `f a b` (whitespace application is a parse error). Lambdas use `λ`:

```
@add = λa. λb. (a + b)
@main = @add(1, 2)
//3
```

Actual run: `3`, 3 interactions, 23 heap nodes. Three interactions for `1 + 2` tells you the price model honestly: every step is graph surgery.

SUPERPOSITION: ONE LOCATION, TWO VALUES

```
@main = (&A{1,2} + 10)
```

Actual run (`-C10`): `11` then `12`, 4 interactions, 13 nodes. Both values flowed through the same addition, sharing everything except the choice. Two corrections to the README here, found by running: the superposition needs a **label** (`&A{…}`, not `&{…}` — bare `&{}` is erasure), and there must be no spaces inside the braces.

LABELS DECIDE: ANNIHILATE OR MULTIPLY

A duplication meeting a superposition with the **same** label extracts pairwise; with **different** labels it multiplies branches:

```
@main =
  !x&A = &A{1, 2};
  [x₀, x₁]
//[1,2]          -- one result, 1 interaction
```

```
@main =
  !x&A = &B{1, 2};
  [x₀, x₁]
//[1,1]
//[2,2]          -- two results, 5 interactions
```

Both verified by running. `!x&A = v; body` binds the two copies as `x₀` and `x₁` (subscripts required); `[a, b]` is list sugar.

AFFINITY: THE COMPILER THAT SAYS NO TWICE

Variables may be used at most once. This fails:

```
@bad = λx. (x + x)   // PARSE/AFFINITY ERROR: x used twice
```

The fix is a cloned binder — `&` tells the parser to insert duplication:

```
@square = λ&x. (x * x)
@main = @square(5)
//25              -- verified: 3 interactions
```

Style rule from the primer, worth memorizing: dups are the expensive primitive, so arrange definitions as case-trees that avoid clones, and never apply an inline match (`λ{...}(x)`).

FIB, WITH A STOPWATCH

```
@fib = λ{
  0: 0
  1: 1
  λ&n. (@fib((n - 1)) + @fib((n - 2)))
}
@main = @fib(10)
//55
```

Actual run: `55` , 760 interactions, 2512 nodes, 19.33M interactions/s on an ordinary CPU. `0:` / `1:` switch on machine ints; the default arm names its binder (`λ&n.` — cloned, since `n` is used twice).

PATTERN MATCHING THAT ACTUALLY PARSES

The primer's `<>:` list-cons pattern does **not** parse on current main (`expected: name`). Match the constructor explicitly:

```
@len = λ{
  []: 0
  #CON: λh. λt. (1 + @len(t))
}
@main = @len([1, 2, 3])
//3               -- verified: 16 interactions
```

`[]` is `#NIL{}` sugar and still works; Peano naturals use `Nn` sugar with `0n:` / `1n+:` arms. Where this guide deviates from upstream docs, the binary was the tiebreaker — every snippet above exited 0.

C.45 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/06-interaction-nets-notes.md

--- title: "Interaction nets, explained" description: "Copying is the hard problem in computing. Interaction nets tame it: local, incremental, consensus-free — from three symmetric symbols." updated: 2026-09-05 verified: 2026-09-05 status: stable canonical: /notes/interaction-nets/ sources:

- https://bend2.dev/notes/interaction-nets-explained/ (https://bend2.dev/notes/interaction-nets-explained/)
- https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6)
- https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus) ---

Interaction nets, explained

Every runtime problem that matters is a copying problem. Using a value twice means duplicating its computation or its data; throwing it away means reclaiming the memory. Conventional machines solve both with coordination — locks, reference counts, stop-the-world collectors — and coordination is what stops programs from scaling. Linear logic's great export was making this explicit: treat values as resources consumed exactly once, and copying becomes a first-class operation you can see, price, and control. Interaction nets (Lafont, POPL 1990) are the machine that takes that lesson literally.

The big insight: no consensus Ten thousand workers can rewrite one program with zero coordination — no locks, no tags, no collector — because the model makes conflicting steps *unrepresentable*. A step touches exactly two adjacent cells, so two steps never share a cell, so no interleaving can go wrong. Consensus isn't optimized away. It is unnecessary by construction.

WHY COORDINATION IS UNNECESSARY

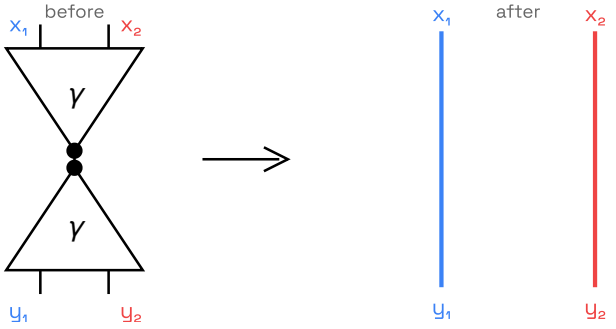
- **No coordination.** Redexes are disjoint before any analysis runs; any idle worker fires any pending pair. Parallelism is scheduling, not a language feature.
- **Copying is incremental and local.** Duplication copies one cell per step, interleaved with the computation consuming the copies — never a whole structure in one gulp. Shared subgraphs stay shared until use forces the split.
- **Erasure is garbage collection.** Throwing a value away is just another local rewrite (ε), with no collector thread pausing anything.
- **Every step is atomic.** Each rule completes in constant time with a clear physical mapping — unlike substitution, which can copy an unbounded term.
- **Perfectly symmetric.** No rule has an active side and a passive side: two cells meet as equals through their principal ports. The whole system is meetings, never commands.

- **Three symbols suffice.** γ, δ, ε under six rules compute anything — the smallest universal machine with all of the above intact.

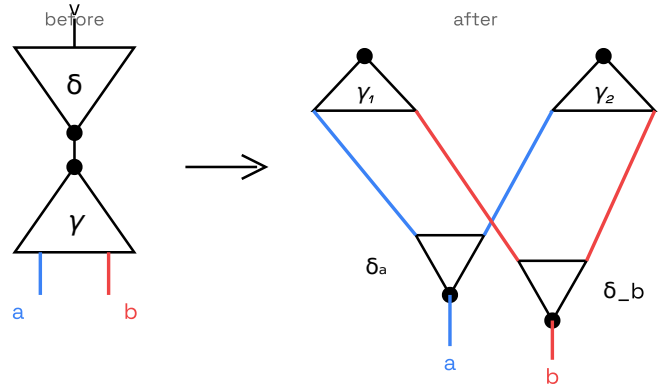
COPYING, ERASURE, AND CONFLUENCE

A program is a graph, and sharing means what it says: two parents point at one child. When both parents need the child, a duplicator (δ) walks it — one cell per interaction, copying each cell and moving on while the rest of the net keeps reducing around it. Nothing stops, nothing locks, and work common to both copies happens once, before the split. Contrast the alternatives: textual substitution duplicates unevaluated work blindly; memcpy-style cloning drags in unused substructure; reference counting serializes every touch. Here the copy *is* the computation, interleaved cell by cell.

The dual move is annihilation: when a symbol meets itself, both vanish and the wires join through — erasure and pairing handled by the same geometry. In the drawings below, matching colors mark the same wire continued across the step. The richer interaction between duplication and alternative values belongs to the advanced runtime chapter; first understand the local graph and its mathematical guarantees.



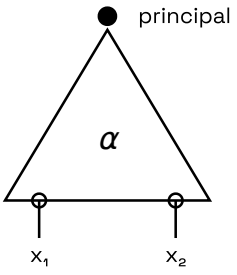
annihilate Same meets same: both cells vanish, and each auxiliary wire splices straight through to its counterpart — blue continues to blue, red to red. Nothing is copied, nothing observed.



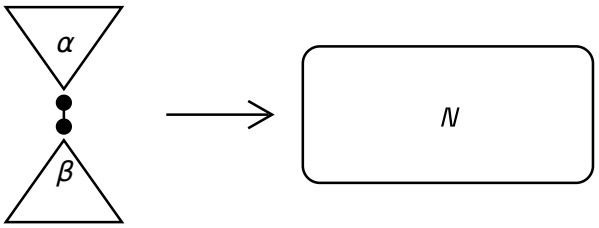
commute Distinct symbols pass through each other: γ is copied once per δ branch and δ once per γ wire. Follow the colors — each original wire (blue *a*, red *b*) gets its own duplicator, and each γ copy takes one copy from each.

THE MODEL IN ONE MINUTE

A program is a graph of **cells**. A cell is one constructor or operation — a number, a `+`, a list node, a λ. It has one **principal port** (drawn at the apex, the only place it can react) and any number of auxiliary ports where arguments and results wire in. Computation happens in exactly one configuration: two cells wired principal-port to principal-port, called an **active pair**. For each pair of symbols there is at most one rule, which consumes the two cells and writes a fixed replacement on the same free ports.



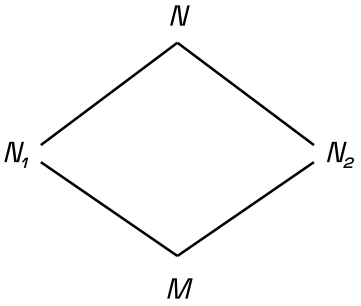
A cell: symbol α, one principal port at the apex, auxiliary ports below.



A rule firing: the active pair α ⋈ β is consumed and the fixed net N takes its place.

THE TWO THEOREMS THAT CARRY EVERYTHING

Confluence (1990). Fire two different active pairs in either order and both orders land on the same net in one step — the pairs share no cell, so firing one leaves the other intact.



diamond One-step confluence: firing a then b, or b then a, lands on the same net M.

Consequences: every order reaches the same result (no scheduler can be wrong), every route takes the same number of steps (work is a property of the program, so profiling counts mean something), and any idle worker may fire any pair it finds.

Universality from three symbols (1997). γ δ ε under six rules suffice for anything computable — and every rule is one of the two symmetric meetings above. That is the whole machine: meet, annihilate, commute, repeat — at any scale, in any order, with no one in charge.

THE Λ-CALCULUS, WITH FINE PRINT

Encode abstraction and application as cells and β-reduction becomes a single constant-time interaction. The difficulty moves to duplication: Lamping's POPL 1990 algorithm answered optimally (never duplicate work in Lévy's sense), and Asperti and Mairson proved in 1998 that full optimality's bookkeeping is unbounded — optimal counts β-steps but does not promise cheap accounting. The HVM line draws the engineering conclusion: duplication is a **primitive** of the calculus, not transparent substitution, trading Lamping-completeness for constant factors.

The mystery, solved Taelin's "mystery" post asked why some functions run *faster the more work you give them* — e.g. N repeated applications in O(log N). Answer: functions that **fuse under self-composition** (their composed normal form stays constant size) can be exponentiated by squaring, exactly like integers — *provided* the runtime normalizes inside lambdas, which only optimal evaluators do. A `copy` prefix can even *make* a function fuse, yielding apparent "negative complexity". Fusion at runtime is also why β-optimality gives Haskell-style deforestation for free.

IN THE MACHINE

A cell is a tagged machine word, an active pair is a queue entry, an interaction is a few loads and stores — none of it locked, because disjointness plus confluence keeps every schedule correct. The price is the constant factor: every step is graph surgery against real memory, and grinding it down is what the runtime generations are for. Numbers meet the same wall: a pure net would write 1000 as a thousand cells, so real runtimes store machine integers inside the node word, and the width left after ports and tags sets the range.

C.46 · `bend-gme/version-1/.opencode/skills/bend2-runtime/references/07-four-interactions.md`

--- title: "The four interactions" description: "HVM's whole dynamics in four rules, with reduction traces: optimal sharing, Church 2^2 in 14 steps, and collapse." updated: 2026-09-05 verified: 2026-09-05 status: stable canonical: /notes/four-interactions/ sources:

- https://github.com/HigherOrderCO/HVM4/blob/main/docs/theory/interaction_calculus.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/theory/interaction_calculus.md)
- https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/memory.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/memory.md) ---

The four interactions

The interaction-net model says rewrites are local and confluent. The Interaction Calculus says which four rewrites. Everything else in HVM — numbers, constructors, pattern matching — is practical extension. The theory below comes from HVM4's own `docs/theory/interaction_calculus.md` ; the traces are quoted from it.

WHY LAZY NEEDS TWO NEW FORMS

Strict evaluation computes everything now; lazy computes on demand but recomputes shared values used twice. Haskell memoizes thunks, which works for data and breaks inside lambdas. The calculus instead makes sharing structural, with two dual primitives (Lafont's fan nodes, opposite polarities):

- **Duplication** `!x&L = v; t` — one value in two locations (x_0 , x_1).
- **Superposition** `&L{a, b}` — two values in one location.

Duplication is incremental: cloning `[1,2,3]` peels one layer per step, and the tail's duplication only fires when both copies are touched. Work is never duplicated — even inside lambdas.

THE FOUR RULES

Two create computation, two propagate it. Superpositions are introduced here, after the language and basic runtime foundations:

SUPERPOSITIONS

Superposition `&L{a, b}` puts two values in one location. It is the runtime primitive behind branching search and shared evaluation; start with the HVM4 hands-on examples before treating it as a programming technique.

Two create computation, two propagate it:

1. **APP-LAM** — application eliminates lambda: `(λx.body)(arg)` binds `x ← arg in body` . Ordinary β.
2. **DUP-SUP** (same label) — duplication eliminates superposition: `!x&L = &L{a,b}; t` becomes $x_0 \leftarrow a$, $x_1 \leftarrow b$ in `t` . Pair projection.
3. **APP-SUP** — application propagates through superposition: `(&L{a,b})(c)` becomes `!x&L = c; &L{a(`x_0`), b(`x_1`)}` . Applying a pair distributes over both elements.
4. **DUP-LAM** — duplication propagates through lambda: `!f&L = λx.body` yields two lambdas with a **superposed bound variable**, `x ← &L{…}` , and the body shared behind a new dup. This is the deep one: the variable temporarily escapes its scope, which is exactly why the calculus needs global variables — and exactly how sharing extends inside lambdas.

With different labels, DUP-SUP commutes instead of annihilating, nesting one superposition inside the other (the cross product from the hands-on page, formalized).

OPTIMAL SHARING, ON PAPER

Duplicate a lambda containing `(2+2)` , apply the copies to different arguments — the addition computes **once** and the result `4` flows to both copies. The trace (abridged from the theory doc) pivots on one step: DUP-LAM shares the body behind `b` , OP2-NUM fires a single time marked "shared!", and both branches observe `4` . Memoization could never do this: the shared work sits inside a duplicated function body.

CHURCH 2^2 IN 14 INTERACTIONS

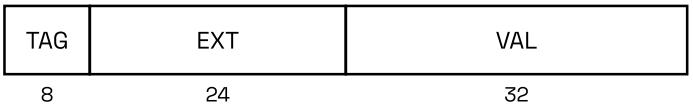
Applying Church-2 to itself is the literature's sharing benchmark. The full trace runs APP-SUP, DUP-LAM, and DUP-SUP (both modes) across a network where bound variables visibly escape scope mid-computation — then collapses to `λf.λx.f(f(f(f(x))))` . HVM completes it in **14 interactions**, which is optimal. The point for Bend2: this is the machinery that makes "the evaluator schedules" more than a slogan.

COLLAPSE: READING THE ANSWER OUT

Reduction has three destinations: WNF (head only), SNF (full, keeps SUPs/DUPs), CNF (collapsed back to pure lambda terms via DUP-VAR/DUP-APP cleanup). The `-C` flag selects collapse — the enumeration of `11`, `12` you saw hands-on. Proofs about programs are proved against SNF; humans read CNF.

ONE WORD PER TERM

The machine guest of honor: every term is one 64-bit word — `TAG (8) | EXT (24) | VAL (32)`. Tags name the node kind (APP, LAM, SUP, DUP, NUM, constructors by arity, MAT/SWI, OP2, REF…); EXT carries the dup label, op code, or binder level; VAL is a heap slot or an unboxed `u32`.



One HVM term: a single 64-bit word. Numbers live unboxed in VAL.

When a binder fires, its slot is overwritten with a substitution cell (SUB bit set) so linked variables resolve without side tables. Static book definitions stay immutable with de Bruijn levels; ALO nodes expand them lazily, one layer at a time, preserving sharing. That is the whole memory story — and why "a cell is a tagged machine word" from the [model](06-interaction-nets-notes.md) page is literal, not metaphor.

C.47 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/08-hvm-runtime-notes.md

--- title: The HVM4 runtime description: "The current runtime Bend2 builds on: Interaction Calculus, superpositions, and a single C file." updated: 2026-09-05 verified: 2026-09-05 status: stable canonical: /notes/hvm-runtime/ sources:

- https://github.com/HigherOrderCO/HVM4 `(https://github.com/HigherOrderCO/HVM4)`
- https://github.com/VictorTaelin/Interaction-Calculus `(https://github.com/VictorTaelin/Interaction-Calculus)` ---

The HVM4 runtime

If interaction nets are the idea, HVM4 is the machine: public, a single C file built with one compiler invocation, with its memory layout and interaction rules documented next to the code. It is the half of Bend2 you can read today — the language, the type checker, and SupGen remain private, but the calculus, the memory layout, and the rules are inspectable, and they constrain what the language above them can be: parallel by default, superposition-capable, compiled rather than interpreted.

Run it yourself Every claim below with a number next to it was reproduced by building HVM4 and running the snippet — see HVM4 hands-on. The repository carries no license text, so treat this as study, not a grant to build on it.

THE CALCULUS UNDERNEATH

HVM4 implements the Interaction Calculus: affine variables (each used at most once — which makes garbage collection cheap and parallelism simple), global lambdas with no scope boundaries (enabling continuations and linear encodings), and two first-class primitives most runtimes lack:

- **Superposition** `&A{a, b}`: one term holding two values. `(&A{1,2} + 10)` yields both `11` and `12` in 4 interactions, sharing all work that does not depend on the choice. This is how an evaluator explores many branches of a search space while paying for the common structure once.
- **Duplication** `!x&A = v; body`: one value feeding two uses (x_0 , x_1) without copying up front — the incremental copy described on the interaction nets page. Same label annihilates against a superposition; different labels multiply branches.

```
@main = (&A{1,2} + 10)
//11
//12
```

METAL AND CUDA BACKENDS

HVM4 ships both GPU backends. The CUDA port was produced overnight by a coding model working against the Metal implementation as executable specification and test oracle; its benchmark puts CUDA ahead of Metal on RTX hardware at roughly 10x parallel C for most programs.

WHAT THE AOT COMPILER CHANGES

An interpreted evaluator pays dispatch on every rewrite: fetch the node, branch on its kind, mutate the graph. Compiling a program's rules ahead of time to native C removes the dispatch and specializes memory access per rule — attacking exactly the constant factor that lets a tuned CPU core rival GPU throughput on many workloads. The AOT compiler was one of three items on the March 2026 launch-blocker list, alongside the GPU runtime (since resolved) and SupGen integration (still private).

9.1 RUNTIME MODEL

An HVM program is a **book** of named terms. A definition such as `@main = ...` is a reusable static term. Evaluation instantiates the term into a mutable graph, then repeatedly finds an active pair: two principal ports connected together.

Each interaction consumes a small local configuration and replaces it with another configuration. The evaluator does not walk the entire program to decide what is safe to run. Locality and confluence make that decision independent for every active pair.

```
clang -O2 -o hvm src/hvm.c
./hvm file.hvm -s       # interaction statistics
./hvm file.hvm -s -C10  # collapse alternatives
./hvm file.hvm -D       # print intermediate steps
```

9.2 SURFACE SYNTAX

The runtime language is deliberately smaller than Bend2. Definitions use `@name = term`; application uses parentheses, not whitespace application: `@add(1, 2)`, not `@add 1 2`.

```
@add = λa. λb. (a + b)
@main = @add(1, 2)
//3
```

Form	Meaning
`λx. body`	Lambda abstraction
`(f x)` or `f(x)`	Application
`#Pair{a,b}`	Constructor
`λ{#Pair: ...}`	Constructor match
`!x&A = value; body`	Explicit duplication with label `A`
`&A{left,right}`	Superposition with label `A`
x_0, x_1	The two branches of a duplicated value
`&{}`	Erasure
`@name`	Book reference

Variables are affine: an ordinary variable may be used at most once. When the same value must be used twice, write a cloned binder (`λ&x`) or an explicit duplication. This is what lets the evaluator represent copying as graph interaction rather than a global memory operation.

9.3 THE 64-BIT TERM

Every term is represented by one 64-bit word:

```
+--------+------------------------+--------------------------------+
| TAG 8  | EXT 24                 | VAL 32                         |
+--------+------------------------+--------------------------------+
```

`TAG` identifies the term kind. `EXT` carries metadata such as an operation code, constructor identity, duplication label, or binder level. `VAL` contains an immediate number or an index into the heap/book.

Dynamic terms live in the mutable heap; static definitions live in the immutable book. ALO allocation terms bridge them: a static term expands into a dynamic term only when evaluation forces that layer, preserving compact definitions and sharing.

9.4 EVALUATION AND COLLAPSE

HVM distinguishes the amount of a term it evaluates:

- **WNF** — weak normal form; expose the head shape.
- **SNF** — strong normal form; reduce the whole term while preserving superpositions and duplication nodes.
- **CNF** — collapsed normal form; enumerate ordinary readable results.

Collapsing is a presentation step, not the same thing as evaluation. A superposition is not automatically a list, and collapsing it too early can destroy the sharing that made the computation cheap.

9.5 GPU BOUNDARY

HVM4 has Metal and CUDA backends. The GPU does not change the interaction rules; it changes where independent graph work is scheduled. Bend2 keeps input/output on the host and sends pure computation to a device call marked with `!`.

Read next: The four interactions, HVM4 hands-on, and Search by superposition.

C.48 · bend-gme/version-1/.opencode/skills/bend2-runtime/references/09-optimal-computer.md

--- title: "Towards an optimal computer" description: "Why HOC exists: computability is settled, computation is not — and both Turing and Church may be distortions of something deeper." updated: 2026-09-05 verified: 2026-09-05 status: stable canonical: /notes/optimal-computer/ sources:

- https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b)
- https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1 (https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1) ---

Towards an optimal computer

This page distills the founding essay of the Higher Order Company. It is the "why" behind every runtime, language, and proof system in these notes — read it before anything else if you want the motive, not just the mechanism.

The one claim Turing machines and the λ-calculus are equivalent in *computability* — but a model can be inherently less efficient than another. Church–Turing says nothing about *computation*.

1936, TWICE

Humanity answered "what is computation" twice in the same year. Turing distilled the common components of early machines into one universal device — the ancestor of every processor, and, through the procedural mindset, of C, Fortran, Java, Python. Church, independently, distilled the common components of branches of mathematics into the λ-calculus — the ancestor of Haskell, Clojure, Agda. Both compute everything computable. The choice between them was treated as taste. Taelin's argument: that was the wrong question. The right one is which model wastes less.

WHAT EACH MODEL GETS WRONG

The procedural model's parallel primitives — mutexes, atomics — are a contrived solution to synchronization, so programmers still write sequentially by default; global state and loops make whole classes of bugs nearly unavoidable. The functional model handles parallelism and correctness far better in principle — yet functional programs stay mostly single-threaded. Why? Performance: the λ-calculus's fundamental operation, **substitution, is not atomic**. Substituting an argument can copy an unboundedly large term, so it cannot complete in bounded steps and has no physical mapping. Workarounds only move the problem — shared references inhibit parallelism, garbage collection isn't atomic. The functional paradigm never got its efficient machine, never went mainstream, and its proof tools never caught up.

ANNIHILATION AND COMMUTATION

In 1997 Lafont proposed the interaction combinators, where substitution breaks into two truly atomic laws: **commutation** (creates and copies information) and **annihilation** (observes and destroys it). Unlike SKI — whose K erases and whose S copies unboundedly large structures — every combinator step finishes in constant time with a clear physical mapping, and all of it is inherently parallel. Everything good in the other models survives; the pathologies mostly vanish. Both the Turing machine and the λ-calculus emulate efficiently onto combinators, while the reverse is not true — equivalent in computability, strictly more capable in computation. Perhaps, the essay suggests, machines and substitutions aren't fundamental at all, and some alien civilization does mathematics and computing entirely in annihilation and commutation.

Where this goes The atomic-step claim becomes the interaction-net model; the efficiency claim becomes the four interactions; the company built four runtimes and two languages on top. Each page tests one corner of this essay.

C.49 · `bend-gme/version-1/.opencode/skills/bend2-runtime/references/10-bend-vs-mojo.md`

--- title: Bend2 vs Mojo description: "Two opposite routes to fast GPU code: explicit kernels you schedule vs parallelism the evaluator finds." updated: 2026-08-30 verified: 2026-09-05 status: prerelease canonical: /notes/vs-mojo/ sources:

- https://bend2.dev/notes/bend2-vs-mojo/ `(https://bend2.dev/notes/bend2-vs-mojo/)` ---

Bend2 vs Mojo

Prerequisites

- Automatic parallelism — the fork both sides argue about
- Running on GPUs — what `!` means before comparing it to kernels

Both want high-level GPU code. They assign the scheduling job to opposite actors: in Mojo, **you** schedule; in Bend2, **the evaluator** does.

MOJO: YOU ARE THE SCHEDULER

Mojo (Modular, designed under Chris Lattner on MLIR, shipped and pre-1.0 with production users) gives explicit control: you write kernels against a grid/block/device-memory model, manage buffers and synchronization, use explicit SIMD types, and one source compiles to NVIDIA, AMD, and Apple targets. Ownership and borrowing govern values. That control is exactly what makes dense, regular, arithmetic-heavy workloads fast — performance there lives in tiling, layout, and occupancy, and no runtime discovers a tiling for you. If your sentence starts "I would have written CUDA", Mojo is today's answer for a Python-shaped codebase.

BEND2: THE EVALUATOR IS THE SCHEDULER

Bend2 programs are ordinary recursive functions with no kernels, no grids, no memory hierarchy in the programmer's model. Recursion over a tree fans out because the branches are disjoint subgraphs — not because anyone scheduled it. The entire device decision is one character: `sum!(24, 0)` runs on the GPU where `sum(24, 0)` would not. Each reduction step still moves graph nodes through memory — the constant factor every graph-reduction benchmark argues about.

WHERE EACH WINS

Mojo wins anywhere you could name the kernel: inference serving, image pipelines, dense linear algebra, HPC stencils — with full-width IEEE numerics and explicit SIMD today. Bend2's bet is everything irregular and recursive, where writing the kernel is the hard part and the evaluator can find work you never named. Until it ships, that side of the table is a design claim with a runtime (HVM4) behind it, not a benchmark you can run.

PART D · HVM5 Interaction Calculus skill (.opencode → hvm5-ic, verbatim)

The bend-gme skill written around the HVM5 spec set: lineage, the v2 README/review/overview and its calculus, runtime, compiler, code-as-data, islands, wire formats, trust model and plan map, plus the v1 calculus and the v1 'HVM4 execution' chapter (the one that explains how HVM4 reduces interaction terms and what an HVM5 evaluator inherits).

Source root: `bend-gme/version-1/.opencode/skills/hvm5-ic` · **16 files · 94.7 KB**

▼ In this part (file list)

- **D.1** · `bend-gme/version-1/.opencode/skills/hvm5-ic/SKILL.md`
- **D.2** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/01-lineage.md`
- **D.3** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/02-v2-readme.md`
- **D.4** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/03-v2-review.md`
- **D.5** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/04-v2-overview.md`
- **D.6** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/05-v2-calculus.md`
- **D.7** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/06-v2-runtime.md`
- **D.8** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/07-v2-compiler.md`
- **D.9** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/08-v2-code-as-data.md`
- **D.10** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/09-v2-islands.md`
- **D.11** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/10-v2-formats.md`
- **D.12** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/11-v2-trust.md`
- **D.13** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/12-v2-plan.md`
- **D.14** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/13-v2-map.md`
- **D.15** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/14-v1-calculus.md`
- **D.16** · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/15-v1-hvm4-execution.md`

D.1 · `bend-gme/version-1/.opencode/skills/hvm5-ic/SKILL.md`

HVM5: an Interaction-Calculus runtime in Bend 2

Code is an immutable, content-addressed book of IC terms; fork is one DUP, test is one SUP of books, merge is a three-way merge of hashes plus a CAS.

MODEL

```
t ::= x | λx.t | (t t) | &L{t,t} | !&L{x,y} = t; t | * | #K{t…} | ~t{#K: t; …} | n | (⊛ t t) | ?t{0: t; +: t} | @f | %io(op,t,λr.t)
```

- Variables are affine with **global scope** (IC, not λ-calculus); sharing is explicit DUP.
- Key rules: `APP-LAM` substitutes; `DUP-SUP` same label **annihilates**, different labels **commute**; `DUP-LAM` splits a lambda without copying its body; `MAT/SWI/OP2` push through SUP; `REF` instantiates a book entry; `IO` yields to the host. Typed clashes return `Stuck{loc}`.
- Five objects: Term, Book (`Map<Name, DefHash>`), Instance `(book, state, epoch, seq)`, Island (replicas, 2–3 IO nodes), Capability. Five operations: deploy, send, fork (`!&F{state, shadow} = state`), test (`(&T{@old,@new} state msg)` shares every step before the first changed definition), merge (`merge3(books)` → migrate? → CAS epoch) (`references/04-v2-overview.md`).
- Pipeline: Bend 2 → official frontend → `bend --emit-core` (one pinned upstream patch) → core → lower → Book, hashed once (`references/07-v2-compiler.md`).

STATUS (FROM `references/02-v2-readme.md`, 2026-10-23)

Milestone	State
M1 IC kernel	done, sequential: kernel rules 28/28, Church c2·c2 = c4
M2 compiler	done for the fragment; SHA-256 vectors pass
M3 code as data	done on one island (`tests/e2e_vault.bend` PASS)
M4 membrane	written, untested
M5 islands	in-process mesh PASS; TCP not started
M6 benchmarks, LAWS/PROOF	not started

Upstream fact: **official Bend 2 has no interaction-net backend**; v2 needs the one `--emit-core` patch.

Superposition and sharing only work on HVM5, and HVM5 does not exist yet as a usable runtime. This skill describes a design and a research prototype. Nothing in bend-gme may depend on it (ADR-031).

Observed here (2026-10-30, local bend 2.0.25): `tests/kernel_rules.bend` 28/28 ok (11 defs rely on unsafe or foreign code); `tests/e2e_vault.bend` **FAIL** (`state after drop: * want Vault{159, 7}`); `tests/e2e_cluster.bend` **FAIL**. The README's PASS rows are not reproduced on 2.0.25 (likely a version difference; unverified). Treat M3 and M5 as **claimed, not reproduced** until re-run on the version they were written for.

RULES (FROM `references/03-v2-review.md`, CORRECTIONS C1–C10)

1. The reducer is a heap, not a tree (C1). No wire crosses a machine; islands talk by messages (C2).
2. Fork is DUP of state, test is SUP of books, merge is CAS. Fork is **not** SUP on the live root (C3, C5).
3. Never hash a live net; hash books and checkpoints (C4).
4. State strong confluence and its limits; say where IC differs from λ-calculus (C6, C7).
5. Do not say "Bend 2 compiles to interaction nets" (C8). Do not say "isolation is proven" beyond the cap-graph model.
6. A semantic change goes in a new `versionN/` directory; old versions stay runnable as evidence.

WHAT TRANSFERS TO A NORMAL BEND APP (E.G. BEND-GME)

- **Transfers**: content-addressed immutable artifacts (BGB segments by SHA), three-way merge of versioned definitions, explicit A/B evaluation of old vs new strategy on shared features, CAS on a ref for concurrent writers, honest status tables.
- **Does not transfer** to the released JS target: real SUP/DUP sharing, O(1) fork of running state, superposed testing. Do these explicitly (`bend2-superposition`).

VERIFY (IN `/workspaces/typescript-node/bend-runtime`)

```
export BEND_NO_TELEMETRY=1
bend hvm5/version2/tests/kernel_rules.bend
bend hvm5/version2/tests/e2e_vault.bend
bend hvm5/version2/tests/e2e_cluster.bend
```

Run before relying on a status row; record the bend version you used.

REFERENCES

File	Covers	Open when
`references/02-v2-readme.md`	thesis, reading order, status	first
`references/03-v2-review.md`	honest review of v1, corrections C1–C10, "stop claiming"	before writing any claim
`references/04-v2-overview.md` … `references/11-v2-trust.md` (spec 00–07)	objects/ops, calculus, runtime, compiler, code as data, islands, formats, trust	implementing a part

File	Covers	Open when
`references/12-v2-plan.md`, `references/13-v2-map.md`	milestones and gates; concept → spec → module map	planning work
`references/14-v1-calculus.md`, `references/15-v1-hvm4-execution.md`	the v1 calculus and HVM4 execution notes	history of the rules
`references/01-lineage.md`	versions and read order	orientation
`../bend2-runtime/references/07-four-interactions.md`, `../bend2-runtime/references/06-interaction-nets-notes.md`	net theory	background

RELATED

`bend2-superposition` (language-level status) · `bend2-runtime` (BendRT reality) · `research-to-release` (this repo's method, HVM5 v1 maps as examples).

D.2 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/01-lineage.md

HVM5

This directory holds the HVM5 lineage. It is versioned because the runtime specification, the graph calculus, and the cluster protocol will change between versions. Do not edit an old version to make a new experiment pass.

```
hvm5/
  reference/
    hvm5-mini/       vendored pure-HVM5 evaluator written in Bend 2
  version1/
    spec/            normative HVM5 v1 + cluster contract specification
    plan/            implementation slices and gates
    maps/            goal, system, type, space, lifecycle, code, risk maps
    src/             runnable Bend 2 homoiconic DSL, cluster, and CLI
    tests/           Bend 2 unit, adversarial, and end-to-end tests
    bench/           reduction and cluster benchmarks
    scenarios/       executable ten-node simulation
    bend/            Bend 2 laws and proof gate
    contracts/       sample HVM5/Bend 2 contract terms
    legacy/          early v0 net model, lattice, and TS oracle, kept as evidence
```

VERSIONS

Version	Status	Meaning
`reference/hvm5-mini`	early draft	A port of `hvm5.c`; kept as reference and proof evidence, not v1
`version1`	active	Bend 2 homoiconic DSL, partitioned contract runtime, ten-node simulation

READ ORDER

1. `version1/spec/00-system.md` (source repo: `bend-runtime/hvm5/version1/spec/00-system.md`, not copied) — what is being built.
2. `version1/maps/00-goal.md` — the goal and observables.
3. `version1/spec/01-calculus.md` — the net calculus.
4. `version1/spec/02-homoiconic-code.md` (source repo: `bend-runtime/hvm5/version1/spec/02-homoiconic-code.md`, not copied) — code as data and fork/merge.
5. `version1/spec/03-types-isolation.md` (source repo: `bend-runtime/hvm5/version1/spec/03-types-isolation.md`, not copied) — capability types and isolation.
6. `version1/spec/04-subspaces.md` (source repo: `bend-runtime/hvm5/version1/spec/04-subspaces.md`, not copied) — partitioned nets and authority.
7. `version1/spec/06-lifecycle.md` (source repo: `bend-runtime/hvm5/version1/spec/06-lifecycle.md`, not copied) — deploy, fork, test, merge.
8. `version1/plan/00-plan.md` (source repo: `bend-runtime/hvm5/version1/plan/00-plan.md`, not copied) — how the implementation is staged.

The old `version1/spec/legacy/` and `version1/plan/legacy/` files are Draft 0. They are useful history, but they are not normative.

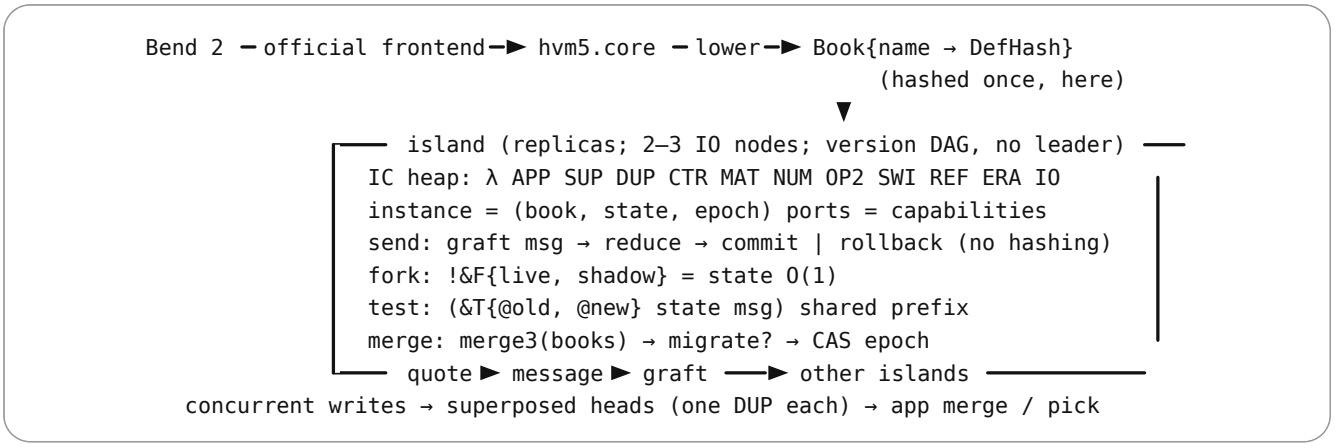
D.3 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/02-v2-readme.md

HVM5 v2 — forkable Bend 2 on islands of interaction nets

Thesis. Code is an immutable, content-addressed book of interaction-calculus terms. Then:

- forking a running program is one `DUP` node (O(1));
- testing a candidate against the live version runs both in one reduction, superposed with `&T{old, new}`, and shares every step before the first changed definition;
- merging is a three-way merge of definition hashes, followed by an atomic epoch swap.

Programs are written in Bend 2. They are checked by the official frontend and lowered by our backend. The kernel that reduces them is written in Bend 2 and compiled by BendRT. A cluster is a set of **islands**: local IC heaps, replicated on every node that holds a Replicate cap, with no leader, talking only through capability ports.



READ IN THIS ORDER

#	Document	Owns
—	`REVIEW.md`	honest review of v1, every correction, what not to claim
0	`docs/03-spec/00-overview.md`	the five objects, five operations, invariants, vocabulary
1	`docs/03-spec/01-calculus.md`	IC syntax, all rules, labels, fuel, confluence, readback, soundness boundary
2	`docs/03-spec/02-runtime.md`	heap kernel in Bend 2 on BendRT: layout, machine, rollback, IO, parallelism
3	`docs/03-spec/03-compiler.md`	`--emit-core`, lowering, linear matches, numbers, fragment, ABI
4	`docs/03-spec/04-code-as-data.md`	books, fork, superposed-book testing, merge3, promote, rollback
5	`docs/03-spec/05-islands.md`	leaderless version DAG, IO nodes, caps, subscriptions, failure
6	`docs/03-spec/06-formats.md`	`.ic` text, `Serial`, core bytes, log/checkpoint records, hash domains
7	`docs/03-spec/07-trust.md`	TCB, invariants, threats, what proofs do and do not cover
—	`docs/02-plan/00-plan.md`	milestones M1–M6, gates, risks, definition of done
—	`maps/00-map.md`	concept → spec → module, module dependency graph

STATUS (2026-10-23)

Milestone	State	Evidence
M0 v1 frozen	done	`../version1`
M1 IC kernel	done (sequential)	`src/kernel/*` ; `tests/kernel_rules.bend` 28/28; K2 Church c2·c2 = c4; K3 shown (same label gives 1, not 4)
M2 compiler	done for the fragment	`tools/emit-core.ts` (pinned patch, build-time only) → `src/compiler/{core,lower,book,hash}.bend` ; SHA-256 FIPS vectors pass
M3 code as data	done on one island	`tests/e2e_vault.bend` **PASS**: live branches Alice+Bob superposed, Carol rejected (Conflict), `merge_all` = hand merge (state and BookHash), broken branch auto-dropped
M4 membrane	checks, untested	`src/membrane/cap.bend`
M5 islands	in-process mesh done (DAG and world-label stores); TCP not started	`tests/e2e_cluster.bend` **PASS** (D1–D5): any arrival order → same heads + StateHash; IO node killed, nobody blocks; revive from a peer log; commutative book never forks; merge = hand oracle; `tests/e2e_worlds.bend` **PASS**: the same gates on one superposed state, plus per-pair independence (24 orders) and sequencer takeover
M6 benchmarks	not started	`bench/` empty; `LAWS.bend` / `PROOF.bend` not written

Islands have **no leader and no consensus** (`spec/05`). Each message names the versions it was written against. Concurrent conflicting messages become superposed heads, each one DUP on its parent. The application resolves them with the book's `merge` , and proven `law commutes` books never fork. 2–3 IO nodes per island write and do IO. Every replica reduces everything, and killing an IO node blocks nobody.

Upstream fact, stated plainly: **official Bend 2 has no interaction-net backend today.** v2 needs exactly one upstream addition (`bend --emit-core` , pinned as a patch in `tools/emit-core.ts`). Everything after the core file is ours.

D.4 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/03-v2-review.md

Review of the HVM5 draft (v1)

Written from the perspective of an interaction-calculus runtime designer (HVM lineage). The question is: does the draft get us to a cluster of partitioned inets running forkable, mergeable Bend 2 code? Every statement is either true of the v1 code today or labelled as a correction.

1. VERDICT

The goal is right, and it is unusual in a good way. Code as immutable terms, updates as term transformations, and an optimal confluent reducer underneath is a combination nobody ships today.

The draft has the right *nouns* and the wrong *machine*. Four of its central mechanisms do not work as written: the reducer, the shared net, fork, and hashing. Several documents describe a system that does not exist. Fixing this is not a rewrite of the idea. It is replacing the parts that fight IC with parts that use IC.

Area	v1 reality	Grade
Goal	coherent, novel	A
Honesty sections (`11-acceptance`)	mostly honest	B+
Compiler	first-order, coupled to compiler internals	C (prototype)
Reducer	tree + name substitution; DUP-LAM impossible	**F as an inet**
Partitioned net	cross-partition active pairs	**wrong model**

Area	v1 reality	Grade
Fork/merge	`SUP` on live root / record swap	**wrong primitive**
Hashing	per-op graph hash (spec), FNV-32 of `show` (code)	**wrong place**
Capabilities	booleans	placeholder
Cluster	one process, pure model	placeholder
Doc structure	18 normative files, overlapping	C

2. KEEP

1. **The deployed artifact is a term, not source.** The runtime never parses Bend.
2. **One frontend.** The official Bend 2 checker is the language authority.
3. **Kernel in Bend 2, executed by BendRT.** BendRT is a fine host as long as the net lives in an explicit heap (`Array<U64>`), not in Bend's own values.
4. **Hashes identify; capabilities authorize.** Never conflate them.
5. **Capabilities never enter terms.**
6. **Reject, never miscompile.**
7. **Failed operations change nothing.**

3. CORRECTIONS

C1 — THE REDUCER MUST BE A HEAP, NOT A TREE

`eval.bend` substitutes by binder name and relies on "globally unique names". The first duplication of a lambda breaks that invariant, because both copies carry the same names. That is why DUP-LAM and commuting labels are listed as "unsupported". They cannot be supported in this representation without deep copies, and deep copies throw away lazy, shared duplication, the only reason to use IC.

Do instead: an HVM4-style kernel. Nodes live in a linear `Array<U64>` . A term is a 64-bit tagged pointer. Variables are heap locations, so substitution is one write. All IC rules, including DUP-LAM and DUP-SUP with different labels. → `docs/03-spec/01-calculus.md` , `docs/03-spec/02-runtime.md` .

C2 — NO WIRE MAY CROSS A MACHINE

"One logical net with bridges between partitions" means some active pairs have agents on two machines. One interaction there costs a network round trip, about $10^4\times$ a local interaction. Worse, the v1 rule "pairs outside the listen set stay in the net" means a pair spanning two spaces is never reduced by anyone.

Do instead: islands. All wires are local. Islands exchange *values*: quote (read back a closed normal form), ship, graft. An island is an actor whose behaviour is an IC term. → `docs/03-spec/05-islands.md` .

C3 — FORK IS NOT `SUP` ON THE LIVE ROOT

Putting the live root under `&L{old, new}` makes every live send run in two worlds and makes the live answer depend on a collapse policy. That is not a fork.

Do instead:

- **fork** = `!&F{live, shadow} = state` . One node. Copying happens lazily, only where a side rewrites. The book is immutable and shared.
- **test** = run a *superposed book*: every definition whose hash differs becomes `@d = &T{@d_old, @d_new}` , and unchanged definitions stay single. One reduction serves both versions. Everything computed before control reaches a changed definition is computed **once**.
- **merge** = three-way merge of `Name → DefHash` , an optional `migrate` , then a compare-and-swap of `(book, state, epoch)` .

→ `docs/03-spec/04-code-as-data.md` .

Precision about the sharing claim: sharing covers the common prefix of the computation, the inputs, and unchanged state. Two *independently written* code paths that happen to compute the same thing are not shared. Report the measured `sharedRatio` , never a promised one.

C4 — NEVER HASH A LIVE NET

A net in mid-reduction has no canonical form. Heap addresses and redex order are scheduling artifacts, and canonical labelling of graphs is expensive. Per-op hashing is also exactly what made the local node slow.

Do instead:

- hash **code once**, at compile time, per definition (Merkle DAG, Unison-style);
- hash **state only at checkpoints**, after readback to a closed normal form, which has a canonical serialization;
- **never hash during reduction.** Replicas agree because of the ordered log and confluence, and checkpoints confirm it every K sends.

→ `docs/03-spec/05-islands.md` §5, `docs/03-spec/06-formats.md`.

C5 — "MERGE" IN V1 IS PROMOTION

Replacing live code with a candidate is a promotion. Merging two people's forks needs structure that survives compilation, and reduced nets lose definition boundaries.

Do instead: books are `Map<Name, DefHash>`. `merge3(base, a, b)` per name. A conflict means the same name changed differently on both sides, and it is resolved by writing a new definition in Bend 2. Type changes go through `migrate`. → `docs/03-spec/04-code-as-data.md` §6.

C6 — USE STRONG CONFLUENCE; STATE ITS LIMITS

IC is strongly confluent: the one-step relation has the diamond property, so all complete reductions have the same result and the same length. Use this:

- results do not depend on the scheduler;
- **fuel is deterministic** if it is defined on a canonical strategy (lazy sends) or on complete reduction (readback);
- replicas need no per-step comparison.

Limits that must be written down: it holds for the pure rules only. `IO`, host nondeterminism and `Stuck` redexes are boundaries.

C7 — IC IS NOT THE Λ-CALCULUS; SAY WHERE THEY DIFFER

Optimal reduction without brackets is incorrect for some terms: a lambda that is duplicated and then meets its own duplicator under the same label. The draft never mentions this. **Do instead:** a label discipline. Compiler-inserted dups on lambda-typed values get fresh labels per expansion. User-written `Sup` / `Dup` are banned in transitions. The runtime's fork and test labels come from a reserved range. First-order contract code is always safe. → `docs/03-spec/01-calculus.md` §7.

C8 — BEND 2 → INET IS NOT AN UPSTREAM FEATURE

`tools/emit-inet.ts` walks compiler-internal HOAS (`Book` / `HTerm`). It will break on upstream refactors and covers only the first-order fragment. **Do instead:** one small upstream addition, `bend --emit-core`, producing an erased first-order core with linearized matches. Delete `emit-inet.ts` the day it lands. → `docs/03-spec/03-compiler.md`.

C9 — NUMBERS

Unary `Nat` in IC would be catastrophically slow. **Do instead:** `NUM` / `OP2` / `SWI` on U32. `Nat.sub` saturates (as in Bend 2). Overflow on add/mul is `Stuck`, not a wrap, because silent wrap in contract code is a security bug.

C10 — DOCUMENT STRUCTURE

There are 18 normative files and the same concept appears in three places. **Do instead:** v2 has one calculus file, one runtime, one compiler, one code-as-data, one cluster, one formats and one trust file, and every gate has a status.

4. STOP CLAIMING

- "Bend 2 compiles to interaction nets." Upstream it does not.
- "The runtime is inet-native." Not until kernel gates K1–K4 pass.
- "One logical net across the cluster." Islands and messages.
- "Fork = DUP, merge = SUP collapse." Fork is `DUP` of state, test is `SUP` of books, merge is CAS.
- "Every operation is hash-quorum committed." Only checkpoints.
- "Isolation is proven." Only for the cap-graph model; not BendRT, the OS or the network.

5. WHY THIS DESIGN IS WORTH BUILDING

System	Has	Lacks
Unison	content-addressed defs, structural code merge	live instances, optimal sharing
Erlang/OTP	hot swap, old/new draining	code as data, deterministic replay
Ethereum	replicated deterministic state machines	cheap fork/test, code merge
HVM/IC	optimal sharing, confluence	distribution, code identity, authority
E / Spritely	object capabilities	optimal reduction, code merge

v2 combines all five properties. The one capability nobody else has is **testing a candidate against live code in a single shared reduction**. Measure it first (benchmark B5). If it does not pay off, fork/merge still works without it.

D.5 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/04-v2-overview.md

00 — Overview

Status: normative, HVM5 v2.

1. FIVE OBJECTS

Object	Definition	Mutable?
Term	an interaction-calculus term in an island heap (`01`)	consumed by reduction
Book	`Map<Name, DefHash>` plus a manifest. Each def is a closed term template	never
Instance	`(book: BookHash, state: Term, epoch: Nat, seq: Nat)`	only by sends and promotes
Island	a set of nodes, each a full replica (own heap) of its documents; 2–3 are IO nodes (`05`)	—
Capability	membrane record `(port, rights, mode, epoch)` behind a random token	revocable

2. FIVE OPERATIONS

```
deploy(book, init)        state := normalize(@init)                      new instance
send(cap, msg)            state', reply := (@transition state msg)       atomic, fuel-bounded
fork(inst)                !&F{state, shadow} = state                     one node
test(fork, msgs)          (@transition_T state msg) on superposed book T shared prefix
promote(inst, cand, e)    CAS epoch e; state := (@migrate state)?        atomic
```

`merge3` (three-way book merge) and `abandon` (erase the shadow) complete the set. All of them are term operations. None of them edits running code in place.

3. LAYERS

```
L6  tools      bend --emit-core · hvm5 CLI · signer
L5  registry   DefHash → Serial blobs · names → BookHash · authority rows
L4  islands    version DAG · writer chains · IO nodes · replicas · subscriptions (no leader)
L3  membrane   caps · rights · linearity · epochs · fuel & heap budgets
L2  instance   init · transition · migrate · fork · test · promote
L1  kernel     IC heap & rules · readback · graft · rollback   (Bend 2 on BendRT)
L0  host       BendRT binary · OS · TCP · disk · Ed25519 · BLAKE3
```

L0–L1 are trusted. L3 is the security boundary. Contract code is untrusted and fuel-bounded. L4–L6 are clients of L3.

4. INVARIANTS

1. **Local wires.** No heap pointer refers outside its island.
2. **No ambient authority.** A term can reach only its book and its cap handles.

3. **Hashes identify, capabilities authorize.**
4. **Books are immutable.** A new version is a new book.
5. **No hashing on the reduction path.** Only compile, checkpoint and records.
6. **Confluent core.** Pure results are scheduler-independent (`01` §6).
7. **Atomicity.** A send or promote commits fully or not at all.
8. **Linear authority is not copyable.**
9. **Declared boundaries.** `IO` , host numbers and the soundness boundary of IC (`01` §7) are explicit.

5. NON-GOALS

Consensus of any kind: v2 has no leader and no total order (`05` §2). Tokens or money. Cross-island atomic transactions (use sagas). Confidentiality from island operators. Lowering every Bend 2 feature (the fragment is listed in `03` §5). GPU (later; the kernel layout is compatible).

6. VOCABULARY

Word	Meaning
redex	an active pair: an eliminator meeting a value (`APP` /λ, `DUP` /x, `MAT` /#K, …)
interaction	one rule application; the unit of fuel
label	24-bit tag on `SUP` / `DUP` ; equal labels annihilate, different labels commute
quote / readback	heap term → canonical `Serial` of its normal form
graft	`Serial` → fresh heap term
superposed book	book where changed defs are `&T{old, new}`
projection π_i^T	readback picking side `i` of every `&T`
epoch	instance generation; incremented by promote
version	one DUP on a parent state; id `V{writer, seq}` (`05` §2)
heads	versions no other version names as parent: the live superposition
checkpoint	canonical heads + StateHash; compaction = restore(take)

D.6 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/05-v2-calculus.md

01 — The calculus

Status: normative, HVM5 v2. The interaction calculus (IC) with typed data, numbers, references and an IO boundary. Notation follows HVM4.

1. SYNTAX

```
t ::= x                      variable (affine: occurs at most once)
    | λx. t                  lambda
    | (t t)                  application
    | &L{t, t}               superposition, label L
    | !&L{x, y} = t; t       duplication, label L, binds x and y
    | *                      eraser
    | #K{t, …}               constructor K (arity from the book)
    | ~t {#K₁: t; …; #Kₙ: t}  match; arm i is a λ over the fields of Kᵢ
    | n                      U32 number
    | (⊛ t t)                binary op ⊛ ∈ {+ -ˢ × / % = ≠ < ≤ & | ^ << >>}
    | ?t {0: t; +: t}        number switch; `+` arm receives n-1
    | @f                     reference to book entry f
    | %io(op, t, λr. t)      IO request; the continuation receives the result
```

Scope is global. A variable may occur outside the body of the λ that binds it. This is IC, not λ-calculus. It is what lets DUP-LAM split a lambda into two without copying its body. Every variable is used at most once. Sharing is explicit `DUP` .

2. RULES

`x ← v` means "substitute v for x" (one heap write).

```
APP-LAM   (λx.f a)              → x ← a; f
APP-SUP   (&L{f,g} a)           → !&L{a₀,a₁} = a; &L{(f a₀), (g a₁)}
APP-ERA   (* a)                 → erase a; *

DUP-LAM   !&L{p,q} = λx.f       → p ← λx₀.g₀; q ← λx₁.g₁; x ← &L{x₀,x₁}; !&L{g₀,g₁} = f
DUP-SUP   !&L{p,q} = &L{a,b}    → p ← a; q ← b                                  (annihilate)
DUP-SUP'  !&L{p,q} = &M{a,b}    → p ← &M{a₀,b₀}; q ← &M{a₁,b₁};
                                    !&L{a₀,a₁} = a; !&L{b₀,b₁} = b                (commute, L≠M)
DUP-CTR   !&L{p,q} = #K{a…}     → p ← #K{a₀…}; q ← #K{a₁…}; ∀i !&L{aᵢ₀,aᵢ₁} = aᵢ
DUP-NUM   !&L{p,q} = n          → p ← n; q ← n
DUP-ERA   !&L{p,q} = *          → p ← *; q ← *

MAT-CTR   ~#Kᵢ{a…}{…}           → (armᵢ a…); erase other arms
MAT-SUP   ~&L{a,b}{arms}        → !&L{r₀,r₁} = arms; &L{~a{r₀}, ~b{r₁}}
MAT-ERA   ~*{arms}              → erase arms; *

SWI-NUM   ?n{z; s}              → n = 0 ? (z; erase s) : ((s n-1); erase z)
SWI-SUP   ?&L{a,b}{z; s}        → as MAT-SUP
SWI-ERA   ?*{z; s}              → erase z, s; *

OP2-NUM   (⊛ n m)               → n ⊛ m                (see §5 for overflow)
OP2-SUP₀  (⊛ &L{a,b} m)         → !&L{m₀,m₁} = m; &L{(⊛ a m₀), (⊛ b m₁)}
OP2-SUP₁  (⊛ n &L{a,b})         → &L{(⊛ n a), (⊛ n b)}  (n is a number; copying is free)
OP2-ERA   (⊛ * m) / (⊛ n *)     → erase other; *

REF       @f                    → instantiate(book[f])  (labels per §7)
IO        %io(op, v, k)         → Yield(op, v, k)       (stops reduction, see runtime §8)

erase λx.f  → x ← *; erase f      erase &L{a,b} → erase a; erase b
erase #K{…} → erase fields        erase n       → ·
```

Typed clash. `APP` on a number or constructor, `MAT` on a λ or a wrong-type constructor, `OP2` on a non-number: the redex is left in place and reduction returns `Stuck{loc}`. A checked program never produces one. Grafted foreign data can.

3. INTERACTION COUNT

Every rule above, including each erasure step and each `REF`, is one interaction. Fuel is interactions.

4. NORMAL FORMS AND READBACK

A term is **normal** when it contains no redex. A normal closed term is a tree of λ, `#K`, numbers, `*`, residual `&L{…}` and residual `!&L` bound to variables.

`quote(t)` normalizes `t` and serializes it (`06` §2): binders as de Bruijn levels, labels renumbered from 0 in order of first occurrence. Two α-equivalent normal forms have identical bytes.

Projection $\pi_i{}^{\mathsf{L}}(t)$: readback that replaces each `&L{a,b}` by side `i`, and each `!&L{x₀,x₁} = v` by $x_i \leftarrow v$, $x_{1-i} \leftarrow *$. This is how superposed test results are separated (`04` §5).

5. NUMBERS

U32. `+`, `×` and `<<` that overflow give `Stuck`; contracts must not wrap silently. $-^{\mathsf{s}}$ saturates at 0 (Bend 2 `Nat.sub`). `/` and `%` by zero give `Stuck`. Comparisons return `1` / `0`. The compiler wraps them into `Bool` constructors.

6. CONFLUENCE AND DETERMINISM

Theorem (strong confluence of IC). If $t \to u$ and $t \to v$ with $u \neq v$, there is w with $u \to w$ and $v \to w$, each in one step. Hence all complete reductions of `t` reach the same normal form in the same number of steps.

v2 extensions (`NUM`, `OP2`, `SWI`, `CTR`, `MAT`, `REF`) are local and non-overlapping, so the diamond property is preserved. This is proof obligation **L1** in `LAWS.bend`.

Consequences used by the system:

Use	Requires
results independent of scheduler	pure fragment (no `IO` pending, no `Stuck`)
fuel identical on all replicas	the canonical strategy (`02` §4) for sends; complete reduction for readback
parallel readback/checkpoint quote	complete reduction (count is order-invariant)

7. SOUNDNESS BOUNDARY: IC IS NOT THE Λ-CALCULUS

Optimal reduction without brackets gives wrong results when a lambda copied by a duplicator with label `L` later meets a `DUP` or `SUP` of the *same* label that came from a different duplication. The `&L{x₀,x₁}` created by DUP-LAM annihilates with the wrong partner.

Label discipline (normative):

1. **Compiler dups** (inserted for variables used ≥2 times) on values whose type is first-order data get a static label per `(DefHash, site)`. That is safe: DUP-CTR/NUM never create SUPs.
2. **Compiler dups on λ-typed values** get a **fresh label per REF expansion** from a per-island counter. Labels are renumbered from 0 at every checkpoint compaction. If the counter would overflow 24 bits, the kernel forces a checkpoint first.
3. **User `Sup` / `Dup` syntax** is rejected in contract books. Superposition is a runtime tool here, not a user feature.
4. **Runtime labels** come in two classes:
   - *rdata* `0xF00000–0xF7FFFF` : dups of data made by the runtime (fork `F` / `G`, version DUPs, the pair split `D`);
   - *rworld* `0xF80000–0xFFFFFF` : worlds (live-branch and test labels `T`).

   Counters: fresh = register 7, rdata = 8, rworld = 21 (`02` §2).
5. **Erase shortcut.** `!&L{x, *} = v → x ← v` is applied only for data-only labels (static, rdata). For fresh and world labels, `v` may hold a SUP of the same label that must still annihilate, so the dup stays and the erased side becomes garbage.
6. **DUP-REF.** A dup of a `REF` copies the reference (`!&L{a,b} = @f → a ← @f, b ← @f`).

Bend 2's kinds guarantee that multi-use binders are Data, hence first-order, so rule 1 covers every compiler dup.

Under 1–6, no unintended annihilation can occur in contract code. (Obligation **L2**.)

8. LAWS (`LAWS.BEND`)

Id	Law
L1	local diamond for every rule pair in §2
L2	label discipline ⇒ no cross-annihilation
L3	`quote` is canonical: α-equivalent normal forms give equal bytes
L4	`π₀ᵀ(run(superpose(A,B,T), x)) = run(A, x)` and `π₁ᵀ(…) = run(B, x)`
L5	`π₀ᶠ` of a forked state equals the original state
L6	`merge3` table (`04` §6) is commutative in its two branches

D.7 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/06-v2-runtime.md

02 — Runtime: the IC kernel in Bend 2 on BendRT

Status: normative, HVM5 v2.

1. HOST MODEL

BendRT compiles Bend 2 to a native program with a flat evaluator. It knows nothing about active pairs. The kernel is an ordinary Bend 2 program that owns an explicit heap and implements `01` on it. Bend 2 `Array` is single-owner and updated in place, which is exactly what a heap needs.

2. POINTER AND HEAP

```
Ptr  = two U32 words: hi = sub:1 | tag:7 | lab:24, lo = loc   (Bend 2 has no U64)
Heap = Array<U32>          single owner, threaded through the kernel
```

Tag	Nodes at `loc`	Notes
`VAR`	—	`loc` is a binder slot. If the slot holds a `SUB`, follow it
`SUB`	value	marks a filled slot (substitution done)
`ERA`	—	
`LAM`	[body]	the binder slot is `loc+0`, the body is stored in it until substituted
`APP`	[fn, arg]	
`SUP`	[a, b]	`lab` = label
`DP0` / `DP1`	[value] shared	the two projections of one `DUP`; `lab` = label
`CTR`	[f_1 … f_n]	`lab` = constructor id; arity from the book table
`MAT`	[value, arm_1 … arm_n]	`lab` = type id
`SWI`	[value, z, s]	
`NUM`	—	value inline in `loc`
`OP2`	[a, b]	`lab` = operator
`REF`	—	`loc` = def index into the island's book table
`IOR`	[op, arg, k]	IO request

Layout: registers in words 0..31 (7 fresh label, 8 rdata label, 21 world label, 22 its tx copy, …), book table at 16, trash at 65552, nodes from 66576.

Allocation is a bump pointer only. **There are no free lists.** The write log grows down from the top, and rollback is bit-identical. Garbage (erased sides, pruned versions) is reclaimed by compaction at checkpoints: quote every live instance, reset the heap, graft back (`island/checkpoint.bend`, compact = restore(take)). There is no GC.

3. BOOK TABLE

Each island keeps `DefIndex → (DefHash, Template)`. A template is the def's `Serial` pre-decoded into a node array with relative locations. `instantiate` copies it to fresh heap space, relocates pointers and assigns labels per `01` §7. One `REF` expansion is one interaction, plus copying work proportional to the template size. That copying is not counted as fuel, but it is counted against the heap budget.

4. MACHINE

- `whnf(heap, p)`: an explicit stack of eliminator frames (`APP`, `MAT`, `SWI`, `OP2`, `DP0/DP1`). Pop a frame when the head is a value and apply the rule. This is the **canonical strategy**: leftmost-outermost, call-by-need through `DUP`. Its interaction count defines fuel for sends.
- `normalize(heap, p)`: `whnf`, then recurse into children. Used by `quote`. Complete reduction, so the count is order-invariant and may run in parallel (§9).
- A send reduces `(@transition state msg)` to WHNF, then forces the `Pair{state', reply}` constructor to WHNF. State may stay lazy between sends. Reply is normalized (it leaves the island).

5. BUDGETS

Per send: `fuel` (interactions) and `heap` (nodes allocated). Exceeding either gives `OutOfFuel` or `OutOfHeap`, and the send rolls back.

6. TRANSACTIONS (ROLLBACK WITHOUT COPYING)

Every heap write in a send appends `(loc, old)` to a **write log**. The allocation pointer is saved at start. `Done` discards the log. `Stuck`, `OutOfFuel`, `OutOfHeap` or `Denied` replays the log in reverse and resets the pointer. Cost is O(writes). The pre-send heap is restored bit for bit (gate K6).

7. QUOTE AND GRAFT

- `quote(p)` : `normalize` , then a pre-order walk emitting `06` §2 bytes. It fails if the term contains `IOR` or `Stuck` .
- `graft(bytes)` : decode, allocate, link, and return the root pointer. The decoder is total: malformed bytes give `Denied` .
- Messages crossing islands also carry `REF` s by **DefHash**. Before grafting, the receiver resolves missing hashes from the registry (`05` §3).

8. IO

When reduction hits `IOR(op, v, k)` , the machine returns `Yield{op, v, k}` . The send rolls back and returns NeedIO (4) with the op in the receipt's `note` . Only an **IO node** (the envelope's writer, `05` §3) asks the host and records the answer in the envelope's `ios` , then runs again. Replicas apply the envelope with the recorded answers and never call the host. Transitions without `IOR` are marked `pure` by the compiler, and only pure transitions can be SUP-tested.

9. PARALLELISM

`normalize` forks children with disjoint subheaps (BendRT `fork`), merging allocation through per-worker arenas. Strong confluence gives the same normal form and the same count. The sequential kernel ships first. The parallel scheduler is enabled only after gate K4 passes on the corpus.

10. MODULES

```
src/kernel/ptr.bend      tags, encode/decode
src/kernel/heap.bend     registers, bump alloc, write log, budgets, label counters
src/kernel/book.bend     book table, instantiate, label assignment
src/kernel/rules.bend    every rule in 01 §2
src/kernel/whnf.bend     canonical strategy, fuel
src/kernel/norm.bend     normalize (sequential, then parallel)
src/kernel/quote.bend    quote / graft / projection πᵢᴸ
```

11. GATES

Gate	Check
K1	each rule in `01` §2 has a unit test
K2	DUP-LAM corpus (Church numerals 2^{2^2} , `λf.λx.f (f x)` duplicated) matches HVM4 normal forms
K3	label discipline: a corpus that breaks with static lambda labels passes with fresh ones
K4	fuel: canonical strategy is deterministic; complete reduction gives the same count sequential vs parallel
K5	quote canonical: α-variants give equal bytes
K6	rollback: failed send leaves a bit-identical heap
K7	conformance: corpus results equal BendRT running the same Bend 2 source
K8	native build reports interactions/s on B1; target within 3× of HVM4 C single-thread (measured, not assumed)

D.8 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/07-v2-compiler.md

03 — Compiler: Bend 2 → IC

Status: normative, HVM5 v2.

1. BASELINE (TRUE TODAY)

Official Bend 2 compiles to BendRT (C/JS). It has **no** inet backend. HVM5 owns a backend and must not own a frontend.

```
contract.bend
  ─ official frontend: layout · modules · dependent types · quantities · totality · laws
      ─ bend --emit-core → hvm5.core (only upstream change)
          ─ src/compiler/lower.bend → Book (ours, Bend 2)
```

`../version1/tools/emit-inet.ts` (which reads compiler internals) stays as a migration aid. Delete it when `--emit-core` lands (gate C7).

2. CORE (`HVM5.CORE`)

Erased, first-order, checked. Byte format in `06` §3.

```
Module = { format, compiler, types: [Type], defs: [Def], exports }
Type   = { name, ctors: [{ name, arity }] }       ctor id = canonical index
Def    = { name, arity, pure: Bool, body: Expr }
Expr   = Var(i) | Ref(name) | Lam(Expr) | App(Expr, Expr)
       | Ctr(type, ctor, [Expr])
       | Mat(type, Expr, [Arm]) , Arm = { fields: n, body: Expr }   all ctors, in order
       | Num(u32) | Op2(op, Expr, Expr) | Swi(Expr, Expr, Expr)
       | Let(Expr, Expr) | Era | Io(op, Expr, Expr)
```

The emitter (upstream side) must:

1. emit only checked modules, rejecting holes, open laws and `@unsafe` in exported transitions;
2. erase types, proofs, `&0` arguments, `Eq` / `Rfl` / `Rwt` evidence;
3. compile nested and dependent matches to one-level total `Mat` s;
4. map `Nat` to `Num` / `Op2` / `Swi` (§4) and `String` to lists of U32;
5. be deterministic: same source and version give the same bytes.

3. LOWERING (OURS)

`lower.bend : Module → Book`.

1. **Linearize matches.** Free variables of a `Mat` are abstracted into every arm and passed after the match: `~x{#A: λa. f(a, y); #B: g(y)}` becomes `(~x{#A: λa. λy. f(a,y); #B: λy. g(y)} y)`. A variable used in several arms is therefore *not* duplicated. This is the standard HVM technique and the biggest constant-factor win in contract code.
2. **Affinity.** After step 1, count occurrences. Zero uses become `*` (erase). Two or more uses build a balanced tree of `DUP` s with labels per `01` §7: static for first-order types, fresh-per-expansion for λ-typed values. The core carries a `ho` bit per binder, set by the emitter from the checked type.
3. **Constructors and matches.** `Ctr` becomes `#K{…}` with a numeric id and `Mat` becomes `~`, both with ids from `Type`.
4. **References.** `Ref(name)` becomes `@index`. The book maps index → DefHash.
5. **Let.** A linear let substitutes directly. A non-linear let goes through the affinity rule.
6. **Reject.** `Sup` / `Dup` written in source (`01` §7.3), arrays, maps and FFI. Each rejection carries the source span.

COUNTER, LOWERED

```
@init       = #Counter{0}
@transition = λs λm ~s{
  #Counter: λv ~m{
    #Inc:   λby λv #Counter{(+ v by)}
    #Reset: λv  (erase v) #Counter{0}
  } v
}
```

(`v` is passed through the inner match by linearization. The `Reset` arm erases it.)

4. NUMBERS

Bend 2	Core	Semantics
`Nat.add`, `Nat.mul`	`Op2(+)`, `Op2(×)`	overflow → `Stuck`
`Nat.sub`	`Op2(-ˢ)`	saturating, as Bend 2

Bend 2	Core	Semantics
`Nat.div` , `Nat.mod`	`Op2(/)` , `Op2(%)`	by zero → `Stuck`
`Nat.eq/lt/le`	`Op2` → `Bool` ctor	
`match n: 0n / 1n+p`	`Swi`	
`n` literal	`Num`	must fit U32 or compile error

5. SUPPORTED FRAGMENT

Feature	v2
ADTs of any arity, nested/dependent match (after erasure)	yes
recursion, mutual recursion	yes
higher-order functions, closures	yes, with λ-label discipline
`Nat` , `Bool` , strings as lists	yes
laws, proofs, `Eq` , rewrites	erased
declared IO ops	yes, marks the def impure
user `Sup` / `Dup`	rejected in contract books
`Array` , `Map` , in-place update	rejected (v3: linear vector nodes)
`@unsafe` , FFI	rejected in exports

6. CONTENT ADDRESSING (WHERE CODE IS HASHED)

```
DefHash  = H("hvm5.def.2\0"  ‖ Serial(template) ‖ DefHash(ref₁) ‖ … )
SCCHash  = H("hvm5.scc.2\0"  ‖ templates of the strongly connected component)   for mutual recursion
BookHash = H("hvm5.book.2\0" ‖ sorted (name, DefHash) ‖ manifest)
```

Computed once, at compile time or registration. Renaming does not change a `DefHash` . `H` = BLAKE3 by default (SHA-256 for interop), fixed per cluster.

7. CONTRACT ABI

```
type State …
type Msg …
type Reply …                                  # optional
def init() -> State
def transition(s: State, m: Msg) -> Pair(State, Reply)    # or -> State
def view(s: State) -> View                    # optional: what subscribers see
def migrate(old: OldState) -> State           # optional: required when State changes
def merge(base: State, a: State, b: State) -> State   # optional: resolves concurrent versions (05 §2.4)
law commutes:                                 # optional, must be proven
  for s: State
  for x: Msg
  for y: Msg
  {next(next(s, x), y) == next(next(s, y), x) : State}
```

`emit-core` emits `merge` as an ordinary def. A filled `law commutes` (checked by the official frontend's `book_valid`) is recorded as the marker def `~commutes` (arity 0, body ERA). Being a def, it is covered by the BookHash. A `law commutes` with no proof is an error. `def independent(x: Msg, y: Msg) -> Bool` is admitted only with a filled `law independent_ok` (`05` §2.7); the frontend rejects a false claim (`contracts/kv.bend`).

Open bug. `Bool.pick(U32, c, y, x)` inside `merge` read back as erased fields (`Kv{*, *}`). The same code with an explicit `match` is correct (`contracts/kv.bend` uses that). The lowering or the erase rule for `Bool.pick` needs a kernel test. Example: `contracts/flags.bend` (a proven grow-only flag set) and `contracts/vault/merge.bend` (delta merge `a + b −`

base).

Manifest: `{ state, msg, reply, view?, entry: {init, transition, migrate?}, pure }` , plus the name-addressed optional defs `merge` and `~commutes` .

8. GATES

Gate	Check
C1	`--emit-core` (upstream or pinned patch) gives deterministic bytes
C2	`lower` covers every `Expr`
C3	corpus (counter, vault with `Nat.sub` , order book with 4-field ctors, list ops, closures, nested match) equals BendRT results
C4	each unsupported feature gives an error with a span
C5	same source twice gives the same `BookHash`
C6	rename a def: same `DefHash`
C7	`emit-inet.ts` deleted

D.9 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/08-v2-code-as-data.md

04 — Code as data: fork, test, merge

Status: normative, HVM5 v2.

1. TWO LEVELS

Level	Representation	Used for
Book	`Map<Name, DefHash>` + manifest	identity, diff, merge, patches, rollback
Term	IC term in a heap	running, forking state, superposed testing

Merging happens on books. A reduced net no longer has definition boundaries.

Homoiconic loop. Each def's core is available as a Bend 2 value of type `Core.Expr` (`06` §3). A Bend 2 program holding the `Registry` right can read it, transform it (refactor, instrument, optimize, patch), and submit the result, which the registry lowers and hashes. Code is ordinary data for code.

```
def add_fee(t: Core.Expr) -> Core.Expr   # an ordinary Bend 2 function over code
```

2. INSTANCE

```
Instance = { id, book: BookHash, state: Ptr, epoch, seq, forks: [ForkId] }
```

3. FORK — ONE NODE

```
fork(inst) -> Fork
  F = fresh runtime label                      (reserved range, 01 §7)
  !&F{s₀, s₁} = inst.state
  inst.state  = s₀                             live continues on s₀
  Fork{ base: (inst.book, inst.seq), book: inst.book, state: s₁, label: F }
```

Cost: one `DUP` node. A side copies a subterm only when its own reduction forces `DP0` / `DP1` through it. State the live side never rewrites is shared forever. The fork is a private instance with its own ports, held by the forker.

4. EDIT

```
patch = { set: [(name, DefHash)], remove: [name], migrate?: DefHash }
fork.book := apply(fork.book, patch)            new BookHash; old book untouched
```

5. TEST — SUPERPOSED BOOK

Given base book `A` and candidate `B`, define for a fresh test label `T`:

```
superpose(A, B, T)[n] =
  A[n]                        if A[n] == B[n]          (same DefHash: one copy)
  &T{A[n], B[n]}              if both exist and differ
  B[n] / A[n]                 if only one side has n   (reachable only from that side)
```

Run one message on both versions at once:

```
r = normalize((@transition state msg))          under superpose(A, B, T)
old = π₀ᵀ(r)      new = π₁ᵀ(r)                  01 §4
```

Why this is the point of using IC:

- control flows through unchanged definitions **once**;
- at the first reference to a changed def, `APP-SUP` / `MAT-SUP` split the computation, and `DUP` copies only the continuation that depends on the result;
- inputs (`msg`, the state) are duplicated lazily, so untouched parts are never copied.

Exact claim (law L4): `π₀ᵀ(r) = run(A)` and `π₁ᵀ(r) = run(B)`. The amount of sharing is a measured quantity, not a promise:

```
sharedRatio = 1 − fuel(r) / (fuel(run A) + fuel(run B))
```

N candidates. Nest labels `T₁…T_k` to test 2^k variant combinations of k changed defs in one run. Combinatorial A/B with shared work.

Replay test.

```
test(fork, inputs) → Report{ diffs: [(i, old, new)], fuel, sharedRatio, laws }
  s := fork.state
  for m in inputs: r := (@transition s m) on superpose; record π₀, π₁; s := π₁ᵀ(state part)
```

Rules:

- Only `pure` transitions can be SUP-tested. Impure ones are tested by plain replay against recorded IO results.
- If `State` changed, test `&T{(t_A s m), (t_B (@migrate s) m)}`.
- `laws` runs the book's declared `LAWS` defs on the candidate.

6. MERGE — THREE-WAY ON BOOKS

```
merge3(O, A, B)[n]:
  a == b        → a
  a == o        → b
  b == o        → a
  otherwise     → Conflict(n, o, a, b)
manifest (State/Msg types): same rule; a State change needs migrate in the result
```

Equality is `DefHash` equality: exact, O(1). Resolve a conflict by writing a new Bend 2 def. Commutative in `A` / `B` (law L6).

7. PROMOTE — ATOMIC

```
promote(inst, cand, mode, expectedEpoch):
  require inst.epoch == expectedEpoch                    CAS
  require Update capability (linear), consumed
  mode = swap   (default, Erlang semantics):
           state' = cand.migrate ? normalize((@migrate inst.state)) : inst.state
  mode = replay (strict): state' = fold cand.transition over log[fork.base.seq+1 .. now]
                          starting from the fork-time state
  inst := { book: cand.book, state: state', epoch: epoch+1 }
  log:   Promote{inst, oldBook, newBook, epoch, mode}
  caps:  old epoch → Draining (deadline) → Dead
```

Sends ordered before the promote entry used the old book. Sends after it use the new one. A send tagged with a stale epoch is rejected (`StaleEpoch`). Nothing is half applied.

Rollback = promote back to the old book (which is still in the registry), with a reverse migrate if the type changed.

8. ABANDON

`erase` the shadow s_1. The `DUP` then resolves to the live side, so no live work is lost. Drop the fork's ports. The live instance is unchanged (gate F7).

9. EXAMPLE: FIXING A VAULT, WHILE BOB ALSO EDITS IT

```
type State is Data:
  Vault{balance: Nat, owner: Nat}
type Msg is Data:
  Deposit{amount: Nat}
  Withdraw{amount: Nat, who: Nat}

def transition(s: State, m: Msg) -> State:     # book A (live): no owner check
  match s:
    case Vault{balance, owner}:
      match m:
        case Deposit{amount}:     Vault{Nat.add(balance, amount), owner}
        case Withdraw{amount, who}: Vault{Nat.sub(balance, amount), owner}
```

Alice forks and edits only `transition` :

```
        case Withdraw{amount, who}:
          match Nat.eq(who, owner):
            case True{}:  Vault{Nat.sub(balance, amount), owner}
            case False{}: Vault{balance, owner}
```

```
hvm5 fork    vault#1                      → f1  (1 node)
hvm5 edit    f1 --set transition=#9c1…    → book A'
hvm5 test    f1 --replay last:1000        → diffs: 3 foreign Withdraws; sharedRatio measured
```

Bob forks the same base, adds `def fee(a: Nat) -> Nat` and changes the `Deposit` arm. That also changes `transition` , so his book is `B` .

```
hvm5 merge3 base=A a=A' b=B
  fee         → take B      (only Bob)
  transition  → Conflict    (both changed)
hvm5 edit    m --set transition=#d40…     # Alice writes the combined transition in Bend 2
hvm5 test    m --replay last:1000
hvm5 promote vault#1 m --epoch 4          → epoch 5; old caps draining
```

10. GATES

Gate	Check
F1	fork allocates O(1) nodes, independent of state size
F2	untouched state subtrees are never copied (allocation counter)
F3	L4: projections of a superposed run equal separate runs (corpus)

Gate	Check
F4	`sharedRatio` reported on B5; > 0 on the prefix-heavy corpus
F5	`merge3` table and type-conflict tests; L6
F6	promote CAS: stale epoch rejected; swap and replay modes; no lost live sends
F7	abandon leaves the live heap observationally unchanged
F8	rollback to the old book reproduces old behaviour

D.10 · `bend-gme/version-1/.opencode/skills/hvm5-ic/references/09-v2-islands.md`

05 — Islands: the leaderless version DAG, IO nodes, capabilities

Status: normative, HVM5 v2. Supersedes the leader-ordered log (`attic/05-islands-leader.md` (source repo: `bend-runtime/hvm5/version2/spec/attic/05-islands-leader.md`, not copied)).

1. WHY ISLANDS

A local interaction costs nanoseconds. A network round trip costs 10^4–10^5 times more. So:

- **every pointer is island-local.** Wires never leave a node;
- nodes exchange **envelopes** (messages as values), never heaps.

```
Island = { id, book table, documents, membrane, nodes }
Node   = { full replica: own heap + version DAG per document }   (island/cluster.bend)
```

Any node holding a `Replicate` cap is a replica and reduces everything. 2–3 of them are **IO nodes** (writers). Reduction needs no node in particular: confluence (`01` §6) makes it the same everywhere. Only IO is tied to a node.

2. NO LEADER, NO CONSENSUS: ORDER LIVES IN THE DATA

Confluence makes one reduction deterministic. It does not make different messages commute. v2 never agrees on a total order. Instead it keeps the **partial order** messages carry themselves:

```
Env{ w, s, parents: [Vid], body }       Vid = V{w, s}   (writer, its own seq)
body = BSend{msg, ios} | BMerge{base} | BPick
```

Relation of A and B	How it is handled	Agreement
B's parents include A (causal)	B waits for A; it applies on A's version	none
concurrent, book proves `law commutes`	applied in arrival order to the one head	none
concurrent, may conflict	both become versions: superposed heads	none, the application resolves later

2.1 VERSIONS ARE DUPS

Each new version is **one DUP node** on its parent's state, the O(1) fork:

```
send   parent p:       !&D{p', v} = state(p);  v ← (@transition v msg)
merge  [a, b], base o:  v ← (@merge o' a' b')        lazy; forced by the next send or read
pick   [x, …]:          v ← x'
```

The parent stays, so a concurrent sibling forks from the same state, and siblings share every subterm neither has forced. The **heads** are the versions no other version names as a parent: a live superposition. A failed send (Stuck, fuel, heap) still makes a version, equal to its parent, so children can attach deterministically.

2.2 WHY REPLICAS AGREE WITHOUT TALKING

A version's state depends only on its parent's state and its body. The heads are "every version not named as a parent". So two replicas holding the **same set** of envelopes have the same heads with the same states, whatever order the envelopes arrived in, duplicates included (gate D1). The rules:

- **causal buffer:** an envelope whose parents (and merge base) are not all known waits. Each application drains the buffer again;
- **dedup:** a known `V{w, s}` is dropped (retries, gossip echoes);
- **canonical order:** heads are kept sorted by `(w, s)`.

No hashing happens on this path.

2.3 COMMUTATIVE BOOKS

If the book carries a proven `law commutes` (`03` §7), envelopes apply in arrival order to a single version: no fork, no buffer (gate D3). The proof is checked by the official Bend 2 frontend at compile time. The runtime trusts the `~commutes` marker exactly as far as it trusts the compiler (`07`).

2.4 RESOLVING

The application decides, at a time it chooses, by writing an envelope:

- `BMerge{base}` with parents `[a, b]` runs the book's own `merge(base, a, b)`, the state-level twin of `merge3` for code (`04` §6). Without a `merge` def it is Denied (5) and behaves as a pick of `a`;
- `BPick` with parents `[x, …]` keeps `x`'s state (one DUP).

Both are ordinary versions, so concurrent resolutions are themselves just more heads.

2.5 FAST PATH: AN OPTIONAL SEQUENCER

A sequencer (as in `bend-game`) is just one writer whose envelopes everyone builds on. While it is up there is one head. When it is down or slow, other writers keep writing: their envelopes become concurrent heads, and nothing waits. A replacement sequencer (any node with the cap) resolves the open heads first. Split brain is harmless: two sequencers make two heads. The sequencer stamp itself is not implemented yet (§9).

2.6 WORLD LABELS: THE INET-NATIVE STORE (`ISLAND/WORLDS.BEND`)

The same delivery rules (§2.2) and the same heads (§2.3), with one superposed state per document instead of one DUP-forked instance per version:

```
message v:  !&Lv{s0, s1} = S;  S ← &Lv{ ~fst(@transition s0 m), s1 }
merge v:    S ← &Lv{ @merge(π_o s0, π_l s0, π_r s0), s1 }
pick v:     S ← &Lv{ π_l s0, s1 }
```

- Each message is applied **once**, at delivery, lazily, in every world at once: one world label, one DUP, one SUP and three app nodes. Side 0 of `Lv` is "v happened", side 1 is "v did not".
- A version is a set of messages, i.e. a world. Reading it installs one collapse dup per label (in the set → 0, else 1) and reduces only that world. Labels of independent messages commute, so worlds that never diverge share their work.
- The host keeps only what the net cannot know: dedup, the causal buffer, which sets are heads, and vid → label.
- `settle` (one head left, at an acknowledged frontier) collapses every label permanently and materializes the state.
- Limits: pure books only (a lazily applied message cannot replay recorded IO; IO books use the DAG store); reads are rolled back, so until `settle` a read redoes the lazy work of its world.

Both stores pass the same gates (`tests/e2e_cluster.bend` DAG, `tests/e2e_worlds.bend` worlds) with identical heads and StateHashes.

2.7 PER-PAIR INDEPENDENCE

`def independent(x, y) -> Bool` plus a proven `law independent_ok` (`Indep(independent(x, y), s, x, y)`, the swap law when True; `03` §7). A head is a consistent set: downward closed, and every two concurrent messages in it independent. A message joins every head whose extra messages it commutes with, and seeds a new set grown greedily (canonical order) from concurrent independent messages. Heads contained in others are dropped. Gate D8 checks all 24 arrival orders of a 4-message mix. The greedy step is not proven to give every maximal consistent set in general.

3. IO NODES

Only a writer performs IO. It runs its own send on a scratch version. Each `NeedIO` (status 4, the receipt's `note` = op) asks the host, and the answer is appended to the envelope's `ios`. Replicas apply the envelope with the recorded answers and never call the host (`02` §8). The reference host is deterministic (clock for op 0, a keyed random word otherwise). A real node substitutes the OS clock and RNG at the host boundary.

If an IO node dies, its chain ends and nobody waits for it: there is no quorum and no lease (gate D2). A client resumes through another writer, naming the last version it saw. IO with external side effects (e.g. an email) is at-least-once if a writer dies between performing and publishing.

4. CAPABILITIES (`membrane/cap.bend`)

```
Cap  = { token, holder: Who{island, id}, island, inst, rights, linear, epoch, state, parent }
Right = Send 0 | Subscribe 1 | Fork 2 | Update 3 | Grant 4 | Replicate 5 | Registry 6 | Io(op) 100+op
state = live 0 | draining 1 | dead 2 | used 3
```

- Tokens are `H(tok.2 ‖ island secret ‖ counter)`. Terms hold only handles.
- `grant` attenuates (rights ⊆ parent's) and records the parent. `revoke` kills a cap and its descendants.
- `epoch_bump` sets older caps to draining; `refresh` trades them for current ones; `drain_end` kills the rest.
- `isolated(X, Y)` searches the cap graph, counting nodes that hold `Replicate` as inside an island. It returns None, or the caps of a witness path.

5. READING AND SUBSCRIPTIONS

`heads(doc)` returns every head with its state, each read in a rolled-back transaction (reads never change heap layout or later fuel). `observe` returns the heads only to a holder of a `Subscribe` cap for the instance (gate D5). The owner's `view` def (`03` §7) is the projection for subscribers. Gossip/streaming transport is outside the kernel: anything that eventually delivers every envelope to every live node works.

6. HASHING (ALL OFF THE REDUCTION PATH)

Event	Hashed
compile / register	`DefHash`, `BookHash` (`03` §6)
send / reduction / delivery	**never**
audit / checkpoint	`StateHash = H(dag.2 ‖ (w, s, canonical state)*)` over the heads; labels and instance ids canonicalized away

7. MEMORY

Every version is pinned while a writer may still name it. `prune` forgets every non-head version; its heap cells become garbage reclaimed by compaction (`island/checkpoint.bend`: restore(take)). An envelope naming a pruned version waits forever, so prune only at a frontier every writer has acknowledged.

Versions and the seen set are persistent maps (`Map<&2, …>`, O(log n)); each version's history is a shared map plus a shared cons list. Only concurrent messages walk history, at most `WINDOW` (64) items per head. Throughput is not measured yet (B7).

8. FAILURE AND RECOVERY

Event	Behaviour
an IO node dies	its chain ends; every other node continues (D2)
replicas partitioned	each side keeps writing; healing = exchanging envelopes; overlapping writes are heads
a fresh / restarted replica	takes any peer's envelope log, in any order (`Cl.revive`)
divergence (bug, bit flip)	caught when audits compare `StateHash`

9. GATES

Gate	Check	State
D1	same envelopes, any arrival order (duplicates, child before parent) → same heads and StateHash	`tests/e2e_cluster.bend` **PASS**
D2	kill an IO node: nobody blocks; a revived replica catches up from a peer's log	PASS
D3	`law commutes` book: concurrent writers, two orders, one head, no fork	PASS
D4	`BMerge` with the book's `merge` equals the hand-merged oracle (vault: 100+50−30+5)	PASS
D5	observe with Subscribe cap ok, Send-only cap denied	PASS
D6	TCP nodes, one replica per process (`net/`)	not started
D7	sequencer fast path and takeover (`Cl.sequence` ; a writer merges open heads, then sequences on)	PASS (both stores)
D8	per-pair independence: 24 arrival orders → same two heads; merge of the heads	PASS (both stores)
W1	worlds: one superposed state, one label per message	PASS
W2	worlds: `settle` collapses every label, state unchanged, sequencing continues	PASS

D.11 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/10-v2-formats.md

06 — Formats

Status: normative, HVM5 v2. Every byte format and hash domain in one place.

1. .IC TEXT (HUMAN-READABLE, HOMOICONIC)

The text form of templates, quotes and debugging output. Grammar = `01` §1.

```
book  ::= def*
def   ::= "@" name "=" term
name  ::= [A-Za-z_][A-Za-z0-9_.]*
ctor  ::= "#" name "{" (term ("," term)*)? "}"
match ::= "~" term "{" ("#" name ":" term)* "}"
```

`parse ∘ print = id` on normal forms (gate X1).

2. SERIAL (CANONICAL BYTES OF A CLOSED NORMAL FORM)

Pre-order. One tag byte per node. Integers are LEB128.

Tag	Node	Payload
0x00	ERA	—
0x01	VAR	de Bruijn level
0x02	LAM	body
0x03	APP	fn, arg (only inside templates; normal forms have none at head)
0x04	SUP	label, a, b
0x05	DUP	label, value, body (binds 2 levels)
0x06	CTR	type id, ctor id, fields…
0x07	MAT	type id, value, arms…

Tag	Node	Payload
0x08	NUM	u32
0x09	OP2	op, a, b
0x0A	SWI	value, z, s
0x0B	REF	DefHash (32 bytes) in quotes/messages; def index in templates
0x0C	CAP	handle index

Canonicality: labels are renumbered 0,1,… in order of first occurrence, and binder levels are assigned in pre-order. `IOR` and `Stuck` are not serializable.

3. HVM5.CORE (COMPILER → BACKEND)

```
magic "HVMC" · u16 version=2 · compiler string
types : vec{ name, vec{ ctor name, arity } }          sorted by name
defs  : vec{ name, arity, pure:u8, ho-bits, Expr }    sorted by name
exports: { init, transition, migrate?, view?, state, msg, reply }
Expr  : tag byte + fields, same tag style as §2 plus 0x10 Let, 0x11 Io
```

Also exposed to Bend 2 as `type Core.Expr is Data: …` for the homoiconic loop (`04` §1).

4. RECORDS

```
Msg        { from: IslandId, token: 32B, nonce: u64, fuel: u64, heap: u64, replyTo?: token, bytes: Serial }
Env        { w: u32, s: u32, parents: [Vid], body }          Vid = (w, s)   (05 §2)
             body = BSend{msg: Serial, ios: [u32]} | BMerge{base: Vid} | BPick
LogEntry   (legacy, island/log.bend: per-op records; not on the v2 ordering path)
Checkpoint { island, seq, epoch, states: sorted [(instance, StateHash)],
             books: sorted [(instance, BookHash)], prev: CheckpointHash, sigs: [(node, sig)] }
Event      { heads: sorted [(Vid, view bytes)], StateHash }
Receipt    { status (0 Done 1 Stuck 2 OutOfFuel 3 OutOfHeap 4 NeedIO 5 Denied 6 StaleEpoch
              7 NoSuchInstance 8 Conflict 9 TypeChange 10 NoSuchBranch 11 Unavailable),
             fuel, reply, shadow: [reply per branch], note (NeedIO: op; branches dropped: 1) }
```

5. HASH DOMAINS

`H` is SHA-256 in pure Bend 2 (`src/crypto/sha256.bend` , FIPS vectors pass). BLAKE3 is optional and not implemented.

Domain prefix	Hashes
`hvm5.def.2\0`	def template + referenced DefHashes
`hvm5.scc.2\0`	mutually recursive group
`hvm5.book.2\0`	sorted (name, DefHash) + manifest
`hvm5.state.2\0`	`Serial` of an instance's normalized state
`hvm5.dag.2\0`	heads: (w, s, canonical state)* (05 §6)
`hvm5.log.2\0`	legacy log entries
`hvm5.ckpt.2\0`	checkpoint record without sigs
`hvm5.msg.2\0`	message bytes (for signatures and dedup)

No hash is ever accepted as a capability.

D.12 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/11-v2-trust.md

07 — Trust, invariants, threats

Status: normative, HVM5 v2.

1. TRUSTED COMPUTING BASE

Component	Trusted for
BendRT + kernel	rules, fuel and heap accounting, rollback, label discipline
membrane	tokens, attenuation, linearity, epochs, IO rights
version DAG	causal buffer, dedup, heads (no ordering layer to trust: `05` §2)
host crypto	SHA-256 (pure Bend); Ed25519 at the host boundary, not implemented
`law commutes`	checked by the official frontend; the runtime trusts the `~commutes` marker
official Bend 2 frontend + `lower`	type soundness of admitted books

Untrusted: contract code, messages, clients, subscribers, other islands, grafted bytes.

2. INVARIANTS

1. Terms cannot construct tokens. Handles index only the holder's own table.
2. Attenuation only: passed rights ⊆ sender's rights.
3. Linear caps are used at most once.
4. Every send is bounded by fuel and heap budgets.
5. Sends and promotes are atomic (write log; one log entry).
6. Epochs are monotone. Stale-epoch operations are rejected.
7. `Stuck` is safe. It rolls back and never crashes the island.
8. Equal `DefHash` means identical code.
9. The only non-pure inputs are logged IO results.
10. No user-visible superposition in contract books (`01` §7.3).

3. THREATS

Threat	Mitigation	Residual
exponential DUP / fuel bomb	fuel + heap budgets	—
label collision to annihilate foreign SUPs	label discipline, reserved runtime range	proof obligation L2
malicious `Serial`	total decoder; structural type check before graft; `Stuck` → rollback	compiler bugs
code-carrying message abuse	receiver budgets; DefHash fetch from the registry only	—
replay	dedup by `(writer, seq)`	a writer reusing a seq for different content: first arrival wins per replica (needs signed envelopes)
stale epoch after promote	CAS	—
reordering network / client	order is the parents in the envelope; arrival order is irrelevant	—
false `commutes`	frontend proof check	compiler bugs

Threat	Mitigation	Residual
version pinning (memory)	`prune` at an acknowledged frontier	unbounded until pruned
divergent replica	StateHash audits over canonical heads	until the next audit
subscriber leaks	`view` projection limits exposure	copying cannot be prevented
operator reads heap	out of scope	needs TEE
cross-domain hash reuse	domain prefixes (`06` §5)	—

4. PROOFS

`LAWS.bend` states L1–L6 (`01` §8). `PROOF.bend` must be green before any release. It covers the rules, the label discipline, canonical quoting, projection correctness, merge3 and the cap-graph isolation model. It does **not** cover BendRT's generated C, the OS, the network or cryptography. Necessary, not sufficient.

D.13 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/12-v2-plan.md

HVM5 v2 plan

Order is fixed: **kernel → compiler → code as data → membrane → islands → benchmarks.** A milestone ends when its gates are green on real code. Pure simulations are allowed only as `LAWS.bend` models, never as evidence for a gate.

M1 — IC KERNEL (CRITICAL PATH)

Spec: `01` , `02` . Files: `src/kernel/{ptr,heap,book,rules,whnf,norm,quote}.bend` , `tests/kernel/*.bend` , `bench/b1.bend` .

1. `ptr` + `heap` + write log; unit tests.
2. Rules APP/DUP/MAT/SWI/OP2/ERA. Then DUP-LAM and DUP-SUP'.
3. `whnf` canonical strategy with fuel; `normalize` .
4. `quote` / `graft` / `πᵢᴸ` ; `.ic` printer and parser.
5. HVM4 cross-check corpus (Church numerals, duplicated lambdas, Scott lists).

Exit: K1–K8. **Nothing downstream is claimed until K2 (DUP-LAM) passes.**

M2 — COMPILER

Spec: `03` . Files: `src/compiler/{core,lower,hash}.bend` , upstream patch `bend --emit-core` .

1. Pin a Bend 2 version and write the `--emit-core` patch (offered upstream).
2. `core.bend` decoder → `Core.Expr` .
3. `lower.bend` : linear matches, affinity/labels, ids, refs.
4. `hash.bend` : DefHash/SCC/BookHash (BLAKE3 through the host boundary).
5. Corpus C3 against BendRT.

Exit: C1–C7. Move `../version1/tools/emit-inet.ts` to an attic (files are never deleted).

M3 — CODE AS DATA

Spec: `04` . Files: `src/instance/{book,instance,fork,test,merge,promote}.bend` .

Exit: F1–F8, plus the vault and Bob demo from `04` §9 running end to end on one island.

M4 — MEMBRANE

Spec: `05` §2, `07` . Files: `src/membrane/{cap,budget,io}.bend` .

Exit: invariants 1–10 of `07` tested by an adversarial corpus.

M5 — ISLANDS

Spec: `05` . Files: `src/island/{version,cluster,checkpoint}.bend` , `net/` host adapter (TCP gossip + disk, at the host boundary).

No leader and no consensus. Envelopes `(writer, seq, parents)`, a causal buffer, dedup, a version DAG with one DUP per version, superposed heads, resolution by the book's `merge` or a pick, and a commutative path for books with a proven `law commutes`.

1. `version.bend + cluster.bend` (in-process mesh). **Written, checks.**
2. `tests/e2e_cluster.bend`, gates D1–D5. **PASS.**
3. `net/`: one replica per process, gossip over TCP (D6).
4. Sequencer fast path and takeover (D7); per-pair commutation (D8).

Exit: D1–D8.

M6 — BENCHMARKS

Id	Workload	Metric
B1	kernel micro (DUP-LAM, Church 2^{22}, list sort)	interactions/s vs HVM4 C
B2	counter sends, one island	sends/s, p50, p99
B3	vault sends, 3 replicas, K = 1/100/10 000	cost of hashing
B4	fork cost vs state size (1 KB … 1 GB)	flat = O(1)
B5	superposed test vs two runs	sharedRatio
B6	cross-island ping-pong	latency
B7	envelopes/s per replica, 2–3 I0 nodes, commuting vs conflicting	throughput, heads, memory

RISKS

Risk	Effect	Mitigation
BendRT array ops too slow for a heap kernel	low interactions/s	measure in M1 week 1; if > 3× off HVM4, run the same spec on HVM4's C kernel via FFI and keep the Bend 2 kernel as the executable spec
`--emit-core` not accepted upstream	coupling	keep a pinned patch; the core format is ours
superposed testing shares little in practice	weaker headline	B5 decides; fork/merge/promote do not depend on it
λ-label discipline too restrictive	some higher-order code rejected	first-order contracts are unaffected; widen with a proof
24-bit label exhaustion	forced checkpoints	per-checkpoint renumbering
many concurrent heads on non-commuting state	work and memory grow with concurrency	`commutes`, per-pair independence (D8), sequencer fast path (D7), `prune` at frontiers
version lookup is a list	O(versions) per delivery	a map keyed by `(w, s)`; measure first (B7)

DEFINITION OF DONE (FINAL HVM5)

1. A Bend 2 contract compiles via `--emit-core → lower`, without any second frontend.
2. It runs on the Bend 2 IC kernel (BendRT native) with all rules and deterministic fuel.
3. Fork is O(1). Superposed testing produces projections equal to separate runs. merge3 and promote are atomic with draining.
4. Ten processes, three islands × 3 replicas: failover with zero failed sends, catch-up from checkpoint, capability-gated subscriptions, no hashing on the reduction path.
5. `PROOF.bend` green. B1–B6 published with hardware and commit.

D.14 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/13-v2-map.md

Map

CONCEPT → SPEC → MODULE

Concept	Spec	Module
IC terms and rules	01 §1–2	src/kernel/rules.bend
labels, soundness boundary	01 §7	src/kernel/book.bend , src/compiler/lower.bend
fuel, canonical strategy	01 §3, 02 §4	src/kernel/whnf.bend
heap, write log, budgets	02 §2, §5–6	src/kernel/heap.bend
quote, graft, projection	01 §4, 02 §7	src/kernel/quote.bend
--emit-core , core	03 §2, 06 §3	upstream patch, src/compiler/core.bend
lowering	03 §3	src/compiler/lower.bend
DefHash, BookHash	03 §6, 06 §5	src/compiler/hash.bend
instance, send, live branches	00 §2, 04 §5	src/instance/{island,branch}.bend
fork, test, merge3, promote	04	src/instance/{fork,test,merge,promote}.bend
caps, rights, epochs	05 §2	src/membrane/cap.bend
IO boundary	02 §8, 05 §3	writers in src/island/version.bend (author , with_io)
version DAG: envelopes, causal buffer, dedup, heads, merge/pick, commutes, prune	05 §2, §7	src/island/version.bend
world-label store: one superposed state, label per message, settle	05 §2.6	src/island/worlds.bend , src/island/cluster_worlds.bend
per-pair independence	05 §2.7, 03 §7	src/island/{version,worlds}.bend , contracts/kv.bend
leaderless mesh: nodes, kill, revive, observe	05 §3–5, §8	src/island/cluster.bend
checkpoints, compaction	02 §2, 05 §6	src/island/checkpoint.bend
legacy op records (not the ordering path)	06 §4	src/island/log.bend
leader-based draft (attic)	attic/05-islands-leader.md	src/island/attic/cluster_leader.bend
merge / law commutes ABI	03 §7	tools/emit-core.ts , contracts/{flags,vault/merge}.bend
TCP and disk	05 §9 D6	net/ (host boundary, not started)

MODULE DEPENDENCIES

```
kernel/ptr → kernel/heap → kernel/book → kernel/rules → kernel/whnf → kernel/norm → kernel/quote
compiler/core → compiler/lower → compiler/hash            (lower emits kernel templates)
instance/* → kernel/*, compiler/hash
membrane/* → instance/*
island/* → membrane/*, instance/*, kernel/quote
net/ (host) → island/*
```

No module imports TypeScript or JavaScript. The only host code is BendRT itself, the `net/` adapter and crypto, all at the boundary.

V1 → V2

v1 artifact	v2 fate
`src/hvm5/eval.bend` (tree/name evaluator)	replaced by `kernel/`; kept as v1 evidence
postfix `.inet` package	replaced by `Serial + .ic` text
`src/bendc.bend` (shape parser)	retired; `--emit-core + lower`
`tools/emit-inet.ts`	migration aid until C1; then moved to an attic
`src/sim.bend`	replaced by real islands; its scenarios become N-gate tests
`PROOF.bend / LAWS.bend`	rewritten for L1–L6

D.15 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/14-v1-calculus.md

01 — HVM5 v1 calculus

Status: normative for HVM5 v1.
Reference: Lafont, *Interaction Combinators* (1997); Taelin, *Interaction Calculus*; `hvm5/reference/hvm5-mini`.

1. NETS

A net is a tuple

```
N = (A, W, R, S)
```

where:

- `A` is a finite set of agents;
- `W` is a set of wires, each joining two distinct ports;
- `R` is a partial map from root names to ports;
- `S` is the space assignment, a map from agents and roots to space paths.

An agent is

```
a = (id, kind, label, arity, space)
```

with one principal port `a.0` and auxiliary ports `a.1 ... a.arity`.

A port is a pair `(agent, index)`. Each port is incident to at most one wire. A root is a named port that is visible to the membrane.

An **active pair** is a wire joining `a.0` and `b.0`. A wire joining a principal port to an auxiliary port is passive.

2. AGENT KINDS IN V1

Kind	Arity	Meaning
`ERA`	0	Eraser. Releases the reachable graph.
`CON`	2	Binary constructor or pair. `label` names the constructor.
`CTOR`	n	Named constructor with n fields. `payload` is inert data.
`DUP`	2	Lazy duplicator with a label.

Kind	Arity	Meaning
SUP	2	Superposition with a label.

DUP and SUP are dual. CON and CTOR are data. ERA is the only agent that can release a graph without producing a value.

CTOR with arity = 0 is a leaf (a number, a string, a unit). Its payload is not inspected by the reducer.

3. INTERACTION RULES

Rules are local: they inspect only the two agents of an active pair and their ports. No rule searches the net.

3.1 ERASURE

```
ERA - ERA   =>  (empty)

ERA - X     =>  for each auxiliary port X.i:
                  create ERA' in X.space
                  connect ERA'.0 to the far end of X.i
                remove X
```

The first rule removes both agents. The second propagates erasure to every child. Each rewrite costs one fuel unit. The created erasers are reduced later by the same rules.

3.2 ANNIHILATION

```
CON(L) - CON(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
DUP(L) - DUP(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
SUP(L) - SUP(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
DUP(L) - SUP(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
```

Same-label DUP / SUP is the primitive fork/merge elimination: the two alternative branches are paired with the two uses.

Different CON labels are a **typed clash**, not an interaction. The pair is reported as an error and never silently reduces.

3.3 DUPLICATION OF DATA

A DUP facing a constructor duplicates the constructor's fields:

```
DUP(L) principal - CON(K) principal
with DUP aux d1,d2 and CON fields c1..cn
=>
create CON1(K), CON2(K)
connect d1 <-> CON1.principal
connect d2 <-> CON2.principal
for each field i:
  create DUPi(L)
  connect the old field wire to DUPi.principal
  connect DUPi.aux1 <-> CON1.aux_i
  connect DUPi.aux2 <-> CON2.aux_i
remove the original DUP and CON
```

The same schema applies to CTOR with n fields. Each field gets its own duplicator. This is the only rule that grows the net in the contract fragment.

3.4 DIFFERENT-LABEL DUP / SUP

The general interaction calculus commutes different labels. v1 does not use this rule on the contract path: the type checker requires equal labels at a fork/merge boundary. A future version may enable the full commutation rule and must then carry the confluence proof for it.

3.5 TYPED STUCKNESS

Any other active pair is a **stuck typed net**, not a silent erase. Examples:

- CON(A) - CON(B) with A != B;
- CTOR principal connected to a data CON;
- a DUP whose label is outside the typed boundary.

The reducer returns `Stuck{at, reason}`. This is an error value, not a crash.

4. REDUCTION RELATION

Define a one-step relation `N -> N'` by choosing an active pair and applying its rule. A reduction sequence is a finite or infinite chain of steps.

A net is **strongly normalizing under fuel** if every sequence of at most `f` steps either reaches a normal form or stops with `OutOfFuel`.

The reducer is:

```
reduce(net, root, fuel, listenerSpaces) -> Result
Result = Done{net, root, interactions}
       | OutOfFuel{net, interactions}
       | Stuck{net, at, reason, interactions}
       | Denied{space, reason}
```

`listenerSpaces` is the set of spaces the calling node is authorized to reduce. An active pair whose agents do not both belong to an allowed space is not reduced; it remains in the net. This is the partition rule.

5. CONFLUENCE

For the v1 typed fragment:

- erasure rules commute with every rule that does not touch the erased agents;
- annihilation rules are local and deterministic;
- data duplication is a local copy schema with fresh agents and labels;
- typed clashes are terminal for the pair and do not rewrite neighbours.

Claim (local diamond). If two active pairs are disjoint, their rewrites commute. If they overlap, they are the same pair. Therefore the fragment is locally confluent.

Boundary. Different-label `DUP` / `SUP` commutation, fusion, packed-number fast paths, and host effects are outside the v1 diamond claim. They may be added only with a rule-specific argument or a `NonConfluent{reason}` marker that removes the agreement claim from the affected subspace.

6. FUEL

Fuel is a natural number. Every rewrite consumes exactly one unit:

Path	Cost
erasure step	1
annihilation	1
data duplication	1
typed clash detection	1
serialization of a graph	0 (outside reduction)
membrane verification	0 (outside reduction)

The fuel counter is part of the receipt. A send that exhausts fuel commits nothing. The membrane may reserve fuel for the reply root; the reservation is explicit in the send envelope.

7. LABELS

Labels are strings or 64-bit integers. They have two jobs:

1. **Correlation.** Same-label `DUP` / `SUP` annihilate; different labels do not.
2. **Typing.** A constructor label identifies a constructor in a package.

Labels are not capabilities and are not secret. A `DUP` label is not a host reference count. A `SUP` is not a list.

8. ROOTS AND CAPABILITIES

A root is a named port. The membrane hides the raw port and exposes a capability token. A capability has:

```
Cap = (root, space, mode, epoch)
mode = shared | linear
```

- A `shared` capability may be used many times.
- A `linear` capability is consumed by one send and cannot be duplicated.
- An epoch binds the capability to a runtime generation. A capability from an old epoch is dead even if its token is replayed.

The capability record lives only in the membrane. It is never a graph node and never serialized raw.

9. HOMOICONIC ENCODING

A graph is encoded canonically as:

```
Graph :=
  agents: sorted [id, kind, label, payload, arity, space]
  wires:  sorted [[agent, port], [agent, port]]
  roots:  sorted [name, agent, port, space]
```

Agent ids are local to the graph. The canonical encoding sorts by stable keys and rejects duplicate ports. The content hash is

```
hash(G) = SHA-256("hvm5.graph.v1\0" || canonical(G))
```

Domain separation prevents a graph hash from colliding with a deploy envelope or a contract state hash.

10. HOMOICONIC TERM SURFACE

v1 exposes the calculus as a Bend 2 DSL:

```
Term = Var | Lam | App | Dup | Sup | Era | Num | Add | Eq | Tag | Match | Ref
```

`Term` is a Bend 2 `Data` type. Code, state, messages, patches, and programs are `Term` values. The interpreter is also Bend 2 and runs on BendRT. A term elaborates to the graph model in sections 1–8; the graph is the semantic model, not a separate host runtime.

Important: BendRT is a flat C evaluator, not an interaction-net machine. This DSL is an object-level HVM5 interpreter running on BendRT. The current evaluator reduces terms; it does not yet maintain a first-class agent/port/wire store. A true inet-native implementation requires the graph kernel described in sections 1–8 to be implemented in Bend 2 (or emitted to the host) and used by the contract runtime. The term interpreter is the frontend/reference for that kernel, not a replacement for it.

The v1 evaluator supports APP-LAM, APP-SUP, DUP-SUP with the same label, ground-data duplication (`Dup` of a `Tag`, `Num`, or `Era`), and the typed `Num` / `Add` / `Eq` / `Tag` / `Match` extension. DUP-LAM and different-label DUP/SUP are explicit v1 boundaries: a term that needs them returns `Stuck`, never a silent approximation. The graph semantics remains the target for those rules.

Binder names in the source format are globally unique. This is a well-formedness restriction that makes substitution capture-free; a frontend may rename before deploy.

11. REFERENCE INTERPRETER

`hvm5/reference/hvm5-mini/main.bend` is an early-draft port of `hvm5.c`. Its `LAWS.bend` and `PROOF.bend` are useful evidence for parser cursor, printer packing, and interaction counting, but it is **not** the v1 semantics.

v1 is the new Bend 2 DSL in `src/hvm5/`. It does not import the early draft. The draft stays in the tree as history and as a comparison point for the term syntax and interaction count.

D.16 · bend-gme/version-1/.opencode/skills/hvm5-ic/references/15-v1-hvm4-execution.md

01a — HVM4 as the execution reference

Status: informative, with normative consequences for v1.
Source checked: `HigherOrderCO/HVM4`, `src/hvm.c` plus `docs/hvm/core.md`, `docs/hvm/memory.md`, and `docs/hvm/interactions/*.md` (2026-10-20).

1. WHAT HVM4 ACTUALLY IS

HVM4 is a single-file C runtime for the Interaction Calculus. It is **not** an explicit graph store with `Agent` / `Port` / `Wire` records. It represents a term as one 64-bit word:

```
TAG (8 bits) | EXT (24 bits) | VAL (32 bits)
```

and reduces terms through a book and a heap:

- **book**: immutable static terms for `@name` definitions;
- **heap**: mutable dynamic terms;
- **SUB cells**: a binder's body/expr slot is replaced by a substitution cell, and linked variables resolve through it;
- **linked variables**: `VAR`, `DP0`, `DP1` point at their binder cell;
- **quoted variables**: `BJV`, `BJ0`, `BJ1` use de Bruijn levels for collapse;
- **DUP nodes**: `DP0` / `DP1` share the duplicated expression slot;
- **ALO**: a static book term plus a bind list, expanded one layer at a time;
- **UNS**: unscoped binding support.

The interaction vocabulary includes:

```
APP-LAM, APP-SUP, DUP-SUP, DUP-LAM,
DDU-SUP, DSU-SUP, ALO-LAM, ALO-DUP,
MAT/SWI/USE, OP2/EQL/AND/OR,
ERA, SUP, DUP, CTR, NUM, REF, NAM/DRY
```

2. THE IMPORTANT CORRECTION

HVM4 is not a port/wire graph even though its semantics is the Interaction Calculus. It is a **term encoding of the calculus** with sharing, binders, labels, substitution cells, and a static book.

Therefore HVM5 v1 should be judged the same way:

> An HVM5 term machine that compiles to BendRT is a legitimate inet representation if it preserves the IC interactions, labels, sharing, and observable normal forms.

The explicit `Agent` / `Port` / `Wire` model in `01-calculus.md` remains the semantic model. It is not required to be the heap representation.

3. WHAT V1 ADOPTS FROM HVM4

HVM4 idea	v1 use
static book + dynamic heap	`Book` of `Term` definitions plus a `Heap` of dynamic cells
linked variables and SUB cells	binder cells for `Lam` and `Dup`; substitution without a global map
`DP0` / `DP1`	the two sides of a `Dup`
`DUP` / `SUP` labels	correlation and fork/merge semantics
APP-LAM / APP-SUP	application rules
DUP-SUP / DUP-LAM	duplication rules

HVM4 idea	v1 use
ALO	a static package term expanded lazily at graft time
collapse	turn a net with `SUP` / `DUP` into a set of pure terms
`-s` interaction count	fuel and benchmark evidence

4. WHAT V1 DOES NOT ADOPT YET

- The exact 64-bit packing. Bend 2 has `U32` and `Nat`, but the reference machine uses a `Term` ADT so it is readable and checkable. A bit-packed representation is an optimization slice.
- The C runtime, GPU lanes, and `-C` collapse implementation.
- The full dynamic-label (`DDU/DSU`) syntax.
- Full ALO/UNS optimization. v1 can expand a book reference eagerly or one-layer-at-a-time; the observable result must match.
- HVM4's exact parser. v1 starts with a smaller Bend 2 subset and a canonical `Term` builder.

5. PIPELINE V1 SHOULD IMPLEMENT

```
Bend 2 source (subset)
  -> parse / check
  -> core IR
  -> lower to HVM5 `Term` (Lam, App, Dup, Sup, Ctr, Mat, Num, Op)
  -> insert DUP/ERA for non-linear binders
  -> Book of static terms
  -> heap instantiation (ALO-like)
  -> reduce (APP-LAM, APP-SUP, DUP-SUP, DUP-LAM, ...)
  -> quote / collapse
  -> state hash + receipt
```

The current `eval.bend` implements the middle of this pipeline (term reduction) but not the Bend 2 frontend, the static book/heap, or the full rule set.

6. WHY THIS GIVES SELF-MODIFICATION

Because the book is data:

- a definition is a `Term` value, not a compiled native function;
- a patch is a `Term` transformation;
- a fork is a `Dup` of a live term/state and a new book entry;
- a merge is a `Sup` collapse plus a visible-root change;
- a book reference can be redirected without rewriting the whole heap, as long as the old term stays reachable for in-flight uses.

That is the homoiconic property the runtime needs. It does not require the host to expose `Port` / `Wire` records.

7. GATES

1. `bend src/hvm5/main.bend` runs the DSL on BendRT.
2. `bend src/hvm5/main.bend -o main` produces a native binary that prints the same output.
3. A Bend 2 subset compiles to `Term` and reduces to the same value as the reference term.
4. APP-LAM, APP-SUP, DUP-SUP, and DUP-LAM tests pass.
5. A `Book` reference can be replaced atomically while an old instance continues to reduce.
6. The term machine has a fuel count and a canonical term digest.

Gates 1–2 are verified. Gates 3–6 are the next slices.

PART E · Runtime-project skills (.opencode → research-to-release, verbatim)

The methodology skill that the bend-runtime HVM5 work used, including the hvm5-v1 map set (reads as: goal → system → types → spaces → lifecycle → code → risks → slices → gates → acceptance → traceability) plus the environment/keys note and the artifact templates.

Source root: `bend-gme/version-1/.opencode/skills/research-to-release` · **16 files · 81.8 KB**

▼ In this part (file list)

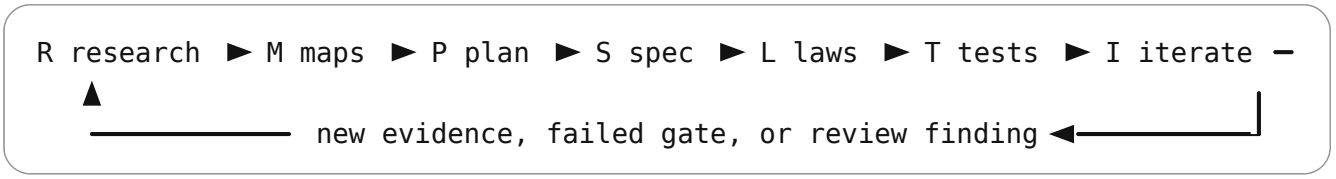
- **E.1** · `bend-gme/version-1/.opencode/skills/research-to-release/SKILL.md`
- **E.2** · `bend-gme/version-1/.opencode/skills/research-to-release/references/01-skill-style.md`
- **E.3** · `bend-gme/version-1/.opencode/skills/research-to-release/references/02-hvm5-v1-maps-readme.md`
- **E.4** · `bend-gme/version-1/.opencode/skills/research-to-release/references/03-hvm5-v1-goal-map.md`
- **E.5** · `bend-gme/version-1/.opencode/skills/research-to-release/references/04-hvm5-v1-system-map.md`
- **E.6** · `bend-gme/version-1/.opencode/skills/research-to-release/references/05-hvm5-v1-types-map.md`
- **E.7** · `bend-gme/version-1/.opencode/skills/research-to-release/references/06-hvm5-v1-spaces-map.md`
- **E.8** · `bend-gme/version-1/.opencode/skills/research-to-release/references/07-env-and-keys.md`
- **E.9** · `bend-gme/version-1/.opencode/skills/research-to-release/references/07-hvm5-v1-lifecycle-map.md`
- **E.10** · `bend-gme/version-1/.opencode/skills/research-to-release/references/08-hvm5-v1-code-map.md`
- **E.11** · `bend-gme/version-1/.opencode/skills/research-to-release/references/09-hvm5-v1-risks-map.md`
- **E.12** · `bend-gme/version-1/.opencode/skills/research-to-release/references/10-hvm5-v1-slices.md`
- **E.13** · `bend-gme/version-1/.opencode/skills/research-to-release/references/11-hvm5-v1-gates.md`
- **E.14** · `bend-gme/version-1/.opencode/skills/research-to-release/references/12-hvm5-v1-acceptance.md`
- **E.15** · `bend-gme/version-1/.opencode/skills/research-to-release/references/13-hvm5-v1-traceability.md`
- **E.16** · `bend-gme/version-1/.opencode/skills/research-to-release/templates/01-artifact-templates.md`

E.1 · `bend-gme/version-1/.opencode/skills/research-to-release/SKILL.md`

Research to release

Evidence first, then a plan you can cut into independent slices, then a spec you can test, then laws you can prove; ship small and review honestly.

MODEL



Each phase has an **entry** condition and an **exit gate**. A phase is done when its gate passes, not when someone says so.

Phase	Output (this repo)	Exit gate
R	`docs/01-research/*.md` , `docs/01-research/evidence/*` (dated, with commands)	every load-bearing fact has a source or a reproducible check, **and a reuse check** (below)
M	maps: goal, system, types, spaces, lifecycle, code, risks (example: `references/02-hvm5-v1-maps-readme.md` … `references/09-hvm5-v1-risks-map.md`)	each goal has an observable; each risk an owner

Phase	Output (this repo)	Exit gate
P	`docs/02-plan/00-roadmap.md` phases, `01-backlog.md` slices with stable IDs, `02-decisions.md` ADRs, `03-risk-register.md`, `05-open-questions.md`	every slice has acceptance criteria and dependencies
S	`docs/03-spec/NN-*.md`: numbered, normative, testable statements	traceability goal → spec § → test (example: `references/13-hvm5-v1-traceability.md`)
L	plain-words law drafts → user approval → `LAWS.bend` → `PROOF.bend`	`bend PROOF.bend` passes for approved laws
T	property, scenario, golden (independent reference), lane parity, budget tests	`bun run check && bun run lint && bun run test` + gates green from a clean tree
I	small slices checked by the inner loop (below), `NOTES.md` entries, REVIEW-style reviews	reviewer finds no blocker; NOTES records deviations and measurements

HONESTY RULES

1. Label every claim **proved** (a law checks), **measured** (machine, version, numbers), or **designed** (not built). Never let designed read as measured.
2. Record negative results and limits in the same place as the wins (`../hvm5-ic/references/03-v2-review.md` is the model).
3. A semantic change gets a new version directory; old versions stay runnable as evidence.
4. Capabilities that need an unreleased runtime (superposition and sharing need HVM5, which does not exist yet) are **designed** at best and never enter a plan as buildable.
5. Scope changes need an ADR; laws need user approval; money or data-safety trade-offs go to the user as an open question.

REUSE BEFORE YOU WRITE (MANDATORY, IN THIS ORDER)

Writing a helper is the last resort, not the first. Before any new module, component, adapter or config path, do this — and it applies to *infrastructure* (config, logging, retries, HTTP, auth, storage) exactly as much as to domain code. "It is only 40 lines" is how duplicates are born.

1. **The sibling reference apps on this machine** — `lambench-pro/reference/ghui` (Effect + Bun) and `hazel/apps/web` (Vite + React). One `cat of src/config.ts or .env.example` beats inventing a pattern. They were written to solve exactly these problems.
2. **Effect v4 itself** — `~/.cache/effect-v4/packages/effect/src/` (the source of truth) and the relevant `effect-*` skill. **Load the skill for *this* slice**, not the one left in context from the last slice. Config → `effect-config`; HTTP → `effect-http-client`; retries → `effect-scheduling`; tests → `effect-testing`.
3. **Bend 2 itself** — `bend base` (List, Array, Map, U32, F32, Maybe, Sigma, IO) and the `bend2-*` skills. If Base has it, wrap it; never re-derive it. **Then the community, in this order:** builtwithbend.com (`https://builtwithbend.com/`) (a searchable directory of 147 Bend 2 builds, each linking its source) → the Bend hub (`https://hub.bend-lang.com`) (`import pkg@version/main.bend`) → lilalittle/bend-packages (`https://github.com/lilalittle/bend-packages`) (the complete A–Z name list, since the hub API caps at 100 names) → Bend Crater (`https://pedrovioliv.github.io/bend-crater/`) (which hub package file compiles on which `bend` release; file-level, and `unsafe` there means a foreign effect, not a failure) → GitHub. Before cloning a shape, read the **import edges**: `bun tools/hub-index/query.mjs <term> --deps` shows what a package composes and who imports it, and the hub's own ranking is hub.bend-lang.com/packages (`https://hub.bend-lang.com/packages`) (dependents/hot — popularity, not correctness). Known prior art worth opening first: **`opencoredev/bendo`** (MIT) for HTTP/1.1, routing, middleware, WebSocket, JSON and proofs over Base TCP, and `Emerging-Patterns/ezhttp` for a packaged HTTP client and server.
4. **The repo** — `grep` for the concept before adding a file. If two packages need the same logic, it moves down a layer; it is not copied.
5. **GitHub and the web** — for a new problem, search for reference implementations *before* designing: `github search "effect config provider"`, `"bend lang" <thing>`, or the ranked list in `docs/01-research/01-github-landscape.md`. Read the implementation, not just the README.

Licence gate (ADR-017). MIT / BSD / Apache may be ported with attribution in `THIRD_PARTY.md`. GPL / AGPL / LGPL / unlicensed are **ideas only** — re-implement from the description, never copy the code.

Record the search where the code lives, in one header line, so a reviewer can check the decision instead of guessing:

```
reused:   <what, from where>        # e.g. "Config.all pattern from ghui src/config.ts; effect-config skill"
own:      <why nothing fit>         # e.g. "no Effect primitive for a 3x3 Cholesky with a ridge"
searched: <queries + sources>       # e.g. "effect-config skill; effect src Config.ts; gh search 'effect config'"
```

A slice whose header has neither `reused:` nor a recorded `searched:` line is incomplete, even if its tests pass.

FAST ITERATION (THE INNER LOOP)

Speed of feedback is the method's engine: many small edits, each checked in seconds. **Two rules decide every run:**

- **Run only what the change can reach.** Never the whole suite in the inner loop. That holds for Bend and for Effect/TypeScript alike.
- **Check directly, run tests compiled.** `bend <file> --check-only` is the fast check (~0.2 s for most files). To *run* a test, `bend <file>` evaluates `main` inside the checker: it is an interpreter, its cost is the test's computation, and nothing is cached. A compiled run (`bend -o` + node) costs a compile that grows with the import graph, then almost nothing, and the runner caches compiled modules by content. Measured 2026-10-31, 66 files: **223 s direct vs 46 s compiled cold, 5 s warm**. `store/tests/codec` takes 13 s direct vs 0.45 s compiled, and `chain/tests/real_gme` 32 s vs 0.3 s. Running directly was faster for only 6 files, by 0.1–2.5 s. So `test-bend.mjs` compiles every test with a pure `main` by default, and uses the direct lane only for an effectful `main() -> IO(…)`.

What do you want to know? Pick the command by the question, not by habit.

The question	Command	Cost	Runs in
Is this file well-formed? (types, ownership, termination, its laws)	`bend <file> --check-only`	~0.2 s, or the import closure (0.17 s leaf, 3.3 s for `bt/vm`)	after every edit
Is the package still well-formed?	`bun run --cwd packages/core check:changed` (or `check` for all 69)	1.8 s changed / 7 s all	after a few edits
Are the laws about it true?	`bun run --cwd packages/core proof:changed` (`proof` = all, incl. `LAWS_TABLE*`)	0.1 s when nothing reaches / 8–20 s per slow law file / ~166 s all	loop / background
Does it **behave** (does `main` return 0)?	`bun run --cwd packages/core test:bend:changed` (or `test:bend`)	~0.3 s per test file, cached / 46 s cold, 5 s warm, all	after an edit with a test; background for the suite
Do the TS/Effect tests still pass?	`bun run --cwd packages/<p> test:related <files>` / `test:changed`; `test:affected` across packages	1–5 s / by package	after an edit
Do the interpreter and the compiled JS agree?	`bun run --cwd packages/core test:bend:direct`	~220 s	background, occasionally
Is a law kernel-valid, not just checker-valid?	`bend <LAWS file> --verdict`	needs Lean v4.34.0 (not installed; R-29)	nightly, once R-29 lands
Is the tree ready to commit?	`fmt:check`, `lint`, `check`, `test`, `proof` (ADR: `proof` is local today, R-10 moves `proof:fast` into CI)	minutes	background before every commit

`--check-only` is a well-formedness gate, not a behaviour gate: a `main` that returns 2 passes it and fails when run. A test's job is to run; a law's job is to be checked. Use both, not one.

The whole loop on one real edit (`# probe` appended to `chain/walls.bend`, 2026-10-31): check **0.23 s**, `check:changed` **1.8 s** (3 modules), `proof:changed` **0.10 s** (no law file imports `walls`), `test:bend:changed` **1.3 s** (3 test files, each recompiled because its import changed). Compiling the edited test itself is 0.2–0.4 s for a `chain` test and 3.4–4.2 s for a `lang` test, whose imports pull in the compiler.

Step (inner loop, foreground)	Command	Selects
1. the module you edit	`BEND_NO_TELEMETRY=1 bend <module> --check-only`	that file (~0.2 s)
2. modules that import it	`bun run --cwd packages/core check:changed`	modules reached through imports (`check.mjs --changed`)
3. laws about it	`bun run --cwd packages/core proof:changed`	`LAWS*.bend` reached by the change, `LAWS_TABLE*` skipped
4. tests about it	`bun run --cwd packages/core test:bend:changed`	`tests/*.bend` reached, compiled and cached (an edit to `chain/walls.bend` : 3 tests in 1.4 s)
5. TS/Effect tests about it	`bun run --cwd packages/<p> test:related <changed files>` (no git) or `test:changed` (git)	vitest files importing the change

Step (background, read before the commit)	Command
lane parity: every non-F32 test on the direct lane (interpreter vs JS backend agree)	`bun run --cwd packages/core test:bend:direct > /tmp/bend-direct.log 2>&1` (~220 s)
TS packages reached by the change	`bun run test:affected` / `check:affected` (turbo, by package)
the full gates	`fmt:check` , `lint` , `check` , `test` , `proof` (incl. `LAWS_TABLE*`). CI runs the first four and the build; `proof` is local until R-10 adds `proof:fast` to CI

Explicit paths work instead of git for every Bend runner: `node test-bend.mjs bend/chain/walls.bend` selects the 3 test files that import it (out of 66). `--list` prints the selection without running anything. Each file prints its time: a test over 10 s or a law file over 5 s is flagged `slow` (`BEND_SLOW_TEST_S` , `BEND_SLOW_LAW_S`). Compiled modules are cached by content (Bend version + the file + everything it imports), so an unchanged test never recompiles (measured 7.3 s → 0.1 s).

1. **Prefer small, fast proofs.** Prove as much as possible, but as many small laws, each checking in well under a second, each about one module, so that `proof:changed` stays fast. State a property over all values with `for` and prove it by cases or induction. An exhaustive enumeration (a whole calendar range, every expiry) is the **last resort**, used only when no inductive proof is practical. It goes in a `LAWS_TABLE_*` file, split by range, and runs in the full gate, never in the loop. A law over one fixed example is a test; call it a checked example, not a proof.
2. **Tests run compiled; keep their import graphs small.** The compiled lane is also the one that ships in the worker, so a test there checks what users run. Its cost is the compile, which follows the import graph (0.3 s for a leaf test, 4 s for a `lang/*` test that pulls in the compiler). So a test imports only the module it tests. `# lane: js` still marks F32 tests: they must be compiled, since `bend <file>` leaves F32 unreduced. Use the direct lane (`bend <file>`) by hand only for a quick look at a tiny pure `main` , or for `IO` mains.
3. **Packages stay separately testable**: each package has its own `check` / `test` ; no test reaches into another package's internals. Each Bend layer has its own `LAWS\*.bend` (a law file checks itself: `bend LAWS.bend`) and `tests/` .
4. **Keep the import graph small, not just the test.** The compile is the cost: a `chain` test whose only import is `chain/walls.bend` recompiles in 0.2–0.4 s, while a `lang/tests/*` file that pulls in `lang/compile` pays 3.4–4.2 s on every edit. A test imports the module it tests and nothing else; a helper used by several tests goes in the layer, not in each test file.
5. **Keep tests small too.** A test file that is flagged `slow` makes every change under it slow. Split it by concern, shrink its fixture, or move its long case to the background lane. The 20-strike real-data max-pain test (~50 s) is the example to avoid in the loop.
6. A bug found late gets a fast test or a law that reproduces it, in the same slice.
7. Full gates run before every commit and in CI, never instead of the inner loop. The inner loop never replaces them either: a selection can only miss what the import graph does not show (data files, generated code), and the gate catches that.

THE BACKLOG IS THE INDEX (EVENT-DRIVEN, SAME COMMIT)

One file, updated on events, never "at the end of the phase":

Event	What changes in `docs/02-plan/01-backlog.md`
a slice is **dispatched**	its row → 🔨 naming what runs; it leaves the ready set
a slice **reports, gates green**	🔨 (or 🔨 with the named remainder) — and this is a **dispatch trigger**: recompute the ready set and launch every slice it now contains **in the background**, up to the cap, without blocking the turn
a slice **reports, a gate red**	stays 🔨 with the failure quoted; dependents named as blocked
a slice is **blocked** on a decision/key	❓ with the question; removed from the ready set
a **dependency or signature changed**	the Mermaid graph in that file is redrawn

Completion is a dispatch event. A finished slice almost always unblocks the next one, so "done" is not the end of a turn: verify, commit, flip the row, recompute the ready set, and **dispatch what it now contains as background work** — never block on it. Leaving a now-available slice idle until the next prompt is a stall, and leaving one blocking is a worse stall; the DAG exists precisely so that this decision needs no human.

That file is also the concurrency authority: it holds the status snapshot, the dependency graph, the ready/waiting sets and the phase gates together, so "what can run in parallel" is never a chart in a chat — and never a second file that drifts from it. `NOTES.md` keeps the *why*.

Diagrams are Mermaid, never ASCII art. GitHub renders ```` ```mermaid ```` blocks in `.md`; ASCII alignment breaks on a different font, width or review tool, which makes a plan unreadable exactly when someone needs it. A graph must **not** encode status either — status goes stale in a picture nobody updates; keep 🔨/❓/✅ in the rows and the snapshot.

KEEP THE LOOP RUNNING (NO "CONTINUE" PROMPTS)

A turn ends when the response ends, so **every turn must leave the next work already running**: the next background subagent, the next gate, or the next slice's edits. Do not end with "shall I continue?" — dispatch and report. Background completions resume the session, so the chain continues on its own.

Stop only for a real blocker: a decision only the user can make (credentials, licence, money), a spec ambiguity that changes behaviour, or three failed attempts at one slice. Name the blocker in one line and state what happens when it is answered.

PARALLEL FAN-OUT (WHAT MAY RUN TOGETHER)

A slice may run in parallel when **both** hold:

1. **Disjoint write paths** — no two agents write the same file, directory or shared artifact.
2. **Frozen reads** — everything it imports is already committed and green. An agent that imports a module another agent is still writing will chase a moving target.

Everything else is sequential, no matter how independent it looks.

Example of a real wave (this repo, 6 agents): `ind` B (`bend/ind/`), `store` manifest+plan (`bend/store/`), `chain` exposure (`bend/chain/exposure`), `lang` AST+checker (`bend/lang/`), `bt` fees/sizing/orders (`bend/bt/`), providers (`workers/src/providers.ts`). Each writes its own directory and reads only frozen modules.

The four limits, in the order they actually bite:

Limit	Why it caps the fan-out	Mitigation
File collision	two agents in one directory rewrite each other's imports/formatting	partition by directory, not by task
Verification bandwidth	every result needs `check.mjs`, its tests, fmt/lint, a commit; six at once is a burst	verify in arrival order, commit one slice at a time
CPU	12 cores, each `bend` run is single-threaded; 5–6 concurrent is comfortable, 10+ contends	`PROOF_CONCURRENCY` for proofs; keep agents' checks short

Limit	Why it caps the fan-out	Mitigation
Context	reading and integrating reports costs tokens	short briefs, short reports, one decision per report

The DAG decides **what** can run; these four decide **how many**. Launch the next wave when its inputs are verified and committed, not on a timer.

After a crash or restart, do not re-dispatch on the assumption that agents died: inspect the tree first. If the slice's files exist and check green, the agent may still be alive or may already have finished — re-dispatching gives two owners of one file, which is the collision this rule exists to prevent. (This happened: `lang` and `bt` were each written twice after a restart. Nothing was lost, but the work was duplicated.)

WHAT TO BACKGROUND (THE DEPENDENCY RULE)

One question decides it: **does the next thing I do need this result?**

- **No → background it now**, before starting the next edit, and read the log at the commit gate.
- **Yes → foreground it**, because waiting is the work.

That is the whole rule; the times below only make the common cases concrete (measured on this repo, bend 2.0.39).

Work	Time	Default	Why
`bend <module> --check-only`	0.2 s	foreground	you need the error to write the next line
`check:changed` / `proof:changed` / `test:bend:changed`	seconds (selected files only)	foreground	they decide the next edit
`test:related <files>` (vitest)	1–5 s	foreground	same
`tsc`, `oxlint`, `oxfmt` per package	1–10 s	foreground	same
`check.mjs` (all 69 modules)	7 s	foreground, once before a commit	not per edit
one test file, compiled (`bend -o` + run)	0.3–7 s cold, 0.1 s cached	foreground (`test:bend:changed`)	the default lane
one test file on the direct lane (`bend <file>`)	0.2–32 s, never cached	by hand, or the background parity run	an interpreter: cost = the test's computation
`bun run build` (core.mjs)	~0.7 s	when the FFI changed	the web and worker tests read it
full `proof` (incl. `LAWS_TABLE*`)	~166 s	**background**	nothing depends on it until commit
whole `bun run test`	~80 s TS + Bend 46 s cold / 5 s warm (compiled, cached)	**background**	same
long enumerations / measurements (1M-row codec, calibre sweeps)	minutes	**background**	same
an independent subagent slice (disjoint files)	minutes	**background**	nothing local depends on it

Rules that follow from it:

1. **Start the background job first**, then do the independent work. Do not finish the foreground work and only then start the slow gate.
2. **Never build on a red gate you have not read.** Backgrounding is not skipping: the log is read before the commit, and a failure invalidates what was built on top of it.

3. **Parallelism follows the dependency graph, not the core count.** Two jobs whose results nobody needs yet run together; two steps where the second consumes the first never do.
4. **Agent fan-out is the same rule at slice level**: brief the independent slices in parallel (disjoint files, no shared artifact), keep the slice whose result decides the next step in the foreground.
5. **The wall clock of a gate is its longest law/file**, so before adding parallel jobs, check that the work is split finely enough for parallelism to matter. Measured: five proof files summing to 177 s took **146 s** in parallel (1.21×), because one law is 166 s of that. Split the long one by range first.

ORCHESTRATION LOOP (AGENT: `ORCHESTRATOR`)

1. **Pick** the next ready slices from the backlog (dependencies met).
2. **Partition** into independent work: disjoint files, no shared mutable artifact, no ordering between them. Anything else runs sequentially.
3. **Brief** each subagent with `templates/01-artifact-templates.md` §Subagent brief: context, exact inputs (paths), outputs (paths), acceptance, constraints, report format. Fresh context means the brief must be self-contained.
4. **Fan out**: independent slices in parallel (background); one slice whose result decides the next step in the foreground.
5. **Verify** every result yourself: re-run the gates, read the diff, check acceptance. A subagent's "done" is a claim, not evidence.
6. **Review** at checkpoints with `reviewer` (read-only) or the critic tool: spec drift, weak tests (does the test fail if the code is weakened?), vacuous proofs, over-claims.
7. **Record**: backlog status, `NOTES.md` (built / measured / deviations), ADR if a decision changed.
8. **Escalate** to the user only for decisions they own (scope, laws, money, data, credentials).

Keep the main context lean: delegate reading of large sources and ask for digests; load a skill in the main session only if the core task needs it. Budget failures (a subagent dies, credits run out) are expected: the orchestrator must be able to finish any slice itself.

ROLES

Agent	Does	Never
`researcher`	sources, evidence files, CORS/terms/price checks	edit code
`bend-dev`	Bend modules, FFI, law drafts, Bend tests	change specs silently
`ui-dev`	Foldkit/Effect UI, worker bridge, UI tests	put domain rules in TypeScript
`prover`	proofs for approved laws; runs the gate	weaken a law to make it pass
`reviewer`	independent audit against spec, rules, evidence	fix what it reviews

VERIFY

```
bun run fmt:check && bun run lint && bun run check && bun run test
BEND_NO_TELEMETRY=1 bend packages/core/bend/PROOF.bend      # once the core exists
agent-browser ...                                          # for a UI slice: the acceptance loop in a real browser
```

For UI slices the gate includes the browser loop (see `foldkit-verified-ui`, "E2E in a real browser"): `agent-browser` 0.37.1 is installed; assert on `data-*` /text and read `console` / `errors`, not screenshots.

REFERENCES

File	Covers	Open when
`references/01-skill-style.md`	how skills here are written	writing or reviewing a skill
`templates/01-artifact-templates.md`	research note, evidence check, map, spec section, ADR, slice, gate, law draft, subagent brief, review, NOTES entry	starting any artifact

File	Covers	Open when
`references/02-hvm5-v1-maps-readme.md` … `references/09-hvm5-v1-risks-map.md`	a complete map set (HVM5 v1)	doing phase M
`references/10-hvm5-v1-slices.md`, `references/11-hvm5-v1-gates.md`	slicing and gates	doing phase P
`references/12-hvm5-v1-acceptance.md`, `references/13-hvm5-v1-traceability.md`	acceptance and traceability	doing phase S/T
`../hvm5-ic/references/03-v2-review.md`	an honest review	doing phase I
`../../../docs/02-plan/`, `../../../docs/03-spec/`, `../../../docs/01-research/`	bend-gme: the live example	working on this repo

RELATED

`bend2` (Bend work) · `bend2-proofs` (phase L) · `foldkit-verified-ui` (UI checks) · `alchemy-github-pages` (release).

E.2 · `bend-gme/version-1/.opencode/skills/research-to-release/references/01-skill-style.md`

How skills in this repo are written

A skill is a compressed expert. It states the model, the rules that follow from it, and where the evidence is. It is short because the references are complete.

LAYOUT

```
.opencode/skills/<id>/
  SKILL.md            # ≤ 160 lines; the only file loaded automatically
  references/         # full source documents: 01-<slug>.md, 02-<slug>.md, … in reading order
  templates/          # optional: 01-<slug>.md, … files an agent copies and fills in
```

- `<id>`: lowercase kebab-case, equal to the directory name and to `name:`.
- No `README.md` or other loose `*.md` directly under `.opencode/skills/` (a root-level Markdown file becomes a skill of its own).
- Reference files are named `NN-<slug>.md`, numbered from `01` in the order a reader should open them; flat (no subfolders). The first line is a provenance comment: `<!-- Source: <original path> (copied verbatim; links rewritten to this skill set) -->`. The body is the source text; only links are rewritten so they resolve.
- Each doc lives in exactly one skill; other skills link to it by relative path (`../bend2-proofs/references/01-laws-and-proofs.md`). Links to documents that were not copied become plain text naming the source path, never dead links.

FRONTMATTER

```
---
name: bend2-proofs
description: <what it covers, in one sentence>. Use when <concrete triggers: tasks, file names, error messages>.
---
```

The description is what the model sees when choosing skills. Put the triggers in it: file names (`LAWS.bend`, `PROOF.bend`), commands, error text.

BODY (IN THIS ORDER)

1. `# Title` and one line: the thesis.
2. `## Model`: the idea everything else follows from, in ≤ 12 lines. A formula, a rule table, or a 5-line diagram beats a paragraph.
3. `## Rules`: numbered, imperative, checkable. Each rule is something a reviewer can say "violated" about.
4. `## Patterns`: the smallest complete code that shows each rule. Every ```` ```bend ```` block is checked with the local compiler before it is written down (`BEND_NO_TELEMETRY=1 bend file.bend`), and the version is stated once in the skill.
5. `## Pitfalls`: a table `symptom / message → cause → fix`. Use the checker's real messages.

6. `## Verify` : the exact commands that prove the work (check, run, proof gate, tests, benchmarks).
7. `## References` : a table `file → covers → open when` . Every claim in the body can be traced to one row.
8. `## Related` : sibling skills and when to switch to them.

VOICE

- Say what is true, then what follows. No hype, no ”powerful”, no ”simply”. Short sentences. Numbers with units.
- Separate the three kinds of claim: **proved** (a law checks), **measured** (a benchmark with machine and version), **designed** (not built yet). Never let a designed claim read like a measured one.
- State limits plainly (”Bend 2 does not run on interaction nets; BendRT is a flat C evaluator”). An honest limit is worth more than a feature.
- Prefer one canonical way. If there are two, say which and why.
- Version-stamp anything that can drift (`checked with bend 2.0.25`).

QUALITY BAR (REVIEWER CHECKLIST)

- <input disabled="" type="checkbox"> ≤ 160 lines; nothing in the body that is not used.
- <input disabled="" type="checkbox"> Every code block was run, or is fenced as `text` .
- <input disabled="" type="checkbox"> Every rule traces to a reference row or to a check the agent can run.
- <input disabled="" type="checkbox"> References are numbered `01…` in reading order, flat, with a provenance line; no dead links.
- <input disabled="" type="checkbox"> Triggers in `description` match how an agent would actually ask.
- <input disabled="" type="checkbox"> No claim stronger than its evidence.

E.3 · bend-gme/version-1/.opencode/skills/research-to-release/references/02-hvm5-v1-maps-readme.md

Maps

The maps are the navigation layer. The spec is normative; the maps are the fastest way to keep the whole system in view.

Map	Use
`00-goal.md`	the goal tree, observables, and failure conditions
`10-system.md`	layers, components, dependency direction
`20-types.md`	types, values, capabilities, forks
`30-spaces.md`	space tree, 10 nodes, quorum, bridges
`40-lifecycle.md`	deploy, send, fork, test, merge sequences
`50-code.md`	file layout, imports, tests
`60-risks.md`	risk heat map and stop rules

Read `00` and `10` before changing code. Read `50` before adding a file. Read `60` before claiming a security property.

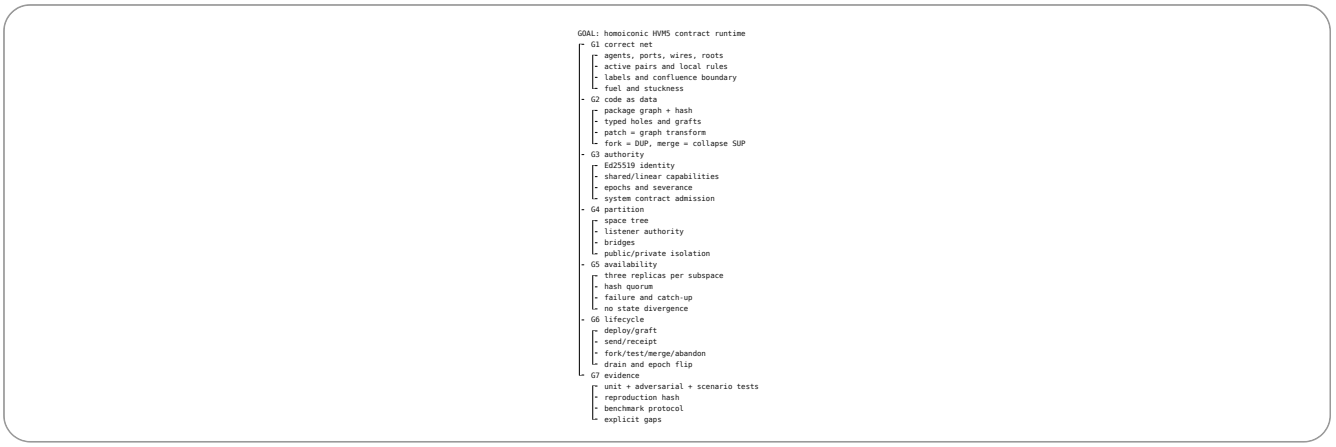
E.4 · bend-gme/version-1/.opencode/skills/research-to-release/references/03-hvm5-v1-goal-map.md

Goal map

THE ONE-SENTENCE GOAL

A signed Bend 2 program can be deployed into one live, partitioned interaction net, invoked by capability, forked into a shadow branch, tested, and merged without stopping the running branch.

GOAL TREE



OBSERVABLES

1. `bend hvm5/version1/src/sim.bend` prints `PASS`.
2. Ten nodes start; three subspaces have three replicas.
3. One public node dies; the other two commit.
4. The dead node restarts and reaches the same hash.
5. A signed deploy creates an instance and two capabilities.
6. A counter send changes state and returns a receipt.
7. A public instance cannot reach a private one; the witness is null.
8. A fork is tested while the live branch serves.
9. Merge flips the root atomically; abandon leaves it untouched.
10. The proof gate is green.

NON-GOALS

- Blockchain consensus, money, finality.
- Native tenant code.
- Source confidentiality from the operator.
- Full Bend 2 compilation in v1.
- Native HVM5 speed in the Bend 2 reference interpreter.
- Browser runtime.

FAILURE OF THE GOAL

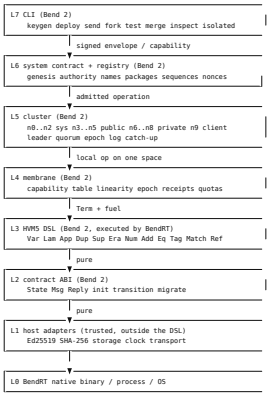
The goal is not met if any of these is true:

- isolation is asserted but no adversarial test exists;
- a node reduces outside its listen set;
- a fork blocks the live branch;
- a merge can leave a mixed state;
- a linear capability can be duplicated;
- a benchmark has no output hash;
- the ten-node run does not include a failure and a catch-up.

E.5 · bend-gme/version-1/.opencode/skills/research-to-release/references/04-hvm5-v1-system-map.md

System map

HIGH LEVEL



LOW LEVEL

```
src/
  README.md          actual module layout and import rules
  bendc.bend         source parser, package model, deploy compile, digest
  deploy.bend        deploy entrypoint; automatic compile
  sim.bend           consolidated system/space/instance/fork/cap model
  hvm5/
    term.bend        Term, Def, Program
    parse.bend       postfix inet package parser
    eval.bend        APP-LAM/SUP, DUP-SUP, Add, Tag/Match, fuel
    show.bend        canonical display
    hash.bend        structural digest
    main.bend        small reducer entrypoint
    run.bend         package runner
```

`model/` and `runtime/` do not exist. The model is currently consolidated in `src/sim.bend`; the deploy/package model is in `src/bendc.bend`. A future split must be a real extraction with updated imports and tests, not empty directories.

DEPENDENCY DIRECTION

```
official Bend 2 frontend -> tools/emit-inet.ts -> inet package
src/bendc.bend (bootstrap) --------------------> inet package
inet package -> parse -> eval
term -> eval
term -> show -> hash

sim -> term, parse, eval, show, hash, bendc
deploy -> sim, bendc
tests -> sim, bendc, hvm5
bench -> sim, term
PROOF -> sim, LAWS
```

The high-level rows (membrane, system contract, cluster, fork) are conceptual layers. In the current reference slice they are implemented as data and functions inside `src/sim.bend`; they are not separate modules yet.

No runtime module imports a host language or a TypeScript file. The only host effects are at executable boundaries (`deploy.bend`, `run.bend`, `main.bend`, tests, bench) and, in a native release, in the explicitly declared adapters.

OWNERSHIP

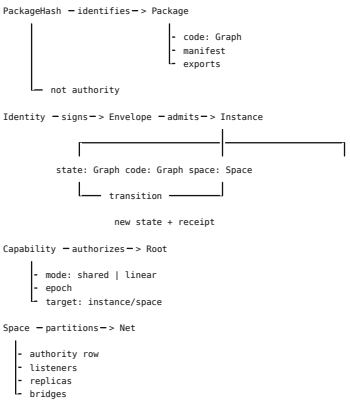
Component	Owns	Must not own
hvm5	term semantics, fuel	keys, tokens
membrane	capability table, epochs	term semantics
system	authority, names, sequences	user state

Component	Owns	Must not own
cluster	log, leader, quorum	term semantics
fork	shadow, Sup, drain	authority rows
CLI	parsing, signing	hidden mutation

E.6 · bend-gme/version-1/.opencode/skills/research-to-release/references/05-hvm5-v1-types-map.md

Type and data map

CORE TYPES



VALUE TREE

```
Value =
  | Ctor{name, fields: Value[]}
  | Nat{value: bigint}
  | Str{value: string}
  | Bool{value: boolean}
```

The graph encoding of a `Value` is a `CTOR` tree:

```
Ctor("Counter", [Nat(7)])
  -> CTOR("Counter", arity=1, space=S)
       aux0 -> CTOR("Nat", arity=0, payload=7)
```

CONTRACT PACKAGE

```
ContractPackage {
  format
  name
  version
  space
  stateType
  msgType
  init: Expr
  transition: Expr
  sourceHash
  code: Graph
  effects: []
}
```

`init` and `transition` are pure ASTs. The package validator checks totality and result type. The graph is the serializable code object.

CAPABILITY

```
CapRecord {
  id
  instance
  space
  root
  mode
  epoch
  rights
  live
}
```

The token is not in the record. The membrane maps `token -> CapRecord`.

FORK

```
Fork {
  id
  instance
  label
  oldRoot
  newRoot
  oldCode
  newCode
  phase: Testing | Merging | Draining | Abandoned
  inputs
  report
}
```

INVARIANTS AS TYPES

- `PackageHash` is opaque and not assignable to `Capability`.
- `LinearCap` is consumed by `send`; the result no longer contains it.
- `Space` is an index, not a string capability.
- `Receipt` is the only result of a mutating operation.
- `Fork` is not an `Instance` until merge.

E.7 · bend-gme/version-1/.opencode/skills/research-to-release/references/06-hvm5-v1-spaces-map.md

Space and authority map

TREE

```
root
├─ root/sys n0 n1 n2 system contract, quorum 2/3
├─ root/public n3 n4 n5 public contracts, quorum 2/3
│  ├─ root/public/acme
│  └─ root/public/demo
└─ root/private n6 n7 n8 private contracts, quorum 2/3
   └─ root/private/vault

n9 = client / observer, listen = ∅
```

AUTHORITY ROWS

Space	Listeners	Deploy	Bridges
root/sys	n0,n1,n2	system	none
root/public	n3,n4,n5	owner/system	none by default

Space	Listeners	Deploy	Bridges
root/public/acme	n3,n4,n5	acme owner	none
root/private	n6,n7,n8	private owner	none
root/private/vault	n6,n7,n8	vault owner	none

PARTITION RULE

```
listen(node, space) = space in node.listen or space is a descendant of one
reduce(node, op) =
  if not listen(node, op.space) then Denied{Space}
  else kernel.reduce(inducedSubnet(net, op.space), op.fuel)
```

A node can hold a capability for a space it does not listen to. It can route a send to a listener. Holding is not reducing.

REPLICATION

```
            n3 (leader)          n4 (follower)        n5 (follower)
send  -----> apply op             apply op             apply op
             state hash H         state hash H         state hash H
             commit if 2 hashes match
```

If `n3` dies, `n4` becomes leader and increments the public epoch. `n5` remains. If `n4` also dies, public is `ReadOnly`.

BRIDGE

```
root/public/acme  ---- bridge(B, shared) ----  root/private/vault
      ^                                              ^
   approval 1                                     approval 2
```

A bridge is an explicit capability edge. The isolation query reports it as a witness. Without both approvals, graft is rejected.

PRIVATE/PUBLIC PROOF

```
publicCapabilities = edges from public instances
privateCapabilities = edges into private instances
path(public, private) = ∅
therefore isolated(public, private) = true
```

The implementation prints the witness, not just the boolean.

FAILURE MAP

```
all alive       -> commit
n4 down         -> n3 + n5 commit
n3 down         -> n4 + n5 commit, epoch++
n3,n4 down      -> n5 alone, ReadOnly
n4 returns      -> catch up, follower again
```

E.8 · bend-gme/version-1/.opencode/skills/research-to-release/references/07-env-and-keys.md

Configuration and keys (Effect, Vite, Bun)

The pattern this repo uses, and the one it got wrong once. Read this before adding any `process.env` read, `.env` parser or key handling.

THE RULE IN ONE LINE

Bun loads `.env`; Effect `Config` reads it; Vite exposes only `VITE_*` (publishable); secrets live in Settings, encrypted. There is no dotenv dependency, no hand-written parser, and no `runSync` at module scope.

NODE SIDE (TOOLING, TESTS) — THE `GHUI` PATTERN

```
// packages/workers/src/config.ts
import { Config, Effect } from 'effect';

const secret = (key: string) =>
         Config.String(key).pipe(Config.withDefault(''), Config.map((v) => v.trim()));

export const providerKeys = Config.all({
         marketdata: secret('MARKETDATA_TOKEN'),
         alphaVantage: secret('ALPHAVANTAGE_KEY')
});

export const loadProviderKeys: Effect.Effect<ProviderKeys, Config.ConfigError> = providerKeys;

// A script needs no Effect runtime; Bun has already put `.env` in process.env.
export const keysFromEnv = (env: Readonly<Record<string, string | undefined>>): ProviderKeys => ({ ... });
```

- `Config.String(name)` — not `Config.string` (v3 name; it does not exist in 4.x).
- Export the **Config value**; run it at the caller's boundary. A library package must not `Effect.runSync` at import time: that turns a module into a side effect and breaks tests and workers.
- Tooling runs under **Bun** (`bun tools/x.mjs`), which loads `.env` itself. `node tools/x.mjs` does not.
- Never parse `.env` by hand, and never copy a loader from another project "because it is small".

BROWSER SIDE — THE `HAZEL` PATTERN, WITH ONE CORRECTION

```
const backendUrl = import.meta.env.VITE_BACKEND_URL; // publishable, baked into the bundle
```

- `VITE_*` values are **public**: they are inlined by Vite. That is fine for a service URL or a publishable key.
- Provider API keys are **secret**; they must never be `VITE_*` , never in `.env` that a build consumes, and never in the repo. In this app they are entered in Settings and stored locally, encrypted (spec/20 §1).
- `.env.example` documents every variable with a comment saying where to get it, and marks optional ones commented out.

WHERE THE ANSWER WAS, BEFORE I WROTE MY OWN

- `lambench-pro/reference/ghui/src/config.ts` — `Config.all + Config.withDefault` , no loader.
- `hazel/apps/web/.env.example + import.meta.env.VITE_*` — the publishable/secret split.
- The `effect-config` skill and `~/.cache/effect-v4/packages/effect/src/Config.ts` .

CHECKLIST (CONFIG OR KEYS TASK)

1. Grep the sibling reference apps for `Config.` , `import.meta.env` , `process.env` .
2. Load the `effect-config` skill; check the real `Config` exports (`Config.String` , `Config.Redacted` , `Config.all`).
3. Decide the value's audience: **secret** (Settings + encrypted), **publishable** (`VITE_*`), or **Node-only** (`.env` via Bun).
4. Write the name in **one** module; no key name appears twice.
5. Verify: a run that prints only **which** providers are configured, never a key; `git check-ignore .env` succeeds; `grep -r <first 6 chars of the key> .` finds nothing.

E.9 · bend-gme/version-1/.opencode/skills/research-to-release/references/07-hvm5-v1-lifecycle-map.md

Lifecycle map

DEPLOY

```
client                membrane              system              subspace
  |                      |                     |                    |
  |-- signed deploy ---->|                     |                    |
  |                      |-- verify key ------>|                    |
  |                      |                     |-- authority ------>|
  |                      |                     |-- sequence ------->|
  |                      |                     |-- admit ---------->|
  |                      |                     |                    |-- graft
  |                      |                     |                    |-- init
  |                      |<--------------------|--------------------|
  |<-- instance + caps --|                     |                    |
```

SEND

```
client --cap + msg--> membrane --resolve--> instance
                        |                     |
                        |                     +-- connect msg root
                        |                     +-- reduce fuel
                        |                     +-- transition
                        |                     +-- new state hash
                        |<--------------------+
                        |-- replicate quorum -> n_leader,n_follower
                        |-- commit receipt
<-- receipt -------------|
```

FORK/TEST/MERGE

```
live root ----+
              |
              +-- DUP(L) --> shadow root
                                |
                            test inputs
                                |
                     +----------+----------+
                     |                     |
                  abort                  merge
                     |                     |
              erase shadow          collapse SUP(L)
                     |                     |
              live root              new root + epoch
                                           |
                                       drain old
```

CODE-AS-DATA

```
old package graph
      |
   patch graph
      |
      v
applyPatch -> new package graph -> hash -> package blob
                                     |
                                  graft copy
                                     |
                                  new instance
```

ATOMIC COMMIT

```
prepare -> validate -> compare-and-swap root+epoch -> append receipt -> drain
              |                 |                          |
           failure           failure                    failure
              |                 |                          |
           old root          old root                 old root or
                                                     idempotent replay
```

STATES

```
Registered -> Deployed -> Live
                    ^         |
                    |         +-- fork --> Testing --merge--> Draining --> Live
                    |                        |
                    +------------------------+--abandon--> Live
```

RECEIPT FIELDS

```
kind, id, instance, space, oldHash, newHash, fuel, epoch, sequence, status
```

No mutation is complete without a receipt.

E.10 · bend-gme/version-1/.opencode/skills/research-to-release/references/08-hvm5-v1-code-map.md

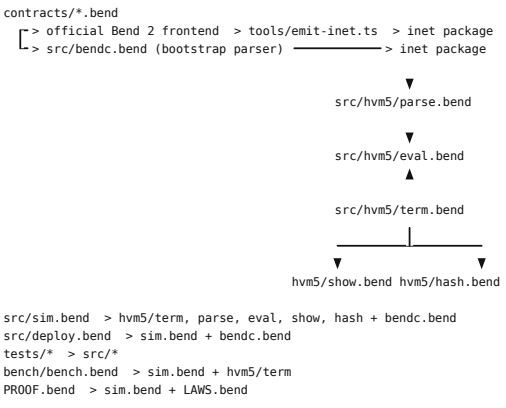
Code map

TREE

```
hvm5/version1/
├── README.md
├── LAWS.bend laws used by PROOF.bend
├── PROOF.bend topology/isolation/bridge proof gate
├── spec/ normative documents
├── plan/ phases, slices, gates, risks
├── map/ this map set
├── compiler/ bootstrap compiler tooling
│   ├── README.md
│   └── bootstrap/
│       └── compile.ts bootstrap source -> inet package
├── tools/
│   └── emit-inet.ts official checked-AST -> inet backend
├── contracts/
│   ├── counter.bend checked Bend 2 contract source
│   ├── tally.bend renamed shape, proves name extraction
│   └── README.md
├── src/
│   ├── README.md actual module layout and import rules
│   ├── bendc.bend Bend 2 source parser, package model, deploy compile
│   ├── deploy.bend deploy entrypoint; auto-compiles source
│   ├── sim.bend ten-node system/space/instance/fork model
│   └── hvm5/
│       ├── term.bend Term, Def, Program (code == data)
│       ├── parse.bend postfix inet package parser
│       ├── eval.bend CEK-style inet reducer
│       ├── show.bend canonical term display
│       ├── hash.bend structural digest
│       ├── main.bend small reducer entrypoint
│       └── run.bend package runner
├── examples/
│   └── fork_merge.bend runnable fork -> test -> merge example
├── tests/
│   ├── eval.bend reducer cases
│   ├── pipeline.bend package -> reducer round trip
│   ├── deploy_compile.bend automatic deploy compile
│   ├── compiler_reject.bend rejection path
│   ├── compiler_names.bend renamed contract shape
│   └── run.bend full ten-node model gate
├── bench/
│   └── bench.bend W3-style repeated-send benchmark
└── legacy/ old v0 work; not normative, not imported
```

There is no `src/model/` or `src/runtime/` directory. The v1 reference slice consolidates those concerns into `src/sim.bend` and `src/bendc.bend`; see `src/README.md`. Do not create placeholder directories for a future split.

DEPENDENCY GRAPH



IMPORT RULES

1. `src/hvm5/*.bend` imports Base and other `hvm5` modules.
2. `src/bendc.bend` imports `hvm5` only.
3. `src/sim.bend` imports `hvm5` and `bendc.bend`.

4. `src/deploy.bend` imports `sim.bend` and `bendc.bend`.
5. Tests may import `src/*`; they must not duplicate the model.
6. `src/README.md` is the source of truth for the actual layout.
7. No circular imports; Bend definitions must appear before use.
8. The runtime does not import TypeScript, Node, or a JavaScript host.
9. `tools/emit-inet.ts` is build-time backend tooling, not a runtime.

NAMING

- Files: lowercase, underscore-free, one concern.
- Types: PascalCase.
- Functions: lowercase, descriptive.
- Term constructors: `Var`, `Lam`, `App`, `Dup`, `Sup`, `Era`, `Num`, `Add`, `Eq`, `Stuck`, `Tag0/1/2`, `Match0/1/2`, `Ref`.
- Space paths: `root/sys`, `root/public`, `root/private`.
- Node ids: `n0` .. `n9` as `Nat` 0..9.

TEST MAP

File	Covers
`tests/eval.bend`	APP-LAM, APP-SUP, DUP-SUP, `Add`, `Eq`, tags/matches, fuel, stuck
`tests/pipeline.bend`	postfix package parse and one-send reduction
`tests/deploy_compile.bend`	automatic deploy compile and reduction
`tests/compiler_reject.bend`	unsupported source is rejected, not miscompiled
`tests/compiler_names.bend`	renamed constructors/fields/parameters
`tests/run.bend`	ten nodes, replicas, quorum, kill/heal, catch-up, isolation, fork/test/merge, digest

External gates:

```
bun tools/emit-inet.ts contracts/counter.bend build/counter-official.inet
bend src/hvm5/run.bend build/counter-official.inet

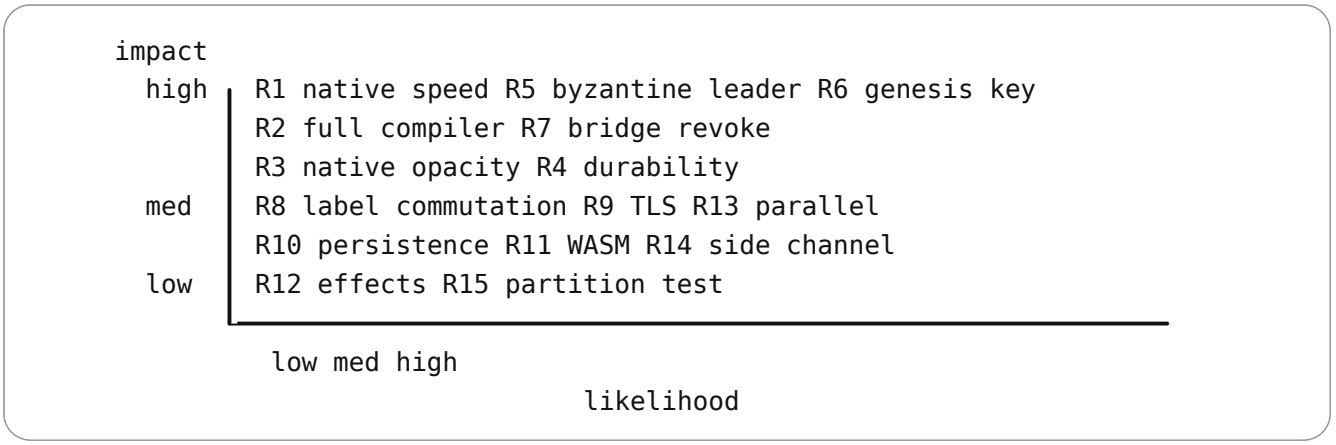
bun tools/emit-inet.ts contracts/tally.bend build/tally-official.inet
bend src/hvm5/run.bend build/tally-official.inet
```

Expected: `done Counter{2} fuel=0` and `done Tally{7} fuel=0`.

E.11 · bend-gme/version-1/.opencode/skills/research-to-release/references/09-hvm5-v1-risks-map.md

Risk map

HEAT MAP



STOP RULES

Stop the release if:

- a node can reduce outside its listen set;
- a linear capability survives two sends;
- a public instance has a path to a private instance;
- two replicas commit different state hashes;
- a merge leaves the old root reachable as live after the epoch flip;
- a benchmark hash does not match the expected output.

MITIGATION SEQUENCE

1. Partition and linearity (R3/R7) before cluster work.
2. Hash quorum (R5) before failover claims.
3. Durable snapshot (R4) before crash claims.
4. Native handle (R3) before security release.
5. Full compiler (R2) before calling it a Bend runtime.
6. Key rotation (R6) before production governance.

ACCEPTANCE EVIDENCE

Each mitigation needs:

- a test name;
- a spec section;
- a command;
- an output hash or receipt;
- a note in `docs/02-plan/03-risks.md`.

Prose is not evidence.

E.12 · bend-gme/version-1/.opencode/skills/research-to-release/references/10-hvm5-v1-slices.md

Slices

Each slice has one seam, one gate, and one rollback. A slice is not a calendar task; it is a reviewable unit of behavior.

ID	Slice	Depends on	Gate
S00	Freeze reference and v0 evidence	—	reference proof green
S01	Canonical graph + hash	S00	same graph, same hash
S02	Kernel erasure/annihilation	S01	A1
S03	Data duplication + fuel	S02	A1, A2
S04	Contract value + expression evaluator	S01	A5
S05	Contract package + validation	S04	total transition
S06	Ed25519 envelopes	S01	A4
S07	Capability membrane + linearity	S06	A4, X1–X5
S08	Space tree + authority	S01	A3
S09	Partitioned reducer	S03, S08	X6
S10	Registry + genesis system contract	S05, S07, S08	A6
S11	Deploy/graft/instance	S10	deploy receipt
S12	Send pipeline	S11	send receipt
S13	Node log + quorum	S12	A7
S14	Ten-node topology	S13	ten nodes start
S15	Failure + catch-up	S14	X9, failover
S16	Fork + SUP record	S12	A9
S17	Test replay + report	S16	A9
S18	Merge/abandon/drain	S17	A9, X10, X11
S19	Code-as-data patch	S05, S16	new package hash
S20	Isolation query + witness	S07, S08	A8
S21	CLI	S11–S20	all commands
S22	Benchmarks	S21	JSON evidence
S23	Full scenario	S14, S18, S20	A11
S24	Bend model + proof	S05	A12
S25	Hardening / gap review	all	09 checklist

STATUS LEGEND

- `TODO` — not started.
- `PARTIAL` — model or test exists, but a named gate is missing.
- `DONE` — gate passes and evidence is recorded.
- `DEFERRED` — explicitly outside v1.

NOTES

S03 and S09 are the two places where a wrong shortcut would invalidate the security story. Do not fake commutation, and do not let a node reduce a pair outside its listen set.

S18 is the only operation with a compare-and-swap on the visible root. It must be tested with a crash before and after the swap.

E.13 · bend-gme/version-1/.opencode/skills/research-to-release/references/11-hvm5-v1-gates.md

Gates

Run these from the repository root. A gate is green only with the expected output. No gate is waived because another gate is green.

G0 — EARLY-DRAFT REFERENCE

```
bend hvm5/reference/hvm5-mini/proof.bend
```

Expected: `All terms check.` This is a historical reference, not v1.

G1 — INET REDUCER

```
bend hvm5/version1/tests/eval.bend
```

Expected: `EVAL True` .

G2 — CHECKER

```
bend hvm5/version1/src/bendc.bend --check-only
bend hvm5/version1/src/sim.bend --check-only
bend hvm5/version1/tests/run.bend --check-only
```

Expected: `All terms check.`

G3 — V1 LAWS

```
bend hvm5/version1/PROOF.bend
```

Expected: `All terms check.`

G4 — CONTRACT → INET → REDUCER

```
bend hvm5/version1/contracts/counter.bend --check-only
bend hvm5/version1/src/deploy.bend hvm5/version1/contracts/counter.bend
bend hvm5/version1/tests/pipeline.bend
bend hvm5/version1/tests/deploy_compile.bend
```

Expected:

```
All terms check.
DEPLOY Counter{0} PKG <u32> DIGEST <u32>
PIPELINE done Counter{2} fuel=0
DEPLOY-COMPILE done Counter{2} fuel=0
```

The official-frontend backend can emit the same package:

```
bun hvm5/version1/tools/emit-inet.ts \
  hvm5/version1/contracts/counter.bend build/counter-official.inet
bend hvm5/version1/src/hvm5/run.bend build/counter-official.inet
```

Expected: `done Counter{2} fuel=0` .

The renamed shape proves name extraction:

```
bun hvm5/version1/tools/emit-inet.ts \
  hvm5/version1/contracts/tally.bend build/tally-official.inet
bend hvm5/version1/src/hvm5/run.bend build/tally-official.inet
```

Expected: `done Tally{7} fuel=0` (init 5 + Inc 2).

The bootstrap frontend can also emit a package file:

```
bun hvm5/version1/compiler/bootstrap/compile.ts \
  hvm5/version1/contracts/counter.bend build/counter.inet
bend hvm5/version1/src/hvm5/run.bend build/counter.inet
```

Expected: `done Counter{2} fuel=0` .

G5 — TEN-NODE SIMULATION

```
bend hvm5/version1/src/sim.bend
bend hvm5/version1/tests/run.bend
```

Expected: the final line is `PASS` , and the report has:

```
NODES 10
PUB Counter{4}
PRIV Counter{3}
N4 Counter{3}
ISOLATED True
PUB-OK True
PRIV-OK True
CATCHUP-OK True
REPLICAS-OK True
QUORUM-OK True
BRIDGE-OK True
CAP-OK True
DIGEST <u32>
PASS
```

G6 — DETERMINISTIC REPLAY

Run the simulation twice:

```
bend hvm5/version1/src/sim.bend
bend hvm5/version1/src/sim.bend
```

The output, including `DIGEST` , must be identical.

G7 — BENCHMARK

```
bend hvm5/version1/bench/bench.bend
```

Expected:

```
BENCH sends=100 ms=<positive>
STATE Counter{100}
```

It is reference evidence, not a native performance claim.

G8 — NEGATIVE/ADVERSARIAL CASES

```
bend hvm5/version1/tests/compiler_reject.bend
bend hvm5/version1/tests/compiler_names.bend
```

Expected: `REJECT True` and `RENAMED True`.

The simulation contains these checks:

- two public replicas down → `QUORUM-OK` (send denied, state unchanged);
- no public/private capability → `ISOLATED True`;
- bridge added → `BRIDGE-OK True` (isolation is false);
- killed node healed → `CATCHUP-OK`;
- fork candidate tested → `PUB Counter{4}` after merge.

The proof gate covers the closed topology and bridge laws.

G9 — NATIVE BENDRT

```
bend hvm5/version1/src/hvm5/run.bend -o build/run
./build/run build/counter.inet
bend hvm5/version1/src/sim.bend -o build/sim
./build/sim
```

Expected: same outputs as the interpreted runs.

G10 — DOCUMENTATION INDEX

`AGENTS.md` lists the docs index, the spec index, the plan index, the maps, and the commands. A new contributor can find every part without searching the tree.

GATE DISCIPLINE

- A red gate blocks the next slice.
- A test may be added, but a green test may not be deleted to make a red gate pass.
- A spec change requires a gate update.
- A benchmark number requires a recorded machine, commit, and digest.
- `bend PROOF.bend` green never excuses a red runtime gate.

E.14 · bend-gme/version-1/.opencode/skills/research-to-release/references/12-hvm5-v1-acceptance.md

11 — Acceptance matrix

Status: normative for HVM5 v1.

1. DEFINITION OF DONE

The v1 goal is met when the following are all true in one run of `bend hvm5/version1/src/sim.bend`:

1. ten nodes start and report their listen sets;
2. three subspaces have three replicas each;
3. the system contract is replicated on `n0,n1,n2`;
4. a signed package deploys into `root/public`;
5. a signed package deploys into `root/private`;
6. a public send changes state and returns a receipt;
7. a private send changes state and returns a receipt;
8. the public instance cannot reach the private instance, and the witness is null;
9. one public node is killed and the remaining two commit;
10. the killed node restarts and catches up to the same hash;
11. a code-as-data patch produces a new package hash;
12. a fork is created and the live branch continues to serve;
13. the fork test replays inputs and reports no differences;
14. merge flips the visible root atomically and drains the old branch;
15. every subspace reports one committed state hash;
16. the proof gate is green.

2. EXECUTABLE TESTS

ID	Test	Gate
A1	`tests/eval.bend` APP-LAM, APP-SUP, DUP-SUP	`EVAL True`
A2	`tests/eval.bend` stuck/label boundary	`EVAL True`
A3	`src/sim.bend` replica count and listener topology	`REPLICAS-OK True`
A4	`src/deploy.bend` source → automatic inet compile	`DEPLOY Counter{0} PKG <u32> DIGEST <u32>`
A5	`tests/deploy_compile.bend` auto-compile + send	`DEPLOY-COMPILE done Counter{2}`
A6	`tests/run.bend` genesis, authority-shaped deploy	`PUB-OK True / PRIV-OK True`
A7	`tests/run.bend` ten nodes, quorum, failover, catch-up	`QUORUM-OK True / CATCHUP-OK True`
A8	`tests/run.bend` isolation, bridge, capability gate	`ISOLATED True / BRIDGE-OK True / CAP-OK True`
A9	`tests/run.bend` fork, test, merge	`PUB Counter{4}`
A10	`tests/run.bend` deterministic state digests	`PASS`
A11	`src/sim.bend` full scenario output	`PASS`
A12	`bend PROOF.bend` for the v1 model	`All terms check.`

3. ADVERSARIAL TESTS

ID	Attack	Expected
X1	forged capability token	`Reject`
X2	replay a deploy envelope	`Reject{Sequence}`
X3	wrong key	`Reject{Signature}`
X4	deploy into a space the owner does not own	`Reject{Authority}`
X5	send a linear capability twice	second `Consumed`
X6	public node reduces a private pair	`Denied{Space}`
X7	bridge without two approvals	`Reject{Bridge}`
X8	fuel exhaustion	`OutOfFuel`, old state
X9	divergent replica hash	`ReplicaDivergence`
X10	merge crash before CAS	old root
X11	merge crash after CAS	new root
X12	old epoch token	`StaleEpoch`

4. CURRENT STATUS (2026-10-20)

This section is deliberately not aspirational.

Implemented:

- `Term`, `Def`, `Program` as Bend 2 `Data` types;
- capture-free substitution under globally unique binder names;
- a CEK-style machine with an explicit frame stack and fuel mode;

- APP-LAM, APP-SUP, DUP-SUP (same label), ground-data `Dup`, `Num` / `Add` / `Eq`, `Tag0/1/2`, `Match0/1/2`, `Ref`;
- a postfix inet package format and a Bend 2 parser;
- a Bend 2 contract subset compiler (`src/bendc.bend`) that parses constructor names, field binders, parameter names, and numeric literals, and is called automatically by deploy (`compile_for_deploy`), plus the bootstrap frontend;
- a checked Bend 2 contract (`contracts/counter.bend`);
- end-to-end: `counter.bend` → inet package → reducer → `Counter{2}`;
- native BendRT build of `run.bend` executes the same package;
- a ten-node pure model: three sys, three public, three private replicas, plus a client; quorum, kill, heal, catch-up;
- deploy from an actual Bend 2 source file with automatic inet compilation;
- public and private deploy, send, isolation without a bridge;
- a negative bridge test (`BRIDGE-OK`);
- fork/test/merge over the inet package;
- a repeated-send benchmark (`BENCH sends=100 ...`);
- closed topology/isolation laws in `LAWS.bend` / `PROOF.bend`.

Not yet implemented:

- full DUP-LAM and different-label DUP/SUP commutation;
- a first-class agent/port/wire heap (HVM4-style terms are current);
- package hashing and native Ed25519 signed envelopes in the v1 pipeline;
- a true process/network cluster and WAL persistence;
- native crypto adapters, or a Worker ABI;
- the stable `hvm5.core.v1` file and full generic lowering; the current official-AST emitter and `bendc.bend` source parser cover the first-order contract fragment only.

The old TypeScript files are in `legacy/ts-oracle/`. They are **not** the runtime and are not imported by `src/`.

5. WHAT IS PARTIAL OR FUTURE

Area	Current	Required for v1
HVM5 term evaluator	Bend 2 CEK machine; APP-LAM/SUP, DUP-SUP, Add, Tag/Match pass	full tested IC fragment, DUP-LAM
Inet graph kernel	semantic spec; term representation	`src/hvm5/graph.bend + rules.bend`
Bend 2 → inet	official frontend + `tools/emit-inet.ts` first-order fragment	stable core IR + broader lowering
Contract model	`Counter` and `Tally` end-to-end	`Vault` and more ADTs
Cluster	ten-node pure model, replicas, quorum, catch-up	real processes/network
Native wire opacity	numeric capability model	native `law` handles
WASM Worker	v0 prototype	single-thread WASM ABI
Persistence	none	WAL, fsync, crash injection
Consensus	hash-quorum model working	process-level failover
Key management	deterministic numeric model	host Ed25519 adapter
Benchmarks	W3 repeated-send working	W1/W5 plus pinned hardware

The distinction matters. The term interpreter is a real Bend 2 program on BendRT. It is not yet an inet-native graph runtime, and the cluster is not yet implemented in Bend 2.

6. HONEST STATEMENT

A passing v1 run proves:

- the partition and capability model is executable;
- a signed deploy and a fork/merge lifecycle work under node failure;

- the public/private isolation claim has a runtime witness;
- the reducer is deterministic and fuel-bounded on the tested fragment.

It does **not** prove:

- that the Bend 2 reference interpreter is a high-performance HVM5;
- that the native membrane is memory-safe;
- that a C effect, the OS sandbox, or Ed25519 is correct;
- that a Bend 2 source program outside the contract fragment compiles;
- that the cluster survives a Byzantine leader.

Those are separate gates. They are listed in `docs/02-plan/03-risks.md`.

E.15 · `bend-gme/version-1/.opencode/skills/research-to-release/references/13-hvm5-v1-traceability.md`

12 — Traceability: knowledge base to requirements

Status: informative map for v1. It records which ideas were extracted from `knowledge/` and where they landed in the spec and implementation.

1. SOURCE FAMILIES

Source	What it contributed
`knowledge/09-hvm5-cluster-runtime/`	homoiconic substrate, fork/merge, registry, OCap, layers, API, update flows, similar projects
`knowledge/07-smart-contracts-dynamic/`	Kindelia stateful functions, hub fork/upgrade, signed runs, metering, transition purity
`knowledge/04-runtime-bendrt-hvm5/`	HVM5-mini reference, BendRT memory/scheduler, fuel, laws gate
`knowledge/03-hvm-inet-theory/`	interaction nets, symmetric combinators, confluence, optimal reduction
`knowledge/08-roadmap-hvm5-spec/`	knowns vs gaps, reference index, spec TODO
`docs/tutorial/`	explicit Bend 2 syntax, laws/proofs, ownership, subspaces of the language
`hvm5/reference/hvm5-mini/`	executable term evaluator and proof gate

2. IDEA MAP

Extracted idea	Requirement	Spec
Code and running graph share one representation	Package/state/patch are graphs	02 §1
Hole is an affine free root	Typed holes and fills	02 §3
Fill is an atomic link	Graft atomicity	02 §4, 06 §6
`dup` = fork	Fork duplicates live graph	02 §6, 06 §4.2
`sup` = merge candidate	Superposition branch and collapse	02 §6–8
Labels control correlation	Same-label DUP/SUP annihilate	01 §3, §7
Never duplicate a duplicator	Type/label restriction on fork	01 §3.4, 03 §3
Old branch drains after cutover	Draining state	06 §4.2, §8
Immutable code hash	PackageHash and registry	02 §2, 05 §3
Mutable name pointer	NameRecord sequence	05 §3.2
Resolve once, pin hash	Name is discovery only	05 §3–4

Extracted idea	Requirement	Spec
Capability is a wire, not a name	Capability type and membrane	03 §3, 05 §8
Linear capability cannot be copied	Linear gate	03 §5, 09 §4.5
Eventual send and reply wire	Send and receipt	06 §3
Turn/vat isolation	Instance and subspace isolation	03 §4–5, 04
Telemetry/metering	Fuel and receipts	01 §6, 10
Proof gate before merge	Merge requires green laws	06 §4.2, 11
State migration is explicit	`migrate` type and alias flip	03 §6, 06 §4.1
Current/old dual generation	Drain and epoch	06 §4.2, §8
Registry content layer	Package blob store	05 §3.1
Registry name layer	Signed pointer	05 §3.2
System contract as genesis	Genesis and authority rows	05 §4–5
Signed envelopes	Ed25519, sequence, nonce	05 §2
Partitioned inet	Space tree and induced subnet	04 §1–2
Listener authorization	Node listen set	04 §3, 07 §9
Three replicas per subspace	Ten-node topology	04 §4, 07 §2
Node failure without downtime	Quorum and failover	04 §6, 07 §7
Isolation as no capability path	Reachability and witness	03 §5
Bridges are explicit	Two-sided approval	04 §8
Type-level shape constraints	Space-indexed types	03 §2, §4
Benchmark with pinned machine	Reproducible runs	10 §3

3. IDEAS DELIBERATELY NOT ADOPTED

Idea	Why
Global mutable namespace	Violates no-ambient-authority
Runtime `grant` / `revoke` primitives	Grant is wire-passing; revoke is sever
Facet type as a first-class runtime value	Facets are separate capabilities
Blockchain consensus in v1	Separate layer; v1 is one logical net
Native tenant code	Escapes the net and the fuel model
Copying a linear wire through DUP	Destroys revocation
Treating `PROOF.bend` as a security proof of C/OS	Different trust boundary
Book-index isolation as the security gate	Books are a prototype, not a boundary
Silent different-label DUP/SUP commutation	Requires a separate confluence argument

4. REFERENCE INDEX

The v0 research index is kept at `knowledge/08-roadmap-hvm5-spec/references-index.md`. The v0 spec and plan are kept at `spec/legacy/` and `plan/legacy/`. They remain useful for the detailed comparative research and the original slice wording, but this v1 set wins on conflict.

5. MAINTENANCE RULE

When an idea from `knowledge/` changes the runtime:

1. update the spec section;
2. update this table;
3. add or change an executable test;
4. record the decision in `docs/02-plan/03-risks.md` if it changes a security claim.

A knowledge note is never normative by itself. A spec section plus a test is.

E.16 · bend-gme/version-1/.opencode/skills/research-to-release/templates/01-artifact-templates.md

Templates (copy a section, fill the brackets, delete what does not apply)

RESEARCH NOTE — DOCS/01-RESEARCH/NN-<TOPIC>.MD

```
# NN · <topic>
Question: <the one question this answers>
Verdict: <one paragraph; label proved / measured / designed / sourced>
| Claim | Evidence (source or command, date) | Consequence for us |
|---|---|---|
Open: <what is still unknown, and how to find out>
```

EVIDENCE CHECK — DOCS/01-RESEARCH/EVIDENCE/<TOPIC>-YYYY-MM-DD.MD

```
# <topic> (checked YYYY-MM-DD)
| Target | Command / URL | Result | Meaning |
|---|---|---|---|
Re-check: <when / which release checklist item>
```

MAP — MAPS/NN-<NAME>.MD

```
# <Goal | System | Types | Spaces | Lifecycle | Code | Risks> map
| Concept | Defined in (spec §) | Implemented in (module) | Checked by (test/law) |
|---|---|---|---|
```

SPEC SECTION — DOCS/03-SPEC/NN-<AREA>.MD

```
## N. <name>
N.1 <Normative statement with MUST / MUST NOT; one testable fact per line.>
Test: <ID> · Law: <ID or none> · Budget: <number + unit or none>
```

ADR — DOCS/02-PLAN/02-DECISIONS.MD

```
### ADR-NNN <decision>. <Proposed | Accepted | Superseded by ADR-MMM>
Context: <forces, facts with sources>
Decision: <what we do>
Consequences: <costs, risks, what becomes easy or hard>
```

SLICE — DOCS/02-PLAN/01-BACKLOG.MD

```
| ID | Task | Size S/M/L | → deps | Acceptance (observable, testable) |
```

GATE

```
Gate <ID>: <name>
Command(s): <exact>
Pass when: <exact output / threshold>
Owner: <agent>   Blocks: <slice IDs>
```

LAW DRAFT (FOR USER APPROVAL)

```
Law <ID> <name>
In words: <one sentence a user can approve>
Formal (LAWS.bend): law <name>: for x: T {lhs == rhs : T}
Why it matters: <the failure it rules out>
Not covered: <cost, host, FFI, other inputs>
```

SUBAGENT BRIEF

```
Role: <researcher | bend-dev | ui-dev | prover | reviewer>
Goal: <one sentence>
Context: <what the repo is; what was decided; links to spec §>
Read first: <skills to load; files to read>
Inputs: <paths>      Outputs: <paths it may create/edit — and nothing else>
Acceptance: <commands that must pass; observable results>
Constraints: <no new deps, no scope change, budgets, honesty labels>
Report: files changed; commands run with results; open issues; claims labeled proved/measured/designed
```

REVIEW — REVIEW.MD STYLE

```
# Review of <thing>, YYYY-MM-DD
Verdict: <one paragraph>
Keep: <what holds, with evidence>
Corrections: C1 <title> — problem / evidence / fix ...
Stop claiming: <sentences that overstate the evidence>
```

NOTES.MD ENTRY

```
## YYYY-MM-DD <slice ID> <title>
Built: <what>   Measured: <numbers, machine, versions>   Deviations: <from spec, with ADR link>   Next: <slice IDs>
```

PART F · Bend2 / HVM5 / INET knowledge base (bend-runtime/knowledge, verbatim)

A 49-file knowledge base built 2026-09-26 from the bend 2.0.25 CLI and GitHub: start-here (overview, glossary, how-to), the Bend 2 language digest, the full CLI dumps (guide/base/effects/shaders/help), the HVM/INET theory set (foundations, symmetric combinators and their rules, confluence/optimal reduction, HVM runtime tags/memory, Bend→HVM compilation and fuel), the BendRT/HVM5 runtime set (architecture, 64-bit heap with no GC, schedulers, FFI determinism, the hvm5-mini port, the proof gate), the ecosystem notes (bendlang/bend, HigherOrderCO, Victor Taelin, papers), runnable examples, the dynamic-smart-contract/Kindelia material, the HVM5 spec roadmap, the hvm5-cluster documents, and the older ALL-COMBINED rollups that preserve files the reorganisation dropped.

Source root: `bend-runtime/knowledge` · **60 files · 820.6 KB**

▼ In this part (file list)

- **F.1** · `bend-runtime/knowledge/README.md`
- **F.2** · `bend-runtime/knowledge/00-start-here/glossary.md`
- **F.3** · `bend-runtime/knowledge/00-start-here/how-to-use-this-knowledge-base.md`
- **F.4** · `bend-runtime/knowledge/00-start-here/overview.md`
- **F.5** · `bend-runtime/knowledge/01-bend-language/io-effects-apps-modules.md`
- **F.6** · `bend-runtime/knowledge/01-bend-language/laws-proofs.md`
- **F.7** · `bend-runtime/knowledge/01-bend-language/match-recursion-arrays-parallelism.md`
- **F.8** · `bend-runtime/knowledge/01-bend-language/quantities-kinds-templates.md`
- **F.9** · `bend-runtime/knowledge/01-bend-language/syntax-reference.md`
- **F.10** · `bend-runtime/knowledge/02-guides-cli/cli-tooling-limits.md`
- **F.11** · `bend-runtime/knowledge/02-guides-cli/guides-index.md`
- **F.12** · `bend-runtime/knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md`
- **F.13** · `bend-runtime/knowledge/02-guides-cli/raw/BASE-TYPES-DUMP.md`
- **F.14** · `bend-runtime/knowledge/02-guides-cli/raw/CLI-HELP-DUMP.txt`
- **F.15** · `bend-runtime/knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md`
- **F.16** · `bend-runtime/knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md`
- **F.17** · `bend-runtime/knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md`
- **F.18** · `bend-runtime/knowledge/03-hvm-inet-theory/01-interaction-nets-foundations.md`
- **F.19** · `bend-runtime/knowledge/03-hvm-inet-theory/02-symmetric-combinators-rules.md`
- **F.20** · `bend-runtime/knowledge/03-hvm-inet-theory/03-confluence-optimal-reduction.md`
- **F.21** · `bend-runtime/knowledge/03-hvm-inet-theory/04-hvm-runtime-tags-memory.md`
- **F.22** · `bend-runtime/knowledge/03-hvm-inet-theory/05-bend-compilation-fuel.md`
- **F.23** · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/00-overview-architecture.md`
- **F.24** · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/01-memory-model-no-gc.md`
- **F.25** · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/02-schedulers-cpu-gpu.md`
- **F.26** · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/03-ffi-determinism-io.md`
- **F.27** · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/04-hvm5-mini-reference-port.md`
- **F.28** · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/05-laws-proof-gate-cli.md`
- **F.29** · `bend-runtime/knowledge/05-ecosystem/bendlang-bend-repo.md`
- **F.30** · `bend-runtime/knowledge/05-ecosystem/higherorderco-org.md`
- **F.31** · `bend-runtime/knowledge/05-ecosystem/papers-formalization-community.md`
- **F.32** · `bend-runtime/knowledge/05-ecosystem/victor-taelin.md`
- **F.33** · `bend-runtime/knowledge/06-examples/01-hello-area-adder.bend`
- **F.34** · `bend-runtime/knowledge/06-examples/02-recursion-parallel-arrays.bend`
- **F.35** · `bend-runtime/knowledge/06-examples/03-io-channels-tcp-files.bend`
- **F.36** · `bend-runtime/knowledge/06-examples/04-laws-proofs-app-loop.bend`
- **F.37** · `bend-runtime/knowledge/06-examples/05-bench-and-proof-templates.md`
- **F.38** · `bend-runtime/knowledge/07-smart-contracts-dynamic/01-kindelia-model.md`

- **F.39** · `bend-runtime/knowledge/07-smart-contracts-dynamic/02-hub-fork-upgrade.md`
- **F.40** · `bend-runtime/knowledge/07-smart-contracts-dynamic/03-domain-types-hvm5-checklist.bend`
- **F.41** · `bend-runtime/knowledge/08-roadmap-hvm5-spec/references-index.md`
- **F.42** · `bend-runtime/knowledge/08-roadmap-hvm5-spec/spec-todo.md`
- **F.43** · `bend-runtime/knowledge/08-roadmap-hvm5-spec/what-we-know-vs-gaps.md`
- **F.44** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/01-concepts.md`
- **F.45** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/02-architecture-layers.md`
- **F.46** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/03-domain-types.md`
- **F.47** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/04-api.md`
- **F.48** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/05-registry-naming.md`
- **F.49** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/06-update-fork-merge.md`
- **F.50** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/07-ocap-wires.md`
- **F.51** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/08-similar-projects.md`
- **F.52** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/09-implementation-plan.md`
- **F.53** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/10-meta-prompt-for-builder-llm.md`
- **F.54** · `bend-runtime/knowledge/09-hvm5-cluster-runtime/README.md`
- **F.55** · `bend-runtime/knowledge/ALL-COMBINED.md`
- **F.56** · `bend-runtime/knowledge/ALL-COMBINED-MINI.md`
- **F.57** · `bend-runtime/knowledge/ALL-COMBINED-MINI-KINDELIA.md`
- **F.58** · `bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.md`
- **F.59** · `bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.META-PROMPT.md`
- **F.60** · `bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.SOURCES.md`

F.1 · `bend-runtime/knowledge/README.md`

Bend2 / HVM5 / INET Knowledge Base

Comprehensive overview of Bend2, HVM5/BendRT, interaction nets, runtime, interfaces, and dynamic smart-contract patterns. Built 2026-09-26 from CLI (`bend 2.0.25`) + GitHub sources.

LAYOUT

- `00-start-here/` — overview, how to use, glossary
- `01-bend-language/` — syntax, types, parallelism, proofs, IO, apps, modules
- `02-guides-cli/` — full `bend guide` / `shaders` / `effects` dumps in `raw/` + summaries, CLI tooling, Base library
- `03-hvm-inet-theory/` — interaction nets, combinators (dup/sup/era/con), optimal reduction, HVM lineage, compilation
- `04-runtime-bendrt-hvm5/` — BendRT architecture, HVM5-mini, memory, schedulers, FFI, determinism
- `05-ecosystem/` — VictorTaelin, HigherOrderCO, bendlang/bend, papers, community
- `06-examples/` — curated demos, benches, templates
- `07-smart-contracts-dynamic/` — Kindelia model, hub content-addressing, fork/upgrade, domain types, HVM5 API checklist
- `08-roadmap-hvm5-spec/` — knowns vs gaps, spec TODO, reference index

SOURCES (ALL FETCHED, NO INVENTED URLS)

- CLI: `bend guide` , `bend guide shaders` , `bend guide effects` , `bend base` , `bend base --types` , `bend --help` (bend 2.0.25) — raw dumps in `02-guides-cli/raw/`
- https://github.com/VictorTaelin (https://github.com/VictorTaelin)
- https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) (org) + HVM1/HVM2/HVM3/HVM4/Bend1/Kind
- https://github.com/bendlang/bend (https://github.com/bendlang/bend) (live Bend2 repo: `bend2/` , `demos/` , `guide/` , `paper/` , `bench/` , `evals/`)
- https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) (esp. `pure_hvm5_mini` , `pure_par_*` , `proof_*` , `app_*` , `io_*`)
- https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) + WHITEPAPER/WHITEBOOK
- https://bend-lang.com (https://bend-lang.com), https://hub.bend-lang.com (https://hub.bend-lang.com)
- Theory: https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets), Lafont POPL 1990 / 1997, HVM2 paper, BendTT/BendRT PDFs

> Bend 2 is a new language. Bend 1 programs and HVM do not carry over. (bendlang/bend README Limitations)

QUICK START

1. Read `00-start-here/overview.md`
2. Run `bend guide` yourself; compare with `02-guides-cli/raw/GUIDE-CLI-DUMP.md`
3. Pick a track: language (`01-`), runtime (`03-` + `04-`), contracts (`07-`)
4. Spec work starts at `08-roadmap-hvm5-spec/spec-todo.md`

F.2 · `bend-runtime/knowledge/00-start-here/glossary.md`

Glossary

BEND2 LANGUAGE

- **Affine**: use at most once (default). Dropping is free.
- **Quantity**: `-x` erased (types/proofs only), `x` affine, `+x` reusable (needs `Data`).
- **Kind**: `Type = Kind(&1)` linear, `Data = Kind(&2)` copyable, `Kind(a)` generic, `a <&> b` minimum.
- **Template (`~f`)**: compile-time inlined arg, must be closed, per-site copy. How `List.map` works.
- **Law**: `law name: for x: T … claim` — human-owned open claim in `LAWS.bend`.
- **Proof**: `def name(…)` of law's type — AI-owned in `PROOF.bend`. `{==}` reflexivity, `%e : P` rewrite (`_` = new side), `?name` / `?TODO` holes.
- **Parallel let**: `a b = f(x) g(y)` — fork/join, promises independence + balance.
- **Bang**: `f!(x)` — run subtree on GPU (else CPU threads).
- **Array**: `Array<T> : Type`, single owner, `[v : T*n]` / `[v : T^d]`, `a[i]` / `a[i] <- v`, read returns `Array & elem`.
- **do-block**: `do M<xs..,R>:` over `M.bind` / `M.pure`; `x : A <- m`, `x : A = v`, `return`.
- **Handle**: affine opaque (`File/Socket/Window/Chan`), handed back beside result, unforgable.

INET / HVM

- **Agent/cell/node**: 1 principal + n aux ports. Only principal–principal pairs rewrite.
- **Active pair / redex**: principal–principal connection; unit of parallel work.
- **CON (`γ`)**: constructor (Bend1: `( )`; IC: LAM/APP side).
- **DUP (`δ`)**: duplicator; `let {a b} = x` (consume) vs `SUP {a b}` (build). Same node, polarity decides.
- **SUP**: superposition `{a b}` — parallel alternatives distributing over consumers.
- **ERA (`ε`, `*`)**: eraser; propagates + frees.
- **REF (`@foo`)**: top-level def pointer; CALL expands, VOID deletes, COPY fast-path.
- **NUM/OPR/SWI**: native numbers, prim ops, zero-vs-succ branch.
- **Annihilate**: same family meets → cancel, auxes wire straight.
- **Commute/copy**: different families meet → duplicate through each other.
- **Oracle/brackets**: Lamping bookkeeping to match dups; avoided by affine restriction.
- **Beta-optimal**: Lévy family sharing; HVM hits it on affine/higher-order fragment.
- **Fork-join**: BendRT scheduler unit; join waits for both children.
- **Unified heap**: one address space for CPU+GPU; no separate copies.
- **BendRT**: Bend2 flat evaluator (not interaction-net reducer). One C file → clang/Metal/CUDA.
- **HVM5-mini**: `demos/pure_hvm5_mini` — minimal pure runtime model in Bend.

CONTRACTS

- **ctr/fun/run/reg**: Kindelia statements (define ctor, stateful fn, metered eval, namespace).
- **mana/bits**: compute cost vs heap growth limits.
- **Registry/dispatch**: stable name → current code hash pointer for upgrades.
- **Fork**: new package importing old hash + overriding defs.
- **Migration**: explicit `V1.State -> V2.State` total function + law.

F.3 · bend-runtime/knowledge/00-start-here/how-to-use-this-knowledge-base.md

How to use this knowledge base

FOR LEARNING BEND2

1. Install: `curl -fsSL https://bend-lang.com/install.sh | sh`
2. `bend guide` (compare `02-guides-cli/raw/GUIDE-CLI-DUMP.md`), `bend guide shaders`, `bend guide effects`
3. `bend base --types` then `bend base Map` for the `Type.verb` scheme
4. Run `demos/io_hello_world`, `demos/pure_par_sum`, `demos/proof_insertion_sort` (see `06-examples/`)

FOR RUNTIME / HVM5 SPEC WORK

1. `03-hvm-inet-theory/` for nets + combinators + optimal reduction
2. `04-runtime-bendrt-hvm5/` for BendRT, memory, schedulers, FFI
3. `07-smart-contracts-dynamic/` for Kindelia + fork/upgrade patterns
4. `08-roadmap-hvm5-spec/spec-todo.md` for gaps before writing the spec

CONVENTIONS

- `(S)` = sourced fact with URL; `(D)` = design suggestion, not spec.
- No invented URLs: every link was fetched 2026-09-26 or comes from CLI dumps.
- `bend guide` text is canonical for syntax; GitHub `guide/GUIDE.md` mirrors it: https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)
- Bend1/HVM CLIs (`bend run-c`, `hvm run`) do NOT apply to Bend2.

REFRESHING

```
bend guide > bend2-knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md
bend guide shaders > bend2-knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md
bend guide effects > bend2-knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md
bend base > bend2-knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md
bend version
```

F.4 · bend-runtime/knowledge/00-start-here/overview.md

Overview — what Bend2 / HVM5 / INETs are

ONE-PARAGRAPH VERSION

Bend2 is a Python-shaped, Haskell/Lean-semantics, Rust-like affine language with dependent types, mandatory termination, and explicit parallelism (`a b = f(x) g(y)` fork/join, `f!(x)` GPU). **BendRT** is its parallel CPU/GPU runtime (one C file = CPU program + GPU kernel, single shared heap, no GC, 64-bit words). **HVM5** exists in-repo only as `demos/pure_hvm5_mini` — a minimal pure model, not a released standalone runtime. **Interaction nets/combinators** (Girard/Lafont: CON/DUP/SUP/ERA, annihilate vs commute) are the historical HVM1–HVM4 execution substrate; BendRT keeps the affine/no-GC/parallel ideas but is a flat evaluator, not an interaction-net reducer. **Kindelia** is the prior functional blockchain (stateful pure functions + metered `run`, mana/bits, namespaces) and the template for dynamic Bend2 contracts via content-hashed hub imports (`import 0x<hash>/main.bend`).

GENERATIONS (DO NOT MIX)

Gen	Repo	Runtime	Status
HVM1 (2022)	https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1)	Rust, lazy, parallel, beta-optimal fragment	Archived predecessor
HVM2 (2024)	https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)	Rust/C/CUDA IR, 7 node kinds	Bend1 target, LTS paper

Gen	Repo	Runtime	Status
HVM3	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3)	Haskell, Interaction Calculus	WIP
HVM4	https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)	C single-file IC runtime	Experimental
Bend1	https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1)	On HVM2	Superseded
Bend2	https://github.com/bendlang/bend (https://github.com/bendlang/bend)	**BendRT** (TS compiler, C/Metal/CUDA/JS)	**Live (bend 2.0.25)**
Kindelia	https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia)	HVM opcodes, PoW, 1s blocks	WIP reference for contracts

CORE CLAIMS

- Pure + affine by default (`is Data` copyable, `is Type` linear). `match` frees consumed node; only `+` values refcounted. No GC.
- Termination mandatory (structural recursion, decreasing arg first); `@unsafe / def f?` escapes proofs.
- Laws (`LAWS.bend` , human) + proofs (`PROOF.bend` , AI) gated by `bend PROOF.bend` .
- Parallelism is explicit + must be balanced (binary fork-join, contention-free; GPU cube ~16384 lanes, aim 4^7 leaves).
- Effects only via `IO` + foreign `.c` / `.js` twins; host event loop; JS can import `.bend` .
- Packages are content hashes on https://hub.bend-lang.com (https://hub.bend-lang.com); `bend --publish` prints the `import 0x…` line.

WHERE TO GO NEXT

- Language: `01-bend-language/`
- CLI + Base: `02-guides-cli/cli-tooling.md` , `base-library.md`
- Deep INET: `03-hvm-inet-theory/`
- Runtime + HVM5: `04-runtime-bendrt-hvm5/`
- Contracts/forking: `07-smart-contracts-dynamic/`
- Spec work: `08-roadmap-hvm5-spec/`

F.5 · `bend-runtime/knowledge/01-bend-language/io-effects-apps-modules.md`

IO, Effects, Apps, and Modules

Sources verified 2026-09-28:

- `bend guide / guide/GUIDE.md` (IO and Concurrency, Monads, Apps, Modules, Base Library, Tooling) on https://github.com/bendlang/bend (https://github.com/bendlang/bend)
- `bend base / bend2/base.bend` (`IO` , `Chan` , `File` , `TCP` , `UDP` , `Window` , `Audio` , `App` , `Process.run`)
- `bend guide effects` and `bend guide shaders` (custom effects, efficient shaders)
- Repo README Get Started + Limitations (effects list, targets, hub/packages)

1. IO MODEL: PURITY + MONAD

Bend is pure; effects are denoted via a Haskell-inspired `IO` monad and sequenced with `do` blocks. Only the event loop runs host code, so proofs, termination, and the GPU never touch it.

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

`do` desugars each `x : A <- v` into `M.bind(.., v, x => ..)` and each `return e` into `M.pure(.., e)`, so any type with those two defs works (`IO`, `Maybe`, `Result`, or your own). Leading quantity args (e.g. `&2`) pass through to both.

2. CORE DO PATTERNS

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1000)             # a step
    return "Hello, " ++ name   # return: wraps a pure value

def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER")) # <-: binds a result
    chan : Chan(String) <- IO.fork(String, greet(name)) # runs concurrently
    IO.print("Waiting...")
    text : String <- IO.join(String, chan)
    IO.print(text)
```

Rules:

- Every bind is annotated (`name : String <- ...`).
- `x : T = v` binds a pure value mid-block.
- `m` alone is a `Unit` step (e.g. `IO.print(..)`, `IO.sleep(..)`).
- `return v` is the result.

Fallible effects:

```
import Base

def read_user() -> IO(String):
  do IO<String>:
    # IO.get_env answers Result<.., String>; IO.try unwraps or exits
    name : String <- IO.try(String, IO.get_env("USER"))
    return name

def die_demo() -> IO(Unit):
  do IO<Unit>:
    IO.die(Unit, 1, "boom")

def args_demo() -> IO(Unit):
  do IO<Unit>:
    args : List<String> <- IO.args()
    IO.print(String.join(args, " "))
```

- `Result<&1,&1,U32 & String,A>`: `IO.try` unwraps or exits with the error; `IO.die` exits with your own.
- `IO.args` is the command line less runtime options; head is the program as invoked (like C `argv[0]`); `--` ends runtime options.
- A handle (`File`, `Socket`, `Window`) is affine and opaque: every effect hands it back beside its result; no program can forge or reuse one.

3. CONCURRENCY: FORK/JOIN, SPAWN, CHANNELS

A Bend program is computations interleaved by one event loop (like Node.js): each runs pure code (in parallel, on every core) up to its next effect; one waiting on a socket/sleep/channel steps aside.

```
import Base

def worker(x: U32) -> IO(U32):
  do IO<U32>:
    IO.sleep(10)
    return (x * 2 : U32)

def main() -> IO(Unit):
  do IO<Unit>:
    c1 : Chan(U32) <- IO.fork(U32, worker(21))
    c2 : Chan(U32) <- IO.fork(U32, worker(22))
    a : U32 <- IO.join(U32, c1)
    b : U32 <- IO.join(U32, c2)
    IO.print(U32.show((a + b : U32)))
```

Primitives (see `bend2/base.bend`):

- `IO.fork(A, act) -> IO(Chan(A))` : start computation, get result channel.
- `IO.join(A, chan) -> IO(A)` : wait for it (dies if channel closed).
- Underneath: `IO.spawn` , `Chan.new` , `Chan.send` , `Chan.recv` , `Chan.close` .
- `IO.within(A, ms, act)` races `act` vs deadline, answers `None{}` if deadline wins; the loser is not cancelled.

```
import Base

def timeout_demo() -> IO(Unit):
  do IO<Unit>:
    res : Maybe<U32> <- IO.within(U32, 100, worker(7))
    match res:
      case None{}:
        IO.print("timed out")
      case Some{v}:
        IO.print(U32.show(v))

def chan_demo() -> IO(Unit):
  do IO<Unit>:
    ch : Chan(String) <- Chan.new(String, 1)
    ok : Bool <- Chan.send(String, ch, "hi")
    m : Maybe<String> <- Chan.recv(String, ch)
    Chan.close(String, ch)
```

Program ends when every computation is done, or reports deadlock when all remaining ones wait.

4. EFFECT CATALOG (BASE)

From `bend base` (full list; see `raw/BASE-FULL-DUMP.md` untouched):

- Console/time/random: `IO.print` , `IO.write` , `IO.print_err` , `IO.sleep` , `IO.now` , `IO.random_u32` , `IO.thread_count` , `IO.get_env` , `IO.args` .
- Processes: `Process.run(program, args, input, max_output, timeout_ms)` — starts executable directly (no shell), inherits cwd/env, writes UTF-8 input to stdin, answers `Done{(status,(stdout,stderr))}` even on non-zero exit. Both limits positive; start failure/timeout/overflow answers `Fail` . Waits for direct child, drains ready output; timeouts do not kill descendants. Native runs child on IO helper thread; JS lane blocks. Linux native needs glibc 2.34+. No sandboxing — check trust before executing.
- Files: `File.open/read/read_bytes/read_at/size/write/write_bytes/close` .
- TCP: `TCP.listen/accept/connect/send/recv/send_bytes/recv_bytes/poll` , `Socket.close` , `Listener.close` .
  - `listen(host,port)` / `bind(host,port)` bind IPv4 literal: `"127.0.0.1"` this machine only, `"0.0.0.0"` every interface. Bad address or port > 65535 fails `EINVAL` . `max 0` fails `EINVAL` .
- UDP: `UDP.bind/send_to/recv_from/poll` .
- Window/Audio: `Window.open/frame/set_title/grab/close` , `Audio.open/write/close` .

File example (affine handle threading):

```
import Base

def cat(path: String) -> IO(Unit):
  do IO<Unit>:
    r : Result<&1, &1, U32 & String, File> <- File.open(path, "r")
    match r:
      case Fail{(code, msg)}:
        IO.print("open failed: " ++ msg)
      case Done{f}:
        (f2, out) = IO.try(File & Result<&1, &1, U32 & String, String>, File.read(f, 4096))
        # NOTE: File.read hands (File & Result); bind accordingly in real code
        File.close(f2)
```

TCP echo sketch:

```
import Base

def serve_once(listener: Listener) -> IO(Unit):
  do IO<Unit>:
    (l2, r) = TCP.accept(listener)
    match r:
      case Fail{(c, m)}:
        Listener.close(l2)
      case Done{sock}:
        (s2, msg) = TCP.recv(sock, 1024)
        (s3, ok) = TCP.send(s2, "hi")
        Socket.close(s3)
        Listener.close(l2)
```

5. CUSTOM EFFECTS (FOREIGN C + JS)

Every effect in Base is a def whose body is `import "./x.js"` plus a `.c` twin, implemented by a host function named after the def, lowercased, dots to underscores. Add your own the same way.

```
# myeff.bend
import Base

def My.ping(x: U32) -> IO(U32):
  import "./my_ping.c"
  import "./my_ping.js"
```

Read `bend guide effects` before writing one (C/JS side contract). Only the event loop runs them.

JS interop in the other direction: a JS file may `import Game from "./game.bend"` (with `bend2/main.ts` preloaded), or from the `./game.mjs` that `-o game.mjs` writes, and call every non-IO def, with constructors as `{$: "Name", field: value}` and `Nat` as `BigInt`. A value crosses without a copy: an `Array` argument is the caller's own array, updated in place — copy first if you keep it.

6. MONADS BEYOND IO

```
import Base

def add_strs(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a) # a None here ends the block with None
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> Maybe<&2, U32>:
  add_strs("40", "2")
```

Define your own by providing `pure` + `bind` with matching leading args.

7. APPS: PURE GRAPHICS

Graphics are pure: a frame is an `Image`; `App` maps states to images.

```
import Base

# tick: folds a frame's events into the next state; None quits the app
def tick(events: List<Event>, color: U32) -> IO(Maybe<U32>):
  match events:
    case Nil{}:
      IO.pure(Maybe<U32>, Some{color})
    case Con{Close{}, rest}:
      IO.pure(Maybe<U32>, None{})
    case Con{e, rest}:
      tick(rest, (color + 1 : U32))

def main() -> IO(Unit):
  # view: returns the state beside its image; Pix paints the whole frame
  App.run(~U32, ~App{+s => (s, Pix{s}), tick}, "Hello", 256, 256, 0)
```

- `Image` is a quadtree: `Pix{color}` paints a square; `Qua{tl,tr,bl,br}` splits in four. Draw by recursion, in parallel if you want.
- Events: `Key`, `Mouse`, `Move`, `Look`, `Scroll`, `Close`. `Scroll{x,y,dx,dy}` signed `F32` deltas toward page top/left (notch = 1 on X11).
- FPS camera: `Window.grab(window, True{})` hides/holds cursor; motion arrives as `Look{dx,dy}` until `Window.grab(window, False{})` or focus loss.
- `App.run` opens window, calls `view` then `tick` per frame until `None`. State is affine: `view` hands it back next to the image.
- Underneath: `Window.open/frame/close`, `Audio.open/write/close`.
- See `demos/app_pong_game_2d` for a complete app; read `bend guide shaders` before writing a graphical/parallel app.

Quadtree example:

```
import Base

def checker(d: Nat, c0: U32, c1: U32) -> Image:
  match d:
    case 0n:
      Pix{c0}
    case 1n+p:
      a b = checker(p, c0, c1) checker(p, c1, c0)
      # join four quadrants (illustrative; thread Images linearly)
      Qua{a, b, b, a}
```

8. MODULES, PACKAGES, PUBLISHING

```
# math.bend
import Base

def square(+x: U32) -> U32:
  (x * x : U32)
```

```
# main.bend
import Base
import ./math.bend as M  # M.x now names every def of math.bend

def main() -> U32:
  M.square(7)
```

- Alias is local to the importing file.
- Publish: `import 0x<hash>/main.bend as P` imports package by content hash (fetched from hub, checked against it); `bend main.bend --publish` uploads file + imports and prints that line.
- Named: `import <name>@<version>/main.bend as P`; publish via `bend main.bend --publish <name>@<version>` after `bend login`.
- Publish is public/permanent under BendHub terms (https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18)). Put a `LICENSE` file next to entry (ideally `SPDX-License-Identifier: MIT` first line); `--publish` bundles every file named exactly `LICENSE` beside a published file; no-LICENSE package is MIT-0. Adding `LICENSE` changes hash: publish as new version.

- Hub has no names/versions/accounts/search yet in early builds; packages are hashes (see Limits file).

9. BASE LIBRARY NAMING SCHEME

`bend base` prints all; `bend base --types` only types; `bend base Map` one name and everything under it.

- Every def is `Type.verb`; same verbs recur across `Nat`, `U32`, `F32`: `add sub mul div mod`, bits `and or xor not shl shr` (`U32` only), `cmp` (→ `Cmp`, not on `F32`), `is_eq is_ne is_lt is_le is_gt is_ge` (→ `Bool`), `show` ↔ `read` (→ `Maybe`).
- Beyond numbers: `Bool`, `Cmp`, `Maybe`, `Result`, `List`, `Array`, string-keyed `Map` (`new set get has del keys`; `get` takes default; `get` / `has` hand map back), `Set` on top, `Equal` lemmas, effects.

```
import Base

def main() -> String:
  +a = (6 * 7 : U32)         # sugar for U32.mul(6, 7)
  b  = U32.to_nat(a)         # conversions are T.to_x and T.from_x
  U32.show(a) ++ " = " ++ Nat.show(b)
```

F.6 · bend-runtime/knowledge/01-bend-language/laws-proofs.md

Laws and Proofs (Bend2)

Sources verified 2026-09-28:

- `bend guide` / `guide/GUIDE.md` (Laws and Proofs) on https://github.com/bendlang/bend (https://github.com/bendlang/bend)
- `bend2/base.bend` (`Equal.cong/sym/trans`, `Nat.ge_refl`, `Word.add_comm`, `U32.add_comm`)
- Repo README (`LAWS.bend` game demo, `bend PROOF.bend`, `--verdict` kernel, `bendtt.lean`, papers)
- `AGENTS.md` at repo root: run `bend guide`, use `LAWS.bend`, run `bend PROOF.bend` before committing

1. BIG PICTURE: LAWS BLOCK AI MISTAKES

Problem: how to trust AI code without reading it? Solution: force the AI to write a correctness proof.

Conventions:

- `LAWS.bend` imports the code and states laws (open claims). Human writes it; AI does not touch it.
- `PROOF.bend` imports `LAWS.bend` and proves each law with a same-named def (`law sorted` proven by `def Laws.sorted` / `def sorted`). AI writes it.
- `bend PROOF.bend` is the gate: `SOME PROOFS FAIL` while any law is open or false; `ALL PROOFS CHECK` once every law holds.
- Bend refuses a `PROOF.bend` beside a `LAWS.bend` without importing it.

README game example (winning is impossible):

```
# LAWS.bend
law you_cant_win:                          # "winning is impossible"
  for moves: List<Game.Move>               # any sequence of moves
  board = Game.replay(Game.start(), moves)  # replayed from the start
  {Game.is_won(board) == False{} : Bool}    # never leads to victory
```

```
# PROOF.bend
def Laws.you_cant_win(moves):
  # ... written by the AI
```

Law-driven development: instead of "implement sort", write "for every list, `F(list)` returns the same numbers in ascending order". Bend demands an actual proof, so models respond with bug-free code.

2. MINIMAL LAW + PROOF

```
import Base

# LAW: "for every x, x plus 0 equals x"
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

# PROOF: case analysis:
# - base case: reflexivity
# - step case: induction, rewrite, reflexivity
def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> {Nat.add(2n, 0n) == 2n : Nat}:
  add_zero(2n)
```

Mechanics:

- A proposition is a type; a proof is a def of that type.
- `{a == b : T}` is equality; `{==}` proves it when both sides compute to the same term (reflexivity).
- `match` refines the goal in each case; a recursive call is the induction hypothesis.
- `%e : P` rewrites with `e : {a == b : T}` : `P` is the goal with `_` marking `b` ; the goal becomes `P` with `a` there.
- `exs y: T` in a law asks for a witness, returned as `(y, proof)` .
- Failed step prints expected vs observed terms; `?name` prints the goal; `?TODO` leaves it open; a law with no def is an open claim.

3. REWRITES IN DETAIL

```
import Base

law add_zero2:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def add_zero2(x):
  match x:
    case 0n:
      {==}
    case 1n+xp:
      # IH: {Nat.add(xp, 0n) == xp : Nat}
      # Goal before rewrite: {1n+Nat.add(xp, 0n) == 1n+xp : Nat}
      # P marks where RHS-of-IH appears with _; rewrite LHS back in:
      %add_zero2(xp) : {1n+Nat.add(xp, 0n) == 1n+_ : Nat}
      {==}
```

Named rewrite variant from the syntax reference:

```
# %e : P; e2      — rewrite, then continue with e2
# %e@E : P; e2    — named rewrite
```

Equal lemmas from Base (`bend base Equal`):

```
law Equal.cong:
  for -A: Type
  for -B: Type
  for -f: A -> B
  for -a: A
  for -b: A
  for  e: {a == b : A}
  {f(a) == f(b) : B}

def Equal.cong(A, B, f, a, b, e):
  %e : {f(a) == f(_) : B}
  {==}
```

```
law Equal.sym:
  for -A: Type
  for -a: A
  for -b: A
  for  e: {a == b : A}
  {b == a : A}

def Equal.sym(A, a, b, e):
  %e : {_ == a : A}
  {==}
```

```
law Equal.trans:
  for -A : Type
  for -a : A
  for -b : A
  for -c : A
  for ab : {a == b : A}
  for bc : {b == c : A}
  {a == c : A}

def Equal.trans(A, a, b, c, ab, bc):
  %bc : {a == _ : A}
  ab
```

Using them:

```
import Base

law U32_add_comm:
  for a: U32
  for b: U32
  {U32.add(a, b) == U32.add(b, a) : U32}

def U32_add_comm(a, b):
  match a b:
    case U32{x} U32{y}:
      Equal.cong(Word(32n), U32, w => U32{w}, Word.add(32n, x, y),
        Word.add(32n, y, x), Word.add_comm(32n, x, y))
```

4. LAWS WITH `where`, `exs`, DEPENDENT PAIRS

From the syntax reference:

```
law f:
  for x: A
  for y: B where P(y)   # y is then the pair (y, proof of P(y))
  exs z: C              # witness the proof must return
  T
```

Example: every `Nat` has a double (existential):

```
import Base

law has_double:
  for x: Nat
  exs y: Nat
  {Nat.add(x, x) == y : Nat}

def has_double(x):
  (Nat.add(x, x), {==})
```

Refuting a clash via a motive (guide):

- To refute `e : {1n == 0n : Nat}`, rewrite through `disc(_)` where `disc` sends `0n` to `Empty` and `1n+p` to `Unit`, then answer `Unit{}`.

```
import Base

def disc(n: Nat) -> Type:
  match n:
    case 0n:
      Empty
    case 1n+p:
      Unit

def clash(e: {1n == 0n : Nat}) -> Empty:
  %e : {disc(1n) == disc(_) : Type}
  Unit{}
```

Closing an impossible branch:

```
import Base

def absurd_head(a, -A: Kind(a), e: Empty) -> A:
  match e:
    # no cases: e : Empty closes the branch

def not_succ_zero(e: {1n == 0n : Nat}) -> Empty:
  clash(e)
```

`{a != b : T}` is `{a == b : T} -> Empty`:

```
import Base

def zero_ne_succ(n: Nat) -> {0n != 1n+n : Nat}:
  e => clash_helper(e)
```

5. LAWS.BEND / PROOF.BEND WORKFLOW

```
project/
  main.bend
  LAWS.bend    # imports code, states laws
  PROOF.bend   # imports LAWS.bend, proves each law
```

```
# LAWS.bend
import Base
import ./main.bend as App

law sorted:
  for xs: List<U32>
  {List.is_sorted(App.sort(xs)) == True{} : Bool}

law preserves_length:
  for xs: List<U32>
  {List.length(&2, U32, App.sort(xs)) == List.length(&2, U32, xs) : Nat}
```

```
# PROOF.bend
import Base
import ./LAWS.bend as Laws

def Laws.sorted(xs):
  ?TODO

def Laws.preserves_length(xs):
  ?TODO
```

Gate commands:

```
bend PROOF.bend              # check; ALL PROOFS CHECK or SOME PROOFS FAIL
bend PROOF.bend --verdict    # recheck with proven BendTT kernel (bendtt.lean)
bend PROOF.bend -o PROOF.bendtt  # write translation the kernel reads
```

`--verdict` prints `ALL PROOFS CHECK` only when every def outside Base is a valid proof accepted by both `bend2` and the kernel, relying on no `@unsafe` or foreign code. The translation has no proof, so read it to confirm a law.

6. WHAT THE KERNEL GUARANTEES (AND DOES NOT)

From guide "Under the Hood" + README Limitations:

- Theory: one universe, no positivity check (`Type : Type` holds; datatypes may recurse left of arrow). Soundness comes from a live/dead wall:
  - Live (code that runs): checked live, recursion must terminate.
  - Dead (types, erased args, equations): may loop or inhabit `Empty` , but never counts as live evidence.
- `bend2/bendtt.lean` is the kernel in Lean, with proof that no accepted def has type `Empty` and live code halts.
- `F32` is axiomatic: nothing about floating point can be proven.
- No tactics or proof search; proving takes extra effort.
- The checker (not kernel) is 99% AI-written, unaudited; checker has no proof and may have bugs — `--verdict` uses the proven kernel.

7. DEBUGGING PROOFS

```
import Base

law demo:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

def demo(x):
  match x:
    case 0n:
      ?base_goal   # prints the goal at this point
    case 1n+p:
      ?TODO        # leaves it open; gate reports it
```

Tips:

- Read expected vs observed on a failed `% / {==}` .
- Ensure the induction hypothesis is in scope (recurse on a pattern variable).
- Annotate ambiguous terms `{e : T}` .
- Keep `LAWS.bend` minimal and precise; push helper lemmas into `PROOF.bend` .
- Run `bend PROOF.bend` before committing (per `AGENTS.md`).

F.7 · bend-runtime/knowledge/01-bend-language/match-recursion-arrays-parallelism.md

Match, Recursion, Arrays, Parallelism

Sources verified 2026-09-28:

- `bend guide` / `guide/GUIDE.md` (Recursion and Termination, Parallelism, Arrays) on https://github.com/bendlang/bend (https://github.com/bendlang/bend)
- `bend base` / `bend2/base.bend` (`List`, `Array.get/set/swap/clone`, `Nat`, `U32`)
- Repo README parallel example (`pow2`, `!` GPU marker) and Limitations (balanced calls, `!` semantics)

1. `match`: THE ONLY BRANCHING FORM

Bend has no `if`. `match` inspects a parameter or a variable bound by a pattern — never a computed value.

```
import Base

def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}:
      acc
    case Con{h, t}:
      sum(t, (acc + h : U32))

def main() -> U32:
  sum([1, 2, 3, 4], 0)
```

Match rules from the guide:

- Scrutinees follow binder order; a `let` may not precede a `match` on a parameter.
- `match sum(xs, 0):` is rejected. To match on a computed value, pass it to a helper that matches on its parameter.
- Patterns nest; `_` catches the rest; `1n+p` matches `Succ`.
- Multi-scrutinee matches read left to right (important for termination).

Correct vs rejected:

```
import Base

# OK: scrutinee is a parameter
def is_nil(a, -A: Kind(a), xs: List<a, A>) -> Bool:
  match xs:
    case Nil{}:
      True{}
    case Con{h, t}:
      False{}

# OK: scrutinee is a pattern-bound variable
def head_is_zero(xs: List<U32>) -> Bool:
  match xs:
    case Nil{}:
      False{}
    case Con{h, t}:
      U32.is_zero(h)

# REJECTED: computed scrutinee — split it manually
# def bad(xs: List<U32>) -> Bool:
#   match sum(xs, 0):
#     case 0:
#       True{}
#     case _:
#       False{}

# FIX: helper takes the computed value as a parameter
def is_zero_u32(x: U32) -> Bool:
  match x:
    case 0:
      True{}
    case _:
      False{}

def good(xs: List<U32>) -> Bool:
  is_zero_u32(sum(xs, 0))
```

Matching numbers and bools:

```
import Base

def pred(n: Nat) -> Nat:
  match n:
    case 0n:
      0n
    case 1n+p:
      p

def neg(b: Bool) -> Bool:
  match b:
    case False{}:
      True{}
    case True{}:
      False{}

# if-then-else replacement
def max_u32(+a: U32, +b: U32) -> U32:
  match U32.is_lt(a, b):
    case True{}:
      b
    case False{}:
      a
```

2. RECURSION AND TERMINATION

Bend uses recursion to repeat work. Tail calls compile to loops.

Termination rules (guide):

- Recursive calls must use smaller parts of inputs, obtained via pattern matching.
- Check reads arguments left to right: each must be passed unchanged until one is a smaller part of its parameter; ones after it are free. Put the shrinking parameter first.
- Termination is mandatory; mutual recursion is not allowed (keeps proofs sound).
- A server loop counts down a `Nat` fuel argument instead.

- Two mutually recursive functions become one def with a selector argument.
- `@unsafe / def f?(..)` turns the check off but voids proof guarantees.

```
import Base

def length(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{h, t}:
      1n+length(a, A, t)

def add(a: Nat, b: Nat) -> Nat:
  match a:
    case 0n:
      b
    case 1n+p:
      1n+add(p, b)
```

Fuel pattern for otherwise non-structural loops:

```
import Base

def loop_with_fuel(fuel: Nat, state: U32) -> U32:
  match fuel:
    case 0n:
      state
    case 1n+f:
      loop_with_fuel(f, (state + 1 : U32))

# Base uses the same idea for merge sort:
# def List.merge.go(~A, ~le, fuel: Nat, st: ...) -> ...:
#   match fuel:
#     case 0n: ...
#     case 1n+f: ... List.merge.go(~A, ~le, f, ...)
```

Unsafe escape hatch:

```
import Base

@unsafe def forever(x: U32) -> U32:
  forever((x + 1 : U32))

def also_unsafe?(x: U32) -> U32:
  also_unsafe?((x + 1 : U32))
```

`bend` runs unsafe code, but checking prints `SOME PROOFS FAIL` and names every def relying on it.

3. ARRAYS: PURE IN-PLACE MUTATION

```
import Base

def main() -> Array<U32> & U32:
  a = [0 : U32*8n] # new array with 8 copies of 0
  a[5] <- 42       # performs an in-place rewrite
  a[5]             # reads index 5
```

Array rules from the guide:

- `Array<T>` is a `Type` : exactly one owner at all times. That is what makes `a[5] <- 42` overwrite in place with no copy.
- A read hands the array back next to the element (`Array<U32> & U32`).
- Indexes wrap around.
- A write followed by another statement re-binds its array: `a[5] <- 42` on its own line is `a = a[5] <- 42` . As the last statement it is the written array.
- Slot count after `*` is a power of two; `[0 : U32^3n]` names depth instead (2^3 = 8 slots).
- `a[i]` sugar assumes `Array<U32>` . For other element types call `Array.get` (`Data` elements; else `Array.swap`) and `Array.set` directly, plus `Array.clone` when two copies are needed.

More examples:

```
import Base

def fill() -> Array<U32>:
  a = [0 : U32*8n]
  a[0] <- 10
  a[1] <- 20
  a[2] <- 30
  a  # last statement: the written array

def sum_array(a: Array<U32>) -> Array<U32> & U32:
  (a1, x0) = a[0]
  (a2, x1) = a1[1]
  (a2, (x0 + x1 : U32))

def generic_get(a: Array<String>, i: U32) -> Array<String> & String:
  Array.get(String, a, i)

def clone_demo(a: Array<U32>) -> Array<U32> & Array<U32>:
  Array.clone(U32, a)
```

Depth vs size:

```
import Base

def sizes() -> Array<U32> & Array<U32>:
  a = [0 : U32*8n]  # 8 slots
  b = [0 : U32^3n]  # 2^3 = 8 slots
  (a, b)
```

> Values are affine: closures and arrays cannot be shared. Clone explicitly.

4. PARALLELISM: = FORK-JOIN AND ! GPU

```
import Base

def pow2(+n: Nat) -> U32:
  match n:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p) # parallel call
      (a + b : U32)

def main() -> IO(Unit):
  IO.print(U32.show(pow2!(20n))) # `!` runs on GPU
```

Parallel-call contract:

1. Calls are independent (purity + affinity guarantee this).
2. They run in roughly the same time (yours to keep).

The scheduler is a contention-free binary fork-join machine: each task is handed to a core exactly once, never migrated. Fast and GPU-friendly, but the workload must stay balanced.

`!` semantics:

- Native binary: `pow2(20n)` runs in parallel on CPU; `pow2!(20n)` on GPU.
- Heap is unified; on unified-memory chips (Apple M-series) CPU→GPU is zero-cost.
- GPU shines on uniform numeric work (mandelbrot, nbody); divergent work (n-queens) stays faster on CPU.
- No GPU present: `!` still runs in parallel on CPU.
- `+` values read by every lane cost an atomic per read — read `bend guide shaders`.
- JS target ignores parallelism and runs sequentially.

Balanced vs unbalanced:

```
import Base

# GOOD: balanced tree, one leaf per unit, d levels
def pow2_bal(+d: Nat) -> U32:
  match d:
    case 0n:
      1
    case 1n+p:
      a b = pow2_bal(p) pow2_bal(p)
      (a + b : U32)

# BAD: unbalanced — one side is trivial, speedup is sub-ideal
def pow2_unbal(+d: Nat) -> U32:
  match d:
    case 0n:
      1
    case 1n+p:
      a b = pow2_unbal(p) 1
      (a + b : U32)

# Parallel map pattern (divide index range, join results)
def par_sum(a: Array<U32>, lo: U32, hi: U32) -> U32:
  match U32.is_eq(lo, hi):
    case True{}:
      fst = Array.get(U32, a, lo)
      Pair.snd(Array<U32>, U32, fst)
    case False{}:
      mid = U32.div((lo + hi : U32), 2)
      # In real code split the range and join; keep halves equal-sized
      x y = par_sum(a, lo, mid) par_sum(a, (mid + 1 : U32), hi)
      (x + y : U32)
```

Current limits (see `02-guides-cli/cli-tooling-limits.md`):

- Parallelism requires balanced calls; flexible parallelism is planned.
- Atomic arrays shared across threads are experimental, need `@unsafe`.
- One GPU per program; one event loop; no multi-machine execution yet.

5. PUTTING IT TOGETHER

```
import Base

def build(n: Nat) -> Array<U32>:
  match n:
    case 0n:
      [0 : U32*8n]
    case 1n+p:
      a = build(p)
      a[0] <- 99
      a

def par_init(depth: Nat) -> U32:
  match depth:
    case 0n:
      1
    case 1n+d:
      x y = par_init(d) par_init(d)
      (x + y : U32)

def main() -> IO(Unit):
  do IO<Unit>:
    result = par_init!(8n)
    IO.print(U32.show(result))
```

F.8 · bend-runtime/knowledge/01-bend-language/quantities-kinds-templates.md

Quantities, Kinds, and Templates

Sources verified 2026-09-28:

- `bend guide / guide/GUIDE.md` (Quantities, Kinds, Templates) on https://github.com/bendlang/bend (https://github.com/bendlang/bend)
- `bend base / bend2/base.bend` (`List<a,-A:Kind(a)>`, `Maybe`, `Word`, quantity-polymorphic defs)
- Repo README limitation notes (affine values, `+` costs refcount, no type classes/traits)

1. WHY QUANTITIES EXIST

Bend is affine by default: a variable may be used at most once. This lets the compiler free memory on `match` without a garbage collector, and lets `Array` mutation stay pure (single owner). Quantities opt specific bindings back into erasure (`-`) or reuse (`+`).

```
# -A: erased (gone at runtime)
#  n: affine (used at most once)
# +x: reusable (requires A to be Data)
def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n:
      Nil{}
    case 1n+p:
      x <> replicate(A, p, x)

def main() -> List<U32>:
  replicate(U32, 3n, 7)
```

Key rules from the guide:

- Erased variables appear only in types and proofs; the compiler deletes them.
- Affine is the default; dropping an affine variable is always free.
- Reusable (`+`) requires `Data`. Functions, arrays, and IO handles are `Type` and can never be copied.
- `main` marks nothing by default; write `+` where copies are needed, and the callee pays with a runtime reference count.
- Matching a `+` value hands out `+` fields; on a plain value, write `+r` in the pattern to make a field reusable.

2. QUANTITY ANNOTATIONS IN PRACTICE

```
import Base

# affine (default): used once
def use_once(x: U32) -> U32:
  (x + 1 : U32)

# reusable: may be used many times (U32 is Data)
def use_twice(+x: U32) -> U32:
  (x + x : U32)

# erased: type-level only, e.g. the element type tag
def length_erased(-A: Data, xs: List<A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{h, t}:
      1n+length_erased(A, t)

def main() -> Nat:
  length_erased(U32, [1, 2, 3])
```

More binding forms:

```
import Base

def lets() -> U32:
  x = 40        # affine let
  +y = 1        # reusable let
  -t = U32      # erased let (types only)
  (x + y : U32)

def lambda_capture() -> U32:
  +k = 10
  f = +kk => (kk + k : U32)  # reusable lambda variable if needed
  f(5)
```

Pattern-side `+`:

```
import Base

type Shape is Data:
  Circle{r: U32}
  Square{s: U32}

def area_twice(shape: Shape) -> U32:
  match shape:
    case Circle{+r}:
      (r * r : U32)
    case Square{+s}:
      (s + s : U32)
```

3. KINDS: `TYPE`, `DATA`, `KIND(A)`

Every type has a kind capping how often its values may be used:

- `Type = Kind(&1)` (affine, single owner)
- `Data = Kind(&2)` (reusable)
- `Kind(a)` accepts both when `a` is a quantity variable

```
import Base

#  a: a quantity (&0, &1 or &2)
# -A: a type whose values may be used a times
def length(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{h, t}:
      1n+length(a, A, t)

def main() -> Nat:
  length(&2, U32, [1, 2, 3])
```

What the guide states:

- `type List<a, -A: Kind(a)> is Kind(a)`: a list is exactly as reusable as its elements. `List<U32> = List<&1,U32>`; `+List<U32> = List<&2,U32>`.
- Short form needs the type declared above it; a type named before its declaration spells every parameter, quantities included.
- A type holding two element types combines quantities with `a <&> b` (the smaller of the two). See `Either`, `Result`, `Sigma` in Base.
- A bare `a` in a parameter list is short for `-a: Quant`.

Base examples to study (`bend base`):

```
type Either<a, b, -A: Kind(a), -B: Kind(b)> is Kind(a <&> b):
  Inl{value: A}
  Inr{value: B}

type List<a, -A: Kind(a)> is Kind(a):
  Nil{}
  Con{head: A, tail: List<a, A>}
```

Usage with closures (which are `Type`, hence `&1`):

```
import Base

def count_closures(xs: List< closures >) -> Nat:
  length(&1, U32 -> U32, xs)

def main2() -> Nat:
  length(&1, U32 -> U32, [(x => (x + 1 : U32))])
```

And with reusable data:

```
import Base

def sum_data(xs: List<&2, U32>) -> U32:
  match xs:
    case Nil{}:
      0
    case h <> t:
      (h + sum_data(t) : U32)
```

4. TEMPLATES (~): COMPILE-TIME INLINING

A template receives its argument as syntax and inlines it at compile time:

```
import Base

# ~f: substituted at compile time, not passed at runtime
def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

Template rules:

- Template parameters come first in the parameter list.
- A `~` argument must be closed: it may mention top-level defs but no local variable of the caller.
- Each distinct set of `~` arguments compiles to its own copy, so `f` costs nothing at runtime and — unlike a closure — may be called many times.
- A template may call only templates declared above it.
- This is how `List.map` is written in Base.

Base's `List.map` (from `bend2/base.bend`):

```
def List.map(~A: Type, ~B: Type, ~f: A -> B, xs: List<A>) -> List<B>:
  match xs:
    case Nil{}:
      Nil{}
    case h <> t:
      f(h) <> List.map(~A, ~B, ~f, t)
```

More template examples:

```
import Base

def apply_thrice(~f: U32 -> U32, x: U32) -> U32:
  f(f(f(x)))

def inc(x: U32) -> U32:
  (x + 1 : U32)

def main2() -> U32:
  apply_thrice(~inc, 0)

# Higher-order helpers must be templates to reuse the function argument:
def List_filter_demo() -> List<U32>:
  List.filter(~U32, ~U32.is_even, [1, 2, 3, 4])
```

Template vs closure cheat sheet:

Feature	Closure (`x => e`)	Template (`~f`)
Runtime cost	allocates, affine (call once)	inlined, zero cost
Reuse	at most once	unlimited
Captures locals	yes	no (must be closed)
Call target	any value	only templates above it

```
import Base

def adder(k: U32) -> U32 -> U32:
  x => (x + k : U32)

def main3() -> U32:
  add2 = adder(2)
  # add2 is affine: calling it consumes it
  add2(40)
```

5. COMMON PITFALLS

1. Forgetting `+` on a twice-used variable. Fix: mark the binder `+x` or the pattern field `+r` — but only if the type is `Data`.
2. Trying to copy a `Type` (closure, `Array`, `IO`, handle). Impossible by design; restructure to thread it linearly or make it a template.
3. Expecting inference: spell `Kind(a)` params and quantity args (`&1` / `&2`).
4. Using a local inside `~arg`. Hoist it to a top-level `def` first.
5. Assuming `+` is free: it adds a reference count. Prefer affine threading in hot loops.

6. MINI EXERCISES

```
import Base

# 1. Make this compile: use x three times (U32 is Data, so + is legal)
def triple(+x: U32) -> U32:
  (x + x + x : U32)

# 2. Quantity-polymorphic length already shown; try counting Maybe values
def count_maybe(a, -A: Kind(a), m: Maybe<a, A>) -> Nat:
  match m:
    case None{}:
      0n
    case Some{x}:
      1n

# 3. Write a template that maps then folds (composes without closures)
def map_sum(~f: U32 -> U32, xs: List<U32>) -> U32:
  match xs:
    case Nil{}:
      0
    case h <> t:
      (f(h) + map_sum(~f, t) : U32)
```

F.9 · `bend-runtime/knowledge/01-bend-language/syntax-reference.md`

Bend2 Syntax Reference

Sources verified 2026-09-28:

- `bend guide` / `guide/GUIDE.md` on https://github.com/bendlang/bend (`https://github.com/bendlang/bend`) (`guide/` directory, syntax-reference section)
- `bend base` / `bend2/base.bend` on https://github.com/bendlang/bend (`https://github.com/bendlang/bend`)
- Repo README examples (hello world, `Shape`, operators, literals)

Bend2 syntax is Python-shaped with Haskell/Lean semantics and Rust-like resource awareness. This file is the canonical local copy of the "Syntax Reference" section of the guide. See also `raw/GUIDE-CLI-DUMP.md` (untouched dump) for the verbatim upstream text.

1. PROGRAM SKELETON

Every file starts with imports, then types, defs, and laws:

```
import Base
import ./math.bend as M

type Shape is Data:
  Circle{r: U32}
  Square{s: U32}

def area(x: Shape) -> U32:
  match x:
    case Circle{+r}:
      (3 * r * r : U32)
    case Square{+s}:
      (s * s : U32)

def main() -> U32:
  area(Square{5})
```

Rules:

- `import Base` is the prelude. Almost every example starts with it.
- `import ./file.bend as M` gives a local alias; `M.x` names every def.
- Dots inside a name are just characters: `U32.show` needs no module.
- A module path is plain names (`letters, digits, _ and -`); `math.extra.bend` is refused.

2. TOP LEVEL FORMS

```
import Base                              # the prelude
import ./file.bend as M                  # a module; its defs are M.x
type D<a, -A: Kind(a)> is Kind(a):       # a datatype and its kind
  K{x: A, xs: List<a, A>}                # one constructor per line
def f(x: A, -y: B, +z: C) -> T:          # a def; the body follows
def f(x, y):                             # fills the law named f
def t(~g: A -> B, x: A) -> B:            # a template
law f:                                   # a claim, proven by def f
  for x: A                               # a parameter (also for -x, for +x)
  for y: B where P(y)                    # y is then the pair (y, P(y) proof)
  exs z: C                               # a witness the proof must return
  T                                      # the claim
@unsafe def f(x: A) -> T:                # skips the termination check
def f?(x: A) -> T:                       # the same, as a sugar
def e(x: A) -> IO(B):                    # a foreign effect
  import "./e.c"
  import "./e.js"
```

Additional top-level notes from the guide:

- `@unsafe / def f?(..)` recurses freely and may call a def declared below it, but falls outside proof guarantees (`SOME PROOFS FAIL`).
- Types are not code: two datatypes, or a datatype and a type-level def, may name each other in any order. Only `def` order matters.
- A law left open in one file may be filled in another as `def M.name(..)` .

3. TYPES

```
Type  Data  Kind(q)                     # kinds; Type = Kind(&1), Data = Kind(&2)
Quant  &0  &1  &2  a <&> b             # quantities and their minimum
A -> B  @x:A -> B  @-x:A -> B          # functions: plain, dependent, erased
A & B  &x:A -> B  A | B                # pairs, dependent pairs, sums
D<A>  +D<A>  D<&2, A>                  # a datatype; + makes it reusable
{a == b : T}  {a != b : T}             # equality and its negation
```

Examples:

```
import Base

def Pair_example() -> U32 & String:
  (42, "answer")

def IsEven(n: Nat) -> Type:
  match n:
    case 0n:
      Unit
    case 1n+p:
      IsOdd(p)

def IsOdd(n: Nat) -> Type:
  match n:
    case 0n:
      Empty
    case 1n+p:
      IsEven(p)
```

Since types are terms, a `def` may return a `Type` (dependent types).

4. TERMS

```
42  1.5  3n  'c'  "s"                  # U32, F32, Nat, Char, String
[a, b]  h <> t  (a, b)                 # a list, a cons, a tuple
K{a, b}  x => e  +x => e               # a constructor, a lambda
f(a, b)  f!(a)  t(~g, a)               # a call, on the GPU, of a template
(a + b * c : T)  {x : T}               # operators over T; an annotation
[v : T*n]  [v : T^d]  a[i]  a[i] <- v  # an array of n or 2^d slots; a read, a write
{==}  %e : P; e2  %e@E : P; e2         # reflexivity, a rewrite, a named one
?name  ?TODO                           # print the goal; leave it open
```

Literals:

```
import Base

def literals() -> String:
  n : Nat = 3n
  u : U32 = 42
  f : F32 = 1.5
  c : Char = 'c'
  s : String = "s"
  U32.show(u) ++ " / " ++ Nat.show(n)
```

- A `Nat` literal past `256n` is `U32.to_nat(n)` underneath, up to `4294967295n`.
- `42` without annotation is `U32`; `3n` is `Nat`; `1.5` is `F32`.

5. OPERATORS

Inside `(.. : T)`, operators desugar to `T.*` defs in Base:

- `+ - * / %` call `T.add` through `T.mod`
- `.&. .|. .^.` call bit ops, `<< >>` shifts (by a `Nat`)
- `< <= > >=` call the `T.is_lt` family
- Without `: T` they are refused; annotate as `{3 : U32}` when checker is stuck
- `&& ||` work on `Bool` and `++` on `String` anywhere
- Operators need spaces on both sides
- Equality of values is a call `T.is_eq(a, b)`; `==` is only the type

```
import Base

def ops(+a: U32, +b: U32) -> U32:
  +c = (a + b * 2 : U32)
  +d = (c .&. 255 : U32)
  (c + d : U32)

def bools(a: Bool, b: Bool) -> Bool:
  a && b || True{}

def greet(a: String, b: String) -> String:
  a ++ ", " ++ b
```

6. STATEMENTS

```
x = v  +x = v  -x = v                  # a let: affine, reusable, erased
(a, b) = v  K{x, y} = v                # a destructuring let
a b = f(x) g(y)                        # a parallel let
match a b:                             # a match on one or more values
  case K{x, _} 1n+p:                   # patterns nest; _ catches the rest
do M<xs.., R>:                         # a monadic block over M.bind, M.pure
  x : A <- m                           # bind
  x : A = v                            # let
  m                                    # a Unit step
  return v                             # the result
```

Example combining lets, destructuring, and annotation:

```
import Base

def area2(shape: Shape) -> U32:
  match shape:
    case Circle{r}:
      +rr = r
      (3 * rr * rr : U32)
    case Square{s}:
      (s * s : U32)

def swap_example(p: U32 & U32) -> U32 & U32:
  (a, b) = p
  (b, a)

def annotated() -> U32:
  {3 : U32}
```

7. NO IF SYNTAX

There is no `if-then-else`. Branch on `True{} / False{}`:

```
import Base

def max_u32(+a: U32, +b: U32) -> U32:
  match U32.is_lt(a, b):
    case False{}:
      a
    case True{}:
      b
```

8. CHECKER ADVICE

- Bend does almost no inference. When the checker cannot decide a type, annotate.
- `{x : T}` is the universal escape hatch.
- `?name` prints the goal; `?TODO` leaves it open (useful while proving).
- A failed proof step prints expected vs observed terms.

9. HELLO-WORLD REMINDER

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

Run with `bend hello.bend`; compile with `bend hello.bend -o hello`. See `02-guides-cli/cli-tooling-limits.md` for the full tooling table.

F.10 · bend-runtime/knowledge/02-guides-cli/cli-tooling-limits.md

Bend2 CLI, Tooling, and Limits

Sources verified 2026-09-28:

- https://github.com/bendlang/bend (`https://github.com/bendlang/bend`) (README Get Started, References, Limitations, Community)
- `bend guide / guide/GUIDE.md` (Tooling section) and `bend --help`
- `bend base / bend2/base.bend`, `bend guide shaders`, `bend guide effects`
- Local untouched dumps: `02-guides-cli/raw/` (`GUIDE-CLI-DUMP.md`, `BASE-*-DUMP.md`, `CLI-HELP-DUMP.txt`, `EFFECTS-CLI-DUMP.md`, `SHADERS-CLI-DUMP.md`)

1. INSTALL AND AGENT SETUP

```
curl -fsSL https://bend-lang.com/install.sh | sh
```

Add to `AGENTS.md` (per repo README / `AGENTS.md`):

```
When using Bend:
- run `bend guide` to learn it
- use `LAWS.bend` to keep important rules
- run `bend PROOF.bend` before committing
- parallelize the code whenever possible
```

Then tell the agent: "use Bend". Works best on back-end, Linux or macOS. This `bend` ignores a project's `bunfig.toml` and `.env`; `bun bend2/main.ts` from a checkout reads them, so check untrusted code with `bend`.

2. THE SINGLE COMMAND

```
bend file.bend            # check; run main (IO compiled; a value normalized)
bend file.bend -o file    # compile to a native binary (clang 14+; 19+ with `!`)
bend file.bend -o file.c  # emit the C source instead
bend file.bend -o file.js # emit the JS source instead
bend file.bend -o f.mjs   # emit an ES module of its non-IO defs, for JS to import
bend file.bend --verdict  # check; then recheck with the proven BendTT kernel
bend page.html -o dist    # bundle a web page that imports .bend files
./file --threads 8        # run a native binary on 8 CPU threads
./file --gpu off          # run ! calls on the CPU (the GPU is on by default)
./file --gpu 4GB          # cap the GPU's heap at 4GB
```

Semantics:

- `main: IO` runs compiled; `main: value` is normalized by the checker (slow for big work) and printed; no `main` just checks.
- Binary using `!` also builds `file.gpu` (must stay beside it): macOS needs Metal, Linux needs CUDA 12 at `/usr/local/cuda`.
- Linux Window needs `libx11-dev`; Audio needs `libasound2-dev`.
- `bend guide` prints the guide; `bend base` prints Base (`bend base Map` one name + everything under it); `bend --help` lists rest.

Learn/print commands:

```
bend guide                # full GUIDE.md
bend guide shaders        # "Shaders in Bend" (AI-written tutorial; read before parallel/graphical apps)
bend guide effects        # "Effects in Bend" (AI-written note on C/JS side; read before custom effects)
bend base                 # whole base.bend
bend base --types         # types only
bend base Map             # one namespace
bend --help               # all commands
```

Publishing / hub:

```
bend main.bend --publish                 # upload file + imports; prints import-by-hash line
bend main.bend --publish <name>@<version> # named publish (after `bend login`)
bend login                               # authenticate for named publishes
```

Example session:

```
bend hello.bend
bend hello.bend -o hello
./hello --threads 8
bend PROOF.bend
bend PROOF.bend --verdict
bend PROOF.bend -o PROOF.bendtt
bend base U32 | head -n 50
```

Minimal programs for each mode:

```
# value mode: normalized + printed
import Base
def main() -> U32:
  (6 * 7 : U32)
```

```
# IO mode: compiled + executed
import Base
def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

```
# GPU mode: ! hands call + inner parallel calls to GPU
import Base
def pow2(+d: Nat) -> U32:
  match d:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)
def main() -> IO(Unit):
  result = pow2!(20n)
  IO.print(U32.show(result))
```

3. JS / WEB TARGETS

- `-o file.js` : JS source (single core, no graphics/audio, parallelism ignored, runs sequentially).
- `-o f.mjs` : ES module of non-IO defs for JS import:

```
import Game from "./game.mjs";
// constructors as {$: "Name", field: value}, Nat as BigInt
```

- `bend page.html -o dist` : bundle a web page importing `.bend` files.
- From Bend source directly (dev): JS file may `import Game from "./game.bend"` with `bend2/main.ts` preloaded.

4. LIMITS (UPSTREAM `LIMITATIONS`, VERBATIM INTENT)

Bend 2 is new; Bend 1 programs and HVM do not carry over.

- Verbose: everything annotated, nothing inferred.
- No type classes, traits; no macros beyond compile-time templates (~).
- No tactics or proof search; proving takes extra effort.
- Affine values: closures and arrays cannot be shared.
- Recursion must terminate; `@unsafe` / `def f?(..)` opts out (voids proofs).
- No computed matches (`match f(x)`); split manually via helper.
- No `if-then-else` ; `match` on `True{}` / `False{}` .
- Numbers only `Nat` , `U32` , `F32` : no `U64` / `I64` / `F64` (Metal has no f64).
- `F32` axiomatic: nothing about floats provable.
- Strings are char linked lists: text processing slow.
- Small Base: expect to write helpers.
- Few effects: print, env, time, sleep, spawn, channels, files, TCP, UDP. No TLS/HTTP lib/JSON/regex yet (add as foreigns).
- Targets: C, Metal, CUDA, JavaScript; Lua/Luau/Python planned.
- JS target: one core, no graphics/audio.
- Parallelism needs balanced calls; flexible parallelism planned.
- Atomic arrays shared across threads experimental, need `@unsafe` .
- One GPU/program, one event loop, no multi-machine yet.
- One C file/program: no separate/incremental compilation.
- Native compile slow (clang/CUDA/Metal); use JS for fast dev iteration.

- Young compiler blind spots (unusually slow programs) — report.
- Few benchmarks (esp. checker).
- Compiler (not kernel) 99% AI-written, unaudited.
- Checker unproven, may have bugs; `--verdict` uses proven kernel.
- Needs clang 14+; `!` needs 19+, Metal or CUDA 12.
- No Windows (WSL works); Linux Window/Audio need X11/ALSA headers.
- Hub: no names/versions/accounts/search yet; packages are hashes.
- Terse errors; no debugger/profiler/REPL.
- Editor tool formats only; community `bend2-lsp` adds errors/hover.
- No test framework; guide is the documentation.

> Upstream notes most limits are being addressed. Be patient, report bugs, request features. See https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues).

5. PERFORMANCE AND CORRECTNESS WORKFLOW

```
# fast dev loop (JS, sequential)
bend app.bend -o app.js

# native CPU parallel
bend app.bend -o app && ./app --threads 16

# GPU path
bend app.bend -o app && ./app --gpu 4GB
./app --gpu off   # force CPU fallback for comparison

# proofs gate before commit
bend PROOF.bend
bend PROOF.bend --verdict
```

- Under the hood: one emitted C file is both CPU program and GPU kernel (clang vs Metal/CUDA compile it). Term = 64-bit word (inline small values, else pointer into single heap shared by every core + GPU). No GC: `match` frees opened node; only `+` carries refcount. No C stack: each def is a segment of a flat state machine; call = jump; parallel call = join task + one task per call across CPU/GPU lanes. See `paper/BendRT.pdf`.
- Theory: one universe, no positivity check; live/dead wall keeps consistency; `bend2/bendtt.lean` kernel + proof; `--verdict` checks with it. See `paper/BendTT.pdf`.

6. REFERENCES AND COMMUNITY

- Guide: `GUIDE.md` in repo (`guide/`), also `bend guide`.
- Demos: `demos/` (apps, servers, proofs, each with `LAWS.bend`).
- Base: `base.bend` (`bend2/`), also `bend base`.
- Papers: `paper/BendTT.pdf`, `paper/BendRT.pdf`.
- Formalization: `bend2/bendtt.lean` (kernel of `--verdict` + proofs, in Lean).
- Benches: `bench/`.
- Formatter: `tools/bend-fmt-lsp`; community server `bend2-lsp` (format, diagnostics, hover).
- Website: https://bend-lang.com (https://bend-lang.com)
- Discord: https://discord.bend-lang.com (https://discord.bend-lang.com)
- X: https://x.com/bendlang (https://x.com/bendlang)
- Reddit: https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/)
- Issues: https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues)

7. CREDITS

Bend created by Victor Taelin; team: Lorenzo W Battistela, Paulo J Cavalcanti, Nico Abril, Vanessa Ostroski, Vitor Chiarelli Neves, Alex Van de Sande (per README; ping upstream if missing). Thanks to Ayush Somani for reserving the `bend-lang` name.

F.11 · bend-runtime/knowledge/02-guides-cli/guides-index.md

Guides index (what `bend guides` can do — bend 2.0.25)

> Note: CLI is `bend guide` (singular). `bend guides` errors (`no such file: guides`). Raw dumps: `raw/`.

Command	File	What it covers
`bend guide`	`raw/GUIDE-CLI-DUMP.md` (621 lines)	Full language: hello, types, closures, recursion, parallelism, arrays, quantities, kinds, templates, laws/proofs, IO/concurrency, monads, apps, Base, modules, tooling, syntax ref, under-the-hood, further reading
`bend guide shaders`	`raw/SHADERS-CLI-DUMP.md` (268 lines)	Perf tutorial from `demos/app_slash_boss_3d` (120 FPS pure Bend): fork tree + flat loops, 4^7 leaves, host-build/device-draw
`bend guide effects`	`raw/EFFECTS-CLI-DUMP.md` (129 lines)	C/JS side of custom effects
`bend base`	`raw/BASE-FULL-DUMP.md` (3006 lines)	Entire Base library
`bend base --types`	`raw/BASE-TYPES-DUMP.md`	Types only
`bend base <Name>`	—	One name + subtree (e.g. `bend base Map`, `bend base IO`)
`bend --help`	`raw/CLI-HELP-DUMP.txt`	CLI usage

Repo mirrors: https://github.com/bendlang/bend/tree/main/guide (https://github.com/bendlang/bend/tree/main/guide) (`GUIDE.md`, `SHADERS.md`, `EFFECTS.md`), https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend)

WHAT BEND CURRENTLY CAN DO (GUIDE CAPABILITIES)

- Python-shaped pure programs with Lean-like proofs + C/CUDA speed; `bend file.bend` check+run, `-o` native/C/JS, HTML bundling.
- Numbers `Nat/U32/F32`, strings, lists, maps/sets, arrays with in-place ops, images/quadtrees, events, audio.
- Effects: files, TCP/UDP, sockets, windows, audio, processes, time, random, env, channels, spawn/sleep. No TLS/HTTP-lib/JSON/regex (use foreigns).
- GPU via `!` (Metal macOS, CUDA 12 Linux `/usr/local/cuda`), `--threads`, `--gpu off/4GB`.
- Proofs gate releases; hub publish by hash or `name@ver`.

F.12 · bend-runtime/knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md

Types

=====

Data

---

type Empty is Data:

type Unit is Data: Unit{}

```
type Bool is Data: False{} True{}

type Cmp is Data: LT{} EQ{} GT{}

type Either<a, b, -A: Kind(a), -B: Kind(b)> is Kind(a <&> b): Inl{value: A} Inr{value: B}

type Sigma<a, b, -A: Kind(a), -B: @-x: A -> Kind(b)> is Kind(a <&> b): Tuple{fst: A, snd: B(fst)}

type Nat is Data: Zero{} Succ{pred: Nat}

type Maybe<a, -A: Kind(a)> is Kind(a): None{} Some{value: A}

type Result<a, b, -E: Kind(a), -A: Kind(b)> is Kind(a <&> b): Fail{error: E} Done{value: A}

type List<a, -A: Kind(a)> is Kind(a): Nil{} Con{head: A, tail: List<a, A>}

law Word: for n: Nat Data

type Word.Nil is Data: WNil{}

type Word.Con<-p: Nat> is Data: WCon{head: Bool, tail: Word(p)}

type U32 is Data: U32{data: Word(32n)}

type F32 is Data: F32{data: Word(32n)}

type Char is Data: Chr{code: U32}

type String is Data: SNil{} SCon{head: Char, tail: String}

type Array<-T: Type> is Type: ALeaf{value: T} ANode{xs: Array<T>, ys: Array<T>}

type Image is Data: Pix{color: U32} Qua{tl: Image, tr: Image, bl: Image, br: Image}

type Event is Data: Key{code: U32, down: Bool} Mouse{x: U32, y: U32, button: U32, down: Bool} Move{x: U32, y: U32}
Close{}

type Map<a, -V: Kind(a)> is Kind(a): MTip{} MLeaf{key: String, val: V} MNode{pos: Nat, lo: Map<a, V>, hi: Map<a, V>}
```

Effects

A handle is opaque, a law with no constructor and no field, so no

program forges one, and linear (Type), so none copies or reuses one.

```
law File: Type

law Socket: Type

law Listener: Type

law Window: Type

law Audio: Type
```

A channel is Data: the computations it links each hold a copy.

```
law Chan: for -A: Type Data

type IO.OP<-R: Type> is Type: Emit{value: R} Halt{code: U32, message: String}

law Pair: for -A: Type for -B: Type Type

law IO: for -A: Type Type

type App<-S: Type> is Type: App{view: S -> Pair(S, Image), tick: List<Event> -> S -> IO(Maybe<S>)}
```

Defs

====

Word

```
def Word(n): match n: case 0n: Word.Nil case 1n+p: Word.Con
```

Pair

```
def Pair(A, B): Sigma<&1, &1, A, _ => B>

def Exists(-A: Type, -B: @-x: A -> Type) -> Type: Sigma<&1, &1, A, B>

def Or(-A: Type, -B: Type) -> Type: Either<&1, &1, A, B>

def Pair.fst(-A: Type, -B: Type, p: A & B) -> A: (a, b) = p a

def Pair.snd(-A: Type, -B: Type, p: A & B) -> B: (a, b) = p b
```

Effects

```
def IO(A): @-R: Type -> @k: (A -> IO.OP<R>) -> IO.OP<R>

def IO.pure(-A: Type, x: A) -> IO(A): R => k => k(x)

def IO.bind(-A: Type, -B: Type, m: IO(A), f: A -> IO(B)) -> IO(B): R => k => m(R, x => f(x, R, k))

def IO.print(text: String) -> IO(Unit): import "./effs/print.c" import "./effs/print.js"

def IO.write(text: String) -> IO(Unit): import "./effs/write.c" import "./effs/write.js"

def IO.print_err(text: String) -> IO(Unit): import "./effs/print_err.c" import "./effs/print_err.js"

def IO.get_env(name: String) -> IO(Result<&1, &1, U32 & String, String>): import "./effs/get_env.c" import "./effs/get_env.js"

def IO.args() -> IO(List<String>): import "./effs/args.c" import "./effs/args.js"

def IO.die(-A: Type, code: U32, msg: String) -> IO(A): R => k => Halt{code, msg}

def IO.pass(-A: Type, r: Result<&1, &1, U32 & String, A>) -> IO(A): match r: case Done{value}: IO.pure(A, value) case Fail{(code, message)}: IO.die(A, code, message)

def IO.try(-A: Type, act: IO(Result<&1, &1, U32 & String, A>)) -> IO(A): IO.bind(Result<&1, &1, U32 & String, A>, A, act, IO.pass(A))

def IO.random_u32() -> IO(Result<&1, &1, U32 & String, U32>): import "./effs/random_u32.c" import "./effs/random_u32.js"

def IO.spawn(-A: Type, act: IO(A)) -> IO(Unit): import "./effs/spawn.c" import "./effs/spawn.js"

def IO.sleep(ms: U32) -> IO(Unit): import "./effs/sleep.c" import "./effs/sleep.js"

def IO.now() -> IO(Nat): import "./effs/now.c" import "./effs/now.js"

def Chan.new(-A: Type, room: U32) -> IO(Chan(A)): import "./effs/chan_new.c" import "./effs/chan_new.js"

def Chan.send(-A: Type, chan: Chan(A), value: A) -> IO(Bool): import "./effs/chan_send.c" import "./effs/chan_send.js"

def Chan.recv(-A: Type, chan: Chan(A)) -> IO(Maybe<&1, A>): import "./effs/chan_recv.c" import "./effs/chan_recv.js"

def Chan.close(-A: Type, chan: Chan(A)) -> IO(Unit): import "./effs/chan_close.c" import "./effs/chan_close.js"

def IO.fork.go(-A: Type, act: IO(A), chan: Chan(A)) -> IO(Chan(A)): +c = chan sent = IO.bind(A, Bool, act, x => Chan.send(A, c, x)) IO.bind(Unit, Chan(A), IO.spawn(Bool, sent), u => IO.pure(Chan(A), c))
```

```
def IO.fork(-A: Type, act: IO(A)) -> IO(Chan(A)): IO.bind(Chan(A), Chan(A), Chan.new(A, 1), IO.fork.go(A, act))

def IO.join.go(-A: Type, chan: Chan(A), got: Maybe<&1, A>) -> IO(A): match got: case None{}: IO.die(A, 1, "IO.join: the channel was closed") case Some{value}: IO.bind(Unit, A, Chan.close(A, chan), u => IO.pure(A, value))

def IO.join(-A: Type, +chan: Chan(A)) -> IO(A): IO.bind(Maybe<&1, A>, A, Chan.recv(A, chan), IO.join.go(A, chan))

def File.open(path: String, mode: String) -> IO(Result<&1, &1, U32 & String, File>): import "./effs/file_open.c" import "./effs/file_open.js"

def File.read(file: File, max: U32) -> IO(File & Result<&1, &1, U32 & String, String>): import "./effs/file_read.c" import "./effs/file_read.js"

def File.read_bytes(file: File, max: U32) -> IO(File & Result<&1, &1, U32 & String, List<&2, U32>>): import "./effs/file_read.c" import "./effs/file_read.js"

def File.read_at(file: File, offset: U32, max: U32) -> IO(File & Result<&1, &1, U32 & String, List<&2, U32>>): import "./effs/file_read.c" import "./effs/file_read.js"

def File.size(file: File) -> IO(File & Result<&1, &1, U32 & String, U32>): import "./effs/file_size.c" import "./effs/file_size.js"

def File.write(file: File, data: String) -> IO(File & Result<&1, &1, U32 & String, Unit>): import "./effs/file_write.c" import "./effs/file_write.js"

def File.write_bytes(file: File, data: List<&2, U32>) -> IO(File & Result<&1, &1, U32 & String, Unit>): import "./effs/file_write.c" import "./effs/file_write.js"

def File.close(file: File) -> IO(Unit): import "./effs/file_close.c" import "./effs/file_close.js"

def TCP.listen(port: U32) -> IO(Result<&1, &1, U32 & String, Listener>): import "./effs/tcp_listen.c" import "./effs/tcp_listen.js"

def TCP.accept(listener: Listener) -> IO(Listener & Result<&1, &1, U32 & String, Socket>): import "./effs/tcp_accept.c" import "./effs/tcp_accept.js"

def TCP.connect(host: String, port: U32) -> IO(Result<&1, &1, U32 & String, Socket>): import "./effs/tcp_connect.c" import "./effs/tcp_connect.js"

def TCP.send(sock: Socket, data: String) -> IO(Socket & Result<&1, &1, U32 & String, Unit>): import "./effs/tcp_send.c" import "./effs/tcp_send.js"

def TCP.recv(sock: Socket, max: U32) -> IO(Socket & Result<&1, &1, U32 & String, String>): import "./effs/tcp_recv.c" import "./effs/tcp_recv.js"

def TCP.poll(sock: Socket, max: U32, ms: U32) -> IO(Socket & Result<&1, &1, U32 & String, Maybe<&1, String>>): import "./effs/tcp_poll.c" import "./effs/tcp_poll.js"

def UDP.bind(port: U32) -> IO(Result<&1, &1, U32 & String, Socket>): import "./effs/udp_bind.c" import "./effs/udp_bind.js"

def UDP.send_to(sock: Socket, host: String, port: U32, data: String) -> IO(Socket & Result<&1, &1, U32 & String, Unit>): import "./effs/udp_send_to.c" import "./effs/udp_send_to.js"

def UDP.recv_from(sock: Socket, max: U32) -> IO(Socket & Result<&1, &1, U32 & String, String & U32 & String>): import "./effs/udp_recv_from.c" import "./effs/udp_recv_from.js"

def UDP.poll(sock: Socket, max: U32) -> IO(Socket & Result<&1, &1, U32 & String, Maybe<&1, String & U32 & String>>): import "./effs/udp_poll.c" import "./effs/udp_poll.js"

def Socket.close(socket: Socket) -> IO(Unit): import "./effs/socket_close.c" import "./effs/socket_close.js"

def Listener.close(listener: Listener) -> IO(Unit): import "./effs/listener_close.c" import "./effs/listener_close.js"

def Window.open(title: String, width: U32, height: U32) -> IO(Result<&1, &1, U32 & String, Window>): import "./effs/window_open.c" import "./effs/window_open.js"

def Window.frame(window: Window, image: Image) -> IO(Window & Image & List<Event>): import "./effs/window_frame.c" import "./effs/window_frame.js"

def Window.set_title(window: Window, title: String) -> IO(Window): import "./effs/window_set_title.c" import "./effs/window_set_title.js"

def Window.close(window: Window) -> IO(Unit): import "./effs/window_close.c" import "./effs/window_close.js"
```

```
def Audio.open(rate: U32) -> IO(Result<&1, &1, U32 & String, Audio>): import "./effs/audio.c" import "./effs/audio_open.js"

def Audio.write(audio: Audio, samples: List<&2, F32>) -> IO(Audio & U32): import "./effs/audio.c" import "./effs/audio_write.js"

def Audio.close(audio: Audio) -> IO(Unit): import "./effs/audio.c" import "./effs/audio_close.js"
```

Equal

```
---

law Equal.cong: for -A: Type for -B: Type for -f: A -> B for -a: A for -b: A for e: {a == b : A} {f(a) == f(b) : B}

def Equal.cong(A, B, f, a, b, e): %e : {f(a) == f(_) : B} {==}

law Equal.sym: for -A: Type for -a: A for -b: A for e: {a == b : A} {b == a : A}

def Equal.sym(A, a, b, e): %e : {_ == a : A} {==}

law Equal.trans: for -A : Type for -a : A for -b : A for -c : A for ab : {a == b : A} for bc : {b == c : A} {a == c : A}

def Equal.trans(A, a, b, c, ab, bc): %bc : {a == _ : A} ab
```

Empty

```
---

def Empty.absurd(-A: Type, e: Empty) -> A: match e:
```

Bool

```
---

def Bool.not(b: Bool) -> Bool: match b: case False{}: True{} case True{}: False{}

def Bool.and(a: Bool, b: Bool) -> Bool: match a: case False{}: False{} case True{}: b

def Bool.or(a: Bool, b: Bool) -> Bool: match a: case False{}: b case True{}: True{}

def Bool.xor(a: Bool, b: Bool) -> Bool: match a: case False{}: b case True{}: Bool.not(b)

def Bool.cmp(a: Bool, b: Bool) -> Cmp: match a b: case False{} False{}: EQ{} case False{} True{}: LT{} case True{} False{}: GT{} case True{} True{}: EQ{}

def Bool.full_add(a: Bool, b: Bool, c: Bool) -> Bool & Bool: match a b c: case False{} False{} False{}: (False{}, False{}) case False{} False{} True{}: (True{}, False{}) case False{} True{} False{}: (True{}, False{}) case False{} True{} True{}: (False{}, True{}) case True{} False{} False{}: (True{}, False{}) case True{} False{} True{}: (False{}, True{}) case True{} True{} False{}: (False{}, True{}) case True{} True{} True{}: (True{}, True{})

def Bool.pick(-A: Type, c: Bool, a: A, b: A) -> A: match c: case False{}: b case True{}: a

def Bool.to_u32(b: Bool) -> U32: match b: case False{}: 0 case True{}: 1
```

Cmp

```
---

def Cmp.is_lt(c: Cmp) -> Bool: match c: case LT{}: True{} case EQ{}: False{} case GT{}: False{}

def Cmp.is_eq(c: Cmp) -> Bool: match c: case LT{}: False{} case EQ{}: True{} case GT{}: False{}

def Cmp.is_gt(c: Cmp) -> Bool: match c: case LT{}: False{} case EQ{}: False{} case GT{}: True{}

def Cmp.is_le(c: Cmp) -> Bool: match c: case LT{}: True{} case EQ{}: True{} case GT{}: False{}

def Cmp.is_ge(c: Cmp) -> Bool: match c: case LT{}: False{} case EQ{}: True{} case GT{}: True{}
```

Nat

```
def Nat.double(n: Nat) -> Nat: match n: case 0n: 0n case 1n+p: 2n+Nat.double(p)

def Nat.add(a: Nat, b: Nat) -> Nat: match a: case 0n: b case 1n+p: 1n+Nat.add(p, b)

def Nat.sub(a: Nat, b: Nat) -> Nat: match a b: case 0n 0n: 0n case 0n 1n+bp: 0n case 1n+ap 0n: 1n+ap case 1n+ap 1n+bp: Nat.sub(ap, bp)

def Nat.mul(a: Nat, +b: Nat) -> Nat: match a: case 0n: 0n case 1n+p: Nat.add(b, Nat.mul(p, b))

def Nat.divmod.go(n: Nat, m: Nat, d: Nat, r: Nat) -> Nat & Nat: match n: case 0n: (d, r) case 1n+np: match m: case 0n: Nat.divmod.go(np, r, 1n+d, 0n) case 1n+mp: Nat.divmod.go(np, mp, d, 1n+r)

def Nat.divmod(a: Nat, b: Nat) -> Nat & Nat: match b: case 0n: (0n, a) case 1n+bp: Nat.divmod.go(a, bp, 0n, 0n)

def Nat.cmp(a: Nat, b: Nat) -> Cmp: match a b: case 0n 0n: EQ{} case 0n 1n+bp: LT{} case 1n+ap 0n: GT{} case 1n+ap 1n+bp: Nat.cmp(ap, bp)

def Nat.is_lt(a: Nat, b: Nat) -> Bool: Cmp.is_lt(Nat.cmp(a, b))

def Nat.is_eq(a: Nat, b: Nat) -> Bool: Cmp.is_eq(Nat.cmp(a, b))

def Nat.is_ne(a: Nat, b: Nat) -> Bool: Bool.not(Cmp.is_eq(Nat.cmp(a, b)))

def Nat.is_le(a: Nat, b: Nat) -> Bool: Cmp.is_le(Nat.cmp(a, b))

def Nat.is_gt(a: Nat, b: Nat) -> Bool: Cmp.is_gt(Nat.cmp(a, b))

def Nat.is_ge(a: Nat, b: Nat) -> Bool: Cmp.is_ge(Nat.cmp(a, b))

def Nat.min(+a: Nat, +b: Nat) -> Nat: match a b: case 0n _: 0n case 1n+ap 0n: 0n case 1n+ap 1n+bp: 1n+Nat.min(ap, bp)

def Nat.max(+a: Nat, +b: Nat) -> Nat: match a b: case 0n _: b case 1n+ap 0n: 1n+ap case 1n+ap 1n+bp: 1n+Nat.max(ap, bp)

law Nat.ge_refl: for a: Nat {Nat.is_ge(a, a) == True{} : Bool}

def Nat.ge_refl(a): match a: case 0n: {==} case 1n+p: Nat.ge_refl(p)

law Nat.max_ge_l: for +a: Nat for +b: Nat {Nat.is_ge(Nat.max(a, b), a) == True{} : Bool}

def Nat.max_ge_l(a, b): match a b: case 0n 0n: {==} case 0n 1n+bp: {==} case 1n+ap 0n: Nat.ge_refl(a) case 1n+ap 1n+bp: Nat.max_ge_l(ap, bp)

law Nat.max_ge_r: for +a: Nat for +b: Nat {Nat.is_ge(Nat.max(a, b), b) == True{} : Bool}

def Nat.max_ge_r(a, b): match a b: case 0n _: Nat.ge_refl(b) case 1n+ap 0n: {==} case 1n+ap 1n+bp: Nat.max_ge_r(ap, bp)

def Nat.div.fin(qr: Nat & Nat) -> Nat: (q, r) = qr q

def Nat.div(a: Nat, b: Nat) -> Nat: Nat.div.fin(Nat.divmod(a, b))

def Nat.mod.fin(qr: Nat & Nat) -> Nat: (q, r) = qr r

def Nat.mod(a: Nat, b: Nat) -> Nat: Nat.mod.fin(Nat.divmod(a, b))

def Nat.pow(+a: Nat, b: Nat) -> Nat: match b: case 0n: 1n case 1n+p: Nat.mul(a, Nat.pow(a, p))
```

Maybe

```
def Maybe.pure(a, -A: Kind(a), x: A) -> Maybe<a, A>: Some{x}

def Maybe.bind( a, -A: Kind(a), -B: Kind(a), m: Maybe<a, A>, f: A -> Maybe<a, B> ) -> Maybe<a, B>: match m: case None{}: None{} case Some{x}: f(x)

def Maybe.default(a, -A: Kind(a), m: Maybe<a, A>, d: A) -> A: match m: case None{}: d case Some{x}: x

def Maybe.is_some(a, -A: Kind(a), m: Maybe<a, A>) -> Bool: match m: case None{}: False{} case Some{x}: True{}
```

```
def Maybe.is_none(a, -A: Kind(a), m: Maybe<a, A>) -> Bool: Bool.not(Maybe.is_some(a, A, m))

def Maybe.map( a, -A: Kind(a), -B: Kind(a), f: A -> B, m: Maybe<a, A> ) -> Maybe<a, B>: match m: case None{}: None{} case Some{x}: Some{f(x)}

def Maybe.or(a, -A: Kind(a), m: Maybe<a, A>, n: Maybe<a, A>) -> Maybe<a, A>: match m: case None{}: n case Some{x}: Some{x}
```

Result

```
------

def Result.pure(a, b, -E: Kind(a), -A: Kind(b), x: A) -> Result<a, b, E, A>: Done{x}

def Result.bind( a, b, -E: Kind(a), -A: Kind(b), -B: Kind(b), r: Result<a, b, E, A>, f: A -> Result<a, b, E, B> ) -> Result<a, b, E, B>: match r: case Fail{e}: Fail{e} case Done{x}: f(x)

def Result.default( a, b, -E: Kind(a), -A: Kind(b), r: Result<a, b, E, A>, d: A ) -> A: match r: case Fail{e}: d case Done{x}: x

def Result.is_done( a, b, -E: Kind(a), -A: Kind(b), r: Result<a, b, E, A> ) -> Bool: match r: case Fail{e}: False{} case Done{x}: True{}

def Result.is_fail( a, b, -E: Kind(a), -A: Kind(b), r: Result<a, b, E, A> ) -> Bool: Bool.not(Result.is_done(a, b, E, A, r))

def Result.map( a, b, -E: Kind(a), -A: Kind(b), -B: Kind(b), f: A -> B, r: Result<a, b, E, A> ) -> Result<a, b, E, B>: match r: case Fail{e}: Fail{e} case Done{x}: Done{f(x)}
```

List

```
----

def List.map(~A: Type, ~B: Type, ~f: A -> B, xs: List<A>) -> List<B>: match xs: case Nil{}: Nil{} case h <> t: f(h) <> List.map(~A, ~B, ~f, t)

def List.length(a, -A: Kind(a), xs: List<a, A>) -> Nat: match xs: case Nil{}: 0n case h <> t: 1n+List.length(a, A, t)

def List.append(a, -A: Kind(a), xs: List<a, A>, ys: List<a, A>) -> List<a, A>: match xs: case Nil{}: ys case h <> t: h <> List.append(a, A, t, ys)

def List.concat(a, -A: Kind(a), xss: List<a, List<a, A>>) -> List<a, A>: match xss: case Nil{}: Nil{} case h <> t: List.append(a, A, h, List.concat(a, A, t))

def List.reverse.go(a, -A: Kind(a), xs: List<a, A>, acc: List<a, A>) -> List<a, A>: match xs: case Nil{}: acc case h <> t: List.reverse.go(a, A, t, h <> acc)

def List.reverse(a, -A: Kind(a), xs: List<a, A>) -> List<a, A>: List.reverse.go(a, A, xs, Nil{})

def List.is_empty(a, -A: Kind(a), xs: List<a, A>) -> Bool: match xs: case Nil{}: True{} case h <> t: False{}

def List.head(a, -A: Kind(a), xs: List<a, A>) -> Maybe<a, A>: match xs: case Nil{}: None{} case h <> t: Some{h}

def List.tail(a, -A: Kind(a), xs: List<a, A>) -> List<a, A>: match xs: case Nil{}: Nil{} case h <> t: t

def List.last.go(a, -A: Kind(a), xs: List<a, A>, last: A) -> A: match xs: case Nil{}: last case h <> t: List.last.go(a, A, t, h)

def List.last(a, -A: Kind(a), xs: List<a, A>) -> Maybe<a, A>: match xs: case Nil{}: None{} case h <> t: Some{List.last.go(a, A, t, h)}

def List.get(a, -A: Kind(a), xs: List<a, A>, n: Nat) -> Maybe<a, A>: match xs n: case Nil{} _: None{} case h <> t 0n: Some{h} case h <> t 1n+p: List.get(a, A, t, p)

def List.set(a, -A: Kind(a), xs: List<a, A>, n: Nat, x: A) -> List<a, A>: match xs n: case Nil{} _: Nil{} case h <> t 0n: x <> t case h <> t 1n+p: h <> List.set(a, A, t, p, x)

def List.take(a, -A: Kind(a), xs: List<a, A>, n: Nat) -> List<a, A>: match xs n: case Nil{} _: Nil{} case h <> t 0n: Nil{} case h <> t 1n+p: h <> List.take(a, A, t, p)

def List.drop(a, -A: Kind(a), xs: List<a, A>, n: Nat) -> List<a, A>: match xs n: case Nil{} _: Nil{} case h <> t 0n: h <> t case h <> t 1n+p: List.drop(a, A, t, p)
```

```
def List.zip( a, -A: Kind(a), b, -B: Kind(b), xs: List<a, A>, ys: List<b, B> ) -> List<&1, A & B>: match xs ys: case Nil{} _: Nil{} case h <> t Nil{}: Nil{} case x <> xt y <> yt: (x, y) <> List.zip(a, A, b, B, xt, yt)

def List.range.go(+n: Nat, acc: List<&2, Nat>) -> List<&2, Nat>: match n: case 0n: acc case 1n+p: List.range.go(p, p <> acc)

def List.range(n: Nat) -> List<&2, Nat>: List.range.go(n, Nil{})

def List.replicate(-A: Data, n: Nat, +x: A) -> List<&2, A>: match n: case 0n: Nil{} case 1n+p: x <> List.replicate(A, p, x)

def List.filter.put(-A: Data, h: A, r: List<&2, A>, keep: Bool) -> List<&2, A>: match keep: case False{}: r case True{}: h <> r

def List.filter(~A: Data, ~f: A -> Bool, xs: List<&2, A>) -> List<&2, A>: match xs: case Nil{}: Nil{} case +h <> t: List.filter.put(A, h, List.filter(~A, ~f, t), f(h))

def List.foldl( ~a: Quant, ~A: Kind(a), ~B: Type, ~f: B -> A -> B, xs: List<a, A>, acc: B ) -> B: match xs: case Nil{}: acc case h <> t: List.foldl(~a, ~A, ~B, ~f, t, f(acc, h))

def List.foldr( ~a: Quant, ~A: Kind(a), ~B: Type, ~f: A -> B -> B, xs: List<a, A>, z: B ) -> B: match xs: case Nil{}: z case h <> t: f(h, List.foldr(~a, ~A, ~B, ~f, t, z))

def List.any(~a: Quant, ~A: Kind(a), ~f: A -> Bool, xs: List<a, A>) -> Bool: match xs: case Nil{}: False{} case h <> t: Bool.or(f(h), List.any(~a, ~A, ~f, t))

def List.all(~a: Quant, ~A: Kind(a), ~f: A -> Bool, xs: List<a, A>) -> Bool: match xs: case Nil{}: True{} case h <> t: Bool.and(f(h), List.all(~a, ~A, ~f, t))

def List.find.put(-A: Data, h: A, r: Maybe<&2, A>, hit: Bool) -> Maybe<&2, A>: match hit: case False{}: r case True{}: Some{h}

def List.find(~A: Data, ~f: A -> Bool, xs: List<&2, A>) -> Maybe<&2, A>: match xs: case Nil{}: None{} case +h <> t: List.find.put(A, h, List.find(~A, ~f, t), f(h))

def List.contains( ~A: Data, ~eq: A -> A -> Bool, xs: List<&2, A>, +x: A ) -> Bool: match xs: case Nil{}: False{} case h <> t: Bool.or(eq(h, x), List.contains(~A, ~eq, t, x))

def List.merge.step( -A: Data, acc: List<&2, A>, x: A, xt: List<&2, A>, y: A, yt: List<&2, A>, le: Bool ) -> List<&2, A> & List<&2, A> & List<&2, A>: match le: case False{}: (y <> acc, x <> xt, yt) case True{}: (x <> acc, xt, y <> yt)
```

The fuel bounds the steps: a sort passes its length, enough for any merge.

```
def List.merge.go( ~A: Data, ~le: A -> A -> Bool, fuel: Nat, st: List<&2, A> & List<&2, A> & List<&2, A> ) -> List<&2, A>: match fuel: case 0n: (acc, xs, ys) = st List.reverse.go(&2, A, acc, List.append(&2, A, xs, ys)) case 1n+f: (acc, xs, ys) = st match xs ys: case Nil{} Nil{}: List.reverse.go(&2, A, acc, Nil{}) case Nil{} y <> yt: List.reverse.go(&2, A, acc, y <> yt) case x <> xt Nil{}: List.reverse.go(&2, A, acc, x <> xt) case +x <> xt +y <> yt: List.merge.go(~A, ~le, f, List.merge.step(A, acc, x, xt, y, yt, le(x, y)))

def List.sort.runs(-A: Data, xs: List<&2, A>) -> List<&2, List<&2, A>>: match xs: case Nil{}: Nil{} case h <> t: [h] <> List.sort.runs(A, t)

def List.sort.pass( ~A: Data, ~le: A -> A -> Bool, +n: Nat, runs: List<&2, List<&2, A>> ) -> List<&2, List<&2, A>>: match runs: case Nil{}: Nil{} case r <> Nil{}: [r] case r1 <> r2 <> rest: List.merge.go(~A, ~le, n, (Nil{}, r1, r2)) <> List.sort.pass(~A, ~le, n, rest)

def List.sort.go( ~A: Data, ~le: A -> A -> Bool, fuel: Nat, +n: Nat, runs: List<&2, List<&2, A>> ) -> List<&2, A>: match fuel: case 0n: List.concat(&2, A, runs) case 1n+f: match runs: case Nil{}: Nil{} case r <> Nil{}: r case r1 <> r2 <> rest: List.sort.go(~A, ~le, f, n, List.sort.pass(~A, ~le, n, r1 <> r2 <> rest))

def List.sort(~A: Data, ~le: A -> A -> Bool, +xs: List<&2, A>) -> List<&2, A>: +n = List.length(&2, A, xs) List.sort.go(~A, ~le, n, n, List.sort.runs(A, xs))

def List.for_each( ~a: Quant, ~A: Kind(a), ~f: A -> IO(Unit), xs: List<a, A> ) -> IO(Unit): match xs: case Nil{}: IO.pure(Unit, Unit{}) case h <> t: IO.bind(Unit, Unit, f(h), u => List.for_each(~a, ~A, ~f, t))
```

Word

```
def Word.zero(n: Nat) -> Word(n): match n: case 0n: WNil{} case 1n+p: WCon{False{}, Word.zero(p)}
```

```
def Word.not(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: WCon{Bool.not(b), Word.not(p, t)}

def Word.and(n: Nat, a: Word(n), b: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: WCon{Bool.and(ab, bb), Word.and(p, at, bt)}

def Word.or(n: Nat, a: Word(n), b: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: WCon{Bool.or(ab, bb), Word.or(p, at, bt)}

def Word.xor(n: Nat, a: Word(n), b: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: WCon{Bool.xor(ab, bb), Word.xor(p, at, bt)}

def Word.shl.put(n: Nat, c: Bool, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: WCon{c, Word.shl.put(p, b, t)}

def Word.shl(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: WCon{False{}, Word.shl.put(p, b, t)}

def Word.shl.out.con(-p: Nat, c: Bool, r: Bool & Word(p)) -> Bool & Word(1n+p): (hi, t2) = r (hi, WCon{c, t2})

def Word.shl.out(n: Nat, c: Bool, w: Word(n)) -> Bool & Word(n): match n: case 0n: (c, WNil{}) case 1n+p: match w: case WCon{b, t}: Word.shl.out.con(p, c, Word.shl.out(p, b, t))

def Word.shr.pad(n: Nat, w: Word(n)) -> Word(1n+n): match n: case 0n: WCon{False{}, w} case 1n+p: match w: case WCon{b, t}: WCon{b, Word.shr.pad(p, t)}

def Word.shr(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: Word.shr.pad(p, t)

def Word.cmp.fin(ab: Bool, bb: Bool, t: Cmp) -> Cmp: match t: case LT{}: LT{} case EQ{}: Bool.cmp(ab, bb) case GT{}: GT{}

def Word.cmp(n: Nat, a: Word(n), b: Word(n)) -> Cmp: match n: case 0n: EQ{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: Word.cmp.fin(ab, bb, Word.cmp(p, at, bt))

def Word.inc(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{False{}, t}: WCon{True{}, t} case WCon{True{}, t}: WCon{False{}, Word.inc(p, t)}

law Word.adc: for n: Nat for a: Word(n) for b: Word(n) for f: Bool for c: Bool Word(n)

def Word.adc.con(p: Nat, at: Word(p), bt: Word(p), f: Bool, sk: Bool & Bool) -> Word(1n+p): (s, k) = sk WCon{s, Word.adc(p, at, bt, f, k)}

def Word.adc(n, a, b, f, c): match n: case 0n: WNil{} case 1n+p: match a b f: case WCon{ab, at} WCon{bb, bt} False{}: Word.adc.con(p, at, bt, False{}, Bool.full_add(ab, bb, c)) case WCon{ab, at} WCon{bb, bt} True{}: Word.adc.con(p, at, bt, True{}, Bool.full_add(ab, Bool.not(bb), c))

def Word.add(n: Nat, a: Word(n), b: Word(n)) -> Word(n): Word.adc(n, a, b, False{}, False{})

def Word.sub(n: Nat, a: Word(n), b: Word(n)) -> Word(n): Word.adc(n, a, b, True{}, True{})

law Word.add_comm.arm: for p : Nat for -h : Bool for at : Word(p) for bt : Word(p) for k : Bool {WCon{h, Word.adc(p, at, bt, False{}, k)} == WCon{h, Word.adc(p, bt, at, False{}, k)} : Word.Con

}

law Word.add_comm.go: for n: Nat for a: Word(n) for b: Word(n) for c: Bool {Word.adc(n, a, b, False{}, c) == Word.adc(n, b, a, False{}, c) : Word(n)}

def Word.add_comm.arm(p, h, at, bt, k): Equal.cong(Word(p), Word(1n+p), w => WCon{h, w}, Word.adc(p, at, bt, False{}, k), Word.adc(p, bt, at, False{}, k), Word.add_comm.go(p, at, bt, k))

def Word.add_comm.go(n, a, b, c): match n: case 0n: match a b: case WNil{} WNil{}: {==} case 1n+p: match a b c: case WCon{False{}, at} WCon{False{}, bt} False{}: Word.add_comm.arm(p, False{}, at, bt, False{}) case WCon{False{}, at} WCon{False{}, bt} True{}: Word.add_comm.arm(p, True{}, at, bt, False{}) case WCon{False{}, at} WCon{True{}, bt} False{}: Word.add_comm.arm(p, True{}, at, bt, False{}) case WCon{False{}, at} WCon{True{}, bt} True{}: Word.add_comm.arm(p, False{}, at, bt, True{}) case WCon{True{}, at} WCon{False{}, bt} False{}: Word.add_comm.arm(p, True{}, at, bt, False{}) case WCon{True{}, at} WCon{False{}, bt} True{}: Word.add_comm.arm(p, False{}, at, bt, True{}) case WCon{True{}, at} WCon{True{}, bt} False{}: Word.add_comm.arm(p, False{}, at, bt, True{}) case WCon{True{}, at} WCon{True{}, bt} True{}: Word.add_comm.arm(p, True{}, at, bt, True{})

law Word.add_comm: for n: Nat for a: Word(n) for b: Word(n) {Word.add(n, a, b) == Word.add(n, b, a) : Word(n)}
```

```
def Word.add_comm(n, a, b): Word.add_comm.go(n, a, b, False{})

def Word.to_nat(n: Nat, w: Word(n)) -> Nat: match n: case 0n: 0n case 1n+p: match w: case WCon{False{}, t}: Nat.double(Word.to_nat(p, t)) case WCon{True{}, t}: 1n+Nat.double(Word.to_nat(p, t))

def Word.mul.go(+n: Nat, m: Nat, a: Word(m), +b: Word(n), acc: Word(n)) -> Word(n): match m: case 0n: acc case 1n+mp: match a: case WCon{False{}, at}: Word.mul.go(n, mp, at, Word.shl(n, b), acc) case WCon{True{}, at}: Word.mul.go(n, mp, at, Word.shl(n, b), Word.add(n, acc, b))

def Word.mul(+n: Nat, a: Word(n), b: Word(n)) -> Word(n): Word.mul.go(n, n, a, b, Word.zero(n))
```

U32

```
def U32.inc(a: U32) -> U32: match a: case U32{x}: U32{Word.inc(32n, x)}

def U32.add(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.add(32n, x, y)}

law U32.add_comm: for a: U32 for b: U32 {U32.add(a, b) == U32.add(b, a) : U32}

def U32.add_comm(a, b): match a b: case U32{x} U32{y}: Equal.cong(Word(32n), U32, w => U32{w}, Word.add(32n, x, y), Word.add(32n, y, x), Word.add_comm(32n, x, y))

def U32.sub(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.sub(32n, x, y)}

def U32.mul(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.mul(32n, x, y)}

def U32.not(a: U32) -> U32: match a: case U32{x}: U32{Word.not(32n, x)}

def U32.and(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.and(32n, x, y)}

def U32.or(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.or(32n, x, y)}

def U32.xor(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.xor(32n, x, y)}

def U32.shl(a: U32) -> U32: match a: case U32{x}: U32{Word.shl(32n, x)}

def U32.shr(a: U32) -> U32: match a: case U32{x}: U32{Word.shr(32n, x)}

def U32.shln(a: U32, n: Nat) -> U32: match n: case 0n: a case 1n+p: U32.shl(U32.shln(a, p))

def U32.shrn(a: U32, n: Nat) -> U32: match n: case 0n: a case 1n+p: U32.shr(U32.shrn(a, p))

def U32.cmp(a: U32, b: U32) -> Cmp: match a b: case U32{x} U32{y}: Word.cmp(32n, x, y)

def U32.is_eq(a: U32, b: U32) -> Bool: Cmp.is_eq(U32.cmp(a, b))

def U32.is_ne(a: U32, b: U32) -> Bool: Bool.not(Cmp.is_eq(U32.cmp(a, b)))

def U32.is_lt(a: U32, b: U32) -> Bool: Cmp.is_lt(U32.cmp(a, b))

def U32.is_le(a: U32, b: U32) -> Bool: Cmp.is_le(U32.cmp(a, b))

def U32.is_gt(a: U32, b: U32) -> Bool: Cmp.is_gt(U32.cmp(a, b))

def U32.is_ge(a: U32, b: U32) -> Bool: Cmp.is_ge(U32.cmp(a, b))

def U32.is_zero(a: U32) -> Bool: U32.is_eq(a, 0)

def U32.to_nat(a: U32) -> Nat: match a: case U32{w}: Word.to_nat(32n, w)

def U32.from_nat(n: Nat) -> U32: match n: case 0n: 0 case 1n+p: U32.inc(U32.from_nat(p))

def U32.log2.go(+k: Nat, +n: U32, +d: Nat) -> Nat: match k: case 0n: d case 1n+p: U32.log2.go(p, U32.shr(n), Nat.add(d, U32.to_nat(Bool.to_u32(U32.is_lt(1, n)))))

def U32.log2(+n: U32) -> Nat: U32.log2.go(32n, n, 0n)

def U32.divmod.go.fin(-p: Nat, q: Word(p), s: U32, b: U32, g: Bool) -> Word(1n+p) & U32: match g: case True{}: (WCon{True{}, q}, U32.sub(s, b)) case False{}: (WCon{False{}, q}, s)

def U32.divmod.go.shl(-p: Nat, q: Word(p), +b: U32, ts: Bool & Word(32n)) -> Word(1n+p) & U32: (t, s) = ts +s2 = {U32{s} : U32} U32.divmod.go.fin(p, q, s2, b, Bool.or(t, U32.is_ge(s2, b)))
```

```
def U32.divmod.go.rec(-p: Nat, a0: Bool, +b: U32, qr: Word(p) & U32) -> Word(1n+p) & U32: (q, r) = qr U32{rw} = r U32.divmod.go.shl(p, q, b, Word.shl.out(32n, a0, rw))

def U32.divmod.go(m: Nat, a: Word(m), +b: U32) -> Word(m) & U32: match m a: case 0n WNil{}: (WNil{}, 0) case 1n+p WCon{a0, hi}: U32.divmod.go.rec(p, a0, b, U32.divmod.go(p, hi, b))

def U32.div.fin(qr: Word(32n) & U32) -> U32: (q, r) = qr U32{q}

def U32.div.if(aw: Word(32n), b: U32, z: Bool) -> U32: match z: case True{}: 0 case False{}: U32.div.fin(U32.divmod.go(32n, aw, b))

def U32.div(a: U32, +b: U32) -> U32: U32{aw} = a U32.div.if(aw, b, U32.is_zero(b))

def U32.mod.fin(qr: Word(32n) & U32) -> U32: (q, r) = qr r

def U32.mod.if(aw: Word(32n), b: U32, z: Bool) -> U32: match z: case True{}: U32{aw} case False{}: U32.mod.fin(U32.divmod.go(32n, aw, b))

def U32.mod(a: U32, +b: U32) -> U32: U32{aw} = a U32.mod.if(aw, b, U32.is_zero(b))

def U32.min(+a: U32, +b: U32) -> U32: Bool.pick(U32, U32.is_lt(a, b), a, b)

def U32.max(+a: U32, +b: U32) -> U32: Bool.pick(U32, U32.is_lt(a, b), b, a)

def U32.clamp(x: U32, lo: U32, hi: U32) -> U32: U32.min(U32.max(x, lo), hi)

def U32.pow(+a: U32, n: Nat) -> U32: match n: case 0n: 1 case 1n+p: U32.mul(a, U32.pow(a, p))

def U32.is_even(a: U32) -> Bool: U32.is_zero(U32.and(a, 1))
```

F32

```
law U32.to_f32: for a: U32 F32

law F32.to_u32: for a: F32 U32

law F32.add: for a: F32 for b: F32 F32

law F32.sub: for a: F32 for b: F32 F32

law F32.mul: for a: F32 for b: F32 F32

law F32.div: for a: F32 for b: F32 F32

law F32.mod: for a: F32 for b: F32 F32

law F32.pow: for a: F32 for b: F32 F32

law F32.atan2: for a: F32 for b: F32 F32

law F32.neg: for a: F32 F32

law F32.abs: for a: F32 F32

law F32.sqrt: for a: F32 F32

law F32.exp: for a: F32 F32

law F32.log: for a: F32 F32

law F32.log2: for a: F32 F32

law F32.log10: for a: F32 F32

law F32.sin: for a: F32 F32

law F32.cos: for a: F32 F32

law F32.tan: for a: F32 F32

law F32.asin: for a: F32 F32

law F32.acos: for a: F32 F32
```

```
law F32.atan: for a: F32 F32
law F32.sinh: for a: F32 F32
law F32.cosh: for a: F32 F32
law F32.tanh: for a: F32 F32
law F32.floor: for a: F32 F32
law F32.ceil: for a: F32 F32
law F32.trunc: for a: F32 F32
law F32.is_eq: for a: F32 for b: F32 Bool
law F32.is_ne: for a: F32 for b: F32 Bool
law F32.is_lt: for a: F32 for b: F32 Bool
law F32.is_le: for a: F32 for b: F32 Bool
law F32.is_gt: for a: F32 for b: F32 Bool
law F32.is_ge: for a: F32 for b: F32 Bool
law F32.show: for +a: F32 String
law F32.bits: for a: F32 U32
law F32.read: for s: String Maybe<&2, F32>
def F32.min(+a: F32, +b: F32) -> F32: Bool.pick(F32, F32.is_lt(a, b), a, b)
def F32.max(+a: F32, +b: F32) -> F32: Bool.pick(F32, F32.is_lt(a, b), b, a)
def F32.clamp(x: F32, lo: F32, hi: F32) -> F32: F32.min(F32.max(x, lo), hi)
def F32.lerp(+a: F32, b: F32, t: F32) -> F32: F32.add(a, F32.mul(F32.sub(b, a), t))
def F32.square(+a: F32) -> F32: F32.mul(a, a)
def F32.hypot(+x: F32, +y: F32) -> F32: F32.sqrt(F32.add(F32.mul(x, x), F32.mul(y, y)))
def F32.round(a: F32) -> F32: F32.floor(F32.add(a, 0.5))
def F32.pi() -> F32: 3.14159265
def F32.from_nat(n: Nat) -> F32: U32.to_f32(U32.from_nat(n))
def F32.to_nat(a: F32) -> Nat: U32.to_nat(F32.to_u32(a))
```

Char

```
----
def Char.cmp(a: Char, b: Char) -> (Char & Char) & Cmp: match a b: case Chr{+x} Chr{+y}: ((Chr{x}, Chr{y}), U32.cmp(x, y))
def Char.to_u32(c: Char) -> U32: match c: case Chr{x}: x
def Char.from_u32(x: U32) -> Char: Chr{x}
def Char.is_eq(a: Char, b: Char) -> Bool: match a b: case Chr{x} Chr{y}: U32.is_eq(x, y)
def Char.is_digit(c: Char) -> Bool: match c: case Chr{+x}: Bool.and(U32.is_ge(x, 48), U32.is_le(x, 57))
def Char.is_upper(c: Char) -> Bool: match c: case Chr{+x}: Bool.and(U32.is_ge(x, 65), U32.is_le(x, 90))
def Char.is_lower(c: Char) -> Bool: match c: case Chr{+x}: Bool.and(U32.is_ge(x, 97), U32.is_le(x, 122))
def Char.is_alpha(+c: Char) -> Bool: Bool.or(Char.is_upper(c), Char.is_lower(c))
def Char.is_space(c: Char) -> Bool: match c: case Chr{+x}: Bool.or(U32.is_eq(x, 32), Bool.and(U32.is_ge(x, 9), U32.is_le(x, 13)))
def Char.to_upper(+c: Char) -> Char: Chr{U32.sub(Char.to_u32(c), U32.mul(Bool.to_u32(Char.is_lower(c)), 32))}
```

```
def Char.to_lower(+c: Char) -> Char: Chr{U32.add(Char.to_u32(c), U32.mul(Bool.to_u32(Char.is_upper(c)), 32))}
```

String

```
def String.append(a: String, b: String) -> String: match a: case SNil{}: b case SCon{h, t}: SCon{h, String.append(t, b)}

def String.cmp.rec(h1b: Char, h2b: Char, rr: (String & String) & Cmp) -> (String & String) & Cmp: ((t1b, t2b), r) = rr ((SCon{h1b, t1b}, SCon{h2b, t2b}), r)

law String.cmp: for a: String for b: String (String & String) & Cmp

def String.cmp.fin(t1: String, t2: String, hc: (Char & Char) & Cmp) -> (String & String) & Cmp: ((h1b, h2b), c) = hc match c: case LT{}: ((SCon{h1b, t1}, SCon{h2b, t2}), LT{}) case EQ{}: String.cmp.rec(h1b, h2b, String.cmp(t1, t2)) case GT{}: ((SCon{h1b, t1}, SCon{h2b, t2}), GT{})

def String.cmp(a, b): match a b: case SNil{} SNil{}: ((SNil{}, SNil{}), EQ{}) case SNil{} SCon{h, t}: ((SNil{}, SCon{h, t}), LT{}) case SCon{h, t} SNil{}: ((SCon{h, t}, SNil{}), GT{}) case SCon{h1, t1} SCon{h2, t2}: String.cmp.fin(t1, t2, Char.cmp(h1, h2))

def String.eq.fin(r: (String & String) & Cmp) -> Bool: ((a2, b2), c) = r Cmp.is_eq(c)

def String.eq(a: String, b: String) -> Bool: String.eq.fin(String.cmp(a, b))

def String.length(s: String) -> Nat: match s: case SNil{}: 0n case SCon{h, t}: 1n+String.length(t)

def String.is_empty(s: String) -> Bool: match s: case SNil{}: True{} case SCon{h, t}: False{}

def String.reverse.go(s: String, acc: String) -> String: match s: case SNil{}: acc case SCon{h, t}: String.reverse.go(t, SCon{h, acc})

def String.reverse(s: String) -> String: String.reverse.go(s, SNil{})

def String.order.fin(r: (String & String) & Cmp) -> Cmp: (ab, c) = r c

def String.order(a: String, b: String) -> Cmp: String.order.fin(String.cmp(a, b))

def String.is_lt(a: String, b: String) -> Bool: Cmp.is_lt(String.order(a, b))

def String.is_le(a: String, b: String) -> Bool: Cmp.is_le(String.order(a, b))

def String.is_gt(a: String, b: String) -> Bool: Cmp.is_gt(String.order(a, b))

def String.is_ge(a: String, b: String) -> Bool: Cmp.is_ge(String.order(a, b))

law String.starts_with: for s: String for p: String Bool

def String.starts_with.if(t: String, pt: String, same: Bool) -> Bool: match same: case False{}: False{} case True{}: String.starts_with(t, pt)

def String.starts_with(s, p): match s p: case SNil{} SNil{}: True{} case SNil{} SCon{h, t}: False{} case SCon{h, t} SNil{}: True{} case SCon{x, xt} SCon{y, yt}: String.starts_with.if(xt, yt, Char.is_eq(x, y))

def String.ends_with(s: String, p: String) -> Bool: String.starts_with(String.reverse(s), String.reverse(p))

law String.contains: for +s: String for +p: String Bool

def String.contains.if(t: String, p: String, here: Bool) -> Bool: match here: case False{}: String.contains(t, p) case True{}: True{}

def String.contains(s, p): match s: case SNil{}: String.is_empty(p) case SCon{h, t}: String.contains.if(t, p, String.starts_with(SCon{h, t}, p))

def String.take(s: String, n: Nat) -> String: match s n: case SNil{} _: SNil{} case SCon{h, t} 0n: SNil{} case SCon{h, t} 1n+p: SCon{h, String.take(t, p)}

def String.drop(s: String, n: Nat) -> String: match s n: case SNil{} _: SNil{} case SCon{h, t} 0n: SCon{h, t} case SCon{h, t} 1n+p: String.drop(t, p)

def String.get(s: String, n: Nat) -> Maybe<&2, Char>: match s n: case SNil{} _: None{} case SCon{h, t} 0n: Some{h} case SCon{h, t} 1n+p: String.get(t, p)

def String.to_list(s: String) -> List<&2, Char>: match s: case SNil{}: Nil{} case SCon{h, t}: h <> String.to_list(t)
```

```
def String.from_list(cs: List<&2, Char>) -> String: match cs: case Nil{}: SNil{} case h <> t: SCon{h, String.from_list(t)}

def String.concat(xs: List<&2, String>) -> String: match xs: case Nil{}: SNil{} case h <> t: String.append(h, String.concat(t))

def String.join.go(xs: List<&2, String>, h: String, +sep: String) -> String: match xs: case Nil{}: h case h2 <> t: String.append(h, String.append(sep, String.join.go(t, h2, sep)))

def String.join(xs: List<&2, String>, +sep: String) -> String: match xs: case Nil{}: SNil{} case h <> t: String.join.go(t, h, sep)

def String.split.push(c: Char, ps: List<&2, String>) -> List<&2, String>: match ps: case Nil{}: [SCon{c, SNil{}}] case h <> t: SCon{c, h} <> t

def String.split.fin(c: Char, r: List<&2, String>, cut: Bool) -> List<&2, String>: match cut: case False{}: String.split.push(c, r) case True{}: SNil{} <> r

def String.split(s: String, +sep: Char) -> List<&2, String>: match s: case SNil{}: [SNil{}] case SCon{+h, t}: String.split.fin(h, String.split(t, sep), Char.is_eq(h, sep))

def String.lines(s: String) -> List<&2, String>: String.split(s, '\n')

def String.repeat(+s: String, n: Nat) -> String: match n: case 0n: SNil{} case 1n+p: String.append(s, String.repeat(s, p))

def String.to_upper(s: String) -> String: match s: case SNil{}: SNil{} case SCon{h, t}: SCon{Char.to_upper(h), String.to_upper(t)}

def String.to_lower(s: String) -> String: match s: case SNil{}: SNil{} case SCon{h, t}: SCon{Char.to_lower(h), String.to_lower(t)}

law String.trim_start: for s: String String

def String.trim_start.if(h: Char, t: String, space: Bool) -> String: match space: case False{}: SCon{h, t} case True{}: String.trim_start(t)

def String.trim_start(s): match s: case SNil{}: SNil{} case SCon{+h, t}: String.trim_start.if(h, t, Char.is_space(h))

def String.trim_end(s: String) -> String: String.reverse(String.trim_start(String.reverse(s)))

def String.trim(s: String) -> String: String.trim_end(String.trim_start(s))
```

Text

```
def Nat.show.put(qr: Nat & Nat) -> Char & Nat: (q, r) = qr (Chr{U32.from_nat(Nat.add(48n, r))}, q)

law Nat.show.go: for f : Nat for n : Nat for acc : String String

def Nat.show.fin(g: Nat, acc: String, dq: Char & Nat) -> String: (d, q) = dq match q: case 0n: SCon{d, acc} case 1n+p: Nat.show.go(g, 1n+p, SCon{d, acc})

def Nat.show.go(f, n, acc): match f: case 0n: acc case 1n+g: Nat.show.fin(g, acc, Nat.show.put(Nat.divmod(n, 10n)))

def Nat.show(n: Nat) -> String: +m = n Nat.show.fin(m, SNil{}, Nat.show.put(Nat.divmod(m, 10n)))

law Nat.read.go: for s : String for +acc : Nat Maybe<&2, Nat>

def Nat.read.max() -> Nat: Nat.mul(U32.to_nat(16777215), U32.to_nat(16777217))

def Nat.read.fit(acc: Nat, qr: Nat & Nat) -> Bool: (q, r) = qr Bool.not(Nat.is_lt(q, acc))

def Nat.read.if(t: String, acc: Nat, d: U32, ok: Bool) -> Maybe<&2, Nat>: match ok: case True{}: Nat.read.go(t, Nat.add(Nat.mul(acc, 10n), U32.to_nat(d))) case False{}: None{}

def Nat.read.go(s, acc): match s: case SNil{}: Some{acc} case SCon{Chr{x}, t}: +d = U32.sub(x, 48) Nat.read.if(t, acc, d, Bool.and(U32.is_lt(d, 10), Nat.read.fit(acc, Nat.divmod(Nat.sub(Nat.read.max(), U32.to_nat(d)), 10n))))

def Nat.read(s: String) -> Maybe<&2, Nat>: match s: case SNil{}: None{} case SCon{h, t}: Nat.read.go(SCon{h, t}, 0n)

law U32.show.go: for f : Nat for +n : U32 for acc : String String

def U32.show.fin(g: Nat, acc: String, +n: U32, z: Bool) -> String: match z: case True{}: acc case False{}: U32.show.go(g, U32.div(n, 10), SCon{Chr{U32.add(48, U32.mod(n, 10))}, acc})
```

```
def U32.show.go(f, n, acc): match f: case 0n: acc case 1n+g: U32.show.fin(g, acc, n, U32.is_zero(n))

def U32.show.if(a: U32, z: Bool) -> String: match z: case True{}: SCon{Chr{48}, SNil{}} case False{}: U32.show.go(10n, a, SNil{})

def U32.show(a: U32) -> String: +b = a U32.show.if(b, U32.is_zero(b))

law U32.read.go: for s : String for +acc : U32 Maybe<&2, U32>

def U32.read.if(t: String, n: U32, ok: Bool) -> Maybe<&2, U32>: match ok: case True{}: U32.read.go(t, n) case False{}: None{}

def U32.read.go(s, acc): match s: case SNil{}: Some{acc} case SCon{Chr{x}, t}: +n = U32.add(U32.mul(acc, 10), U32.sub(x, 48)) U32.read.if(t, n, U32.is_eq(U32.div(n, 10), acc))

def U32.read(s: String) -> Maybe<&2, U32>: match s: case SNil{}: None{} case SCon{h, t}: U32.read.go(SCon{h, t}, 0)

def Bool.show(b: Bool) -> String: match b: case False{}: "False" case True{}: "True"

def Char.show(c: Char) -> String: SCon{c, SNil{}}

def Maybe.show(~a: Quant, ~A: Kind(a), ~f: A -> String, m: Maybe<a, A>) -> String: match m: case None{}: "None" case Some{x}: "Some(" ++ f(x) ++ ")"

def List.show.go(~a: Quant, ~A: Kind(a), ~f: A -> String, xs: List<a, A>) -> String: match xs: case Nil{}: "]" case h <> t: ", " ++ f(h) ++ List.show.go(~a, ~A, ~f, t)

def List.show(~a: Quant, ~A: Kind(a), ~f: A -> String, xs: List<a, A>) -> String: match xs: case Nil{}: "[]" case h <> t: "[" ++ f(h) ++ List.show.go(~a, ~A, ~f, t)
```

Array

```
def Array.size.node(-T: Type, ys: Array<T>, r: Array<T> & U32) -> Array<T> & U32: (xs2, n) = r (ANode{xs2, ys}, U32.shl(n))

def Array.size(-T: Type, a: Array<T>) -> Array<T> & U32: match a: case ALeaf{x}: (ALeaf{x}, 1) case ANode{xs, ys}: Array.size.node(T, ys, Array.size(T, xs))

def Array.swap.lo(-T: Type, ys: Array<T>, r: Array<T> & T) -> Array<T> & T: (nxs, old) = r (ANode{nxs, ys}, old)

def Array.swap.hi(-T: Type, xs: Array<T>, r: Array<T> & T) -> Array<T> & T: (nys, old) = r (ANode{xs, nys}, old)

law Array.swap.go: for -T: Type for a: Array<T> for n: U32 for +i: U32 for v: T Array<T> & T

def Array.swap.if( -T: Type, xs: Array<T>, ys: Array<T>, +h: U32, i: U32, v: T, z: Bool ) -> Array<T> & T: match z: case True{}: Array.swap.lo(T, ys, Array.swap.go(T, xs, h, i, v)) case False{}: Array.swap.hi(T, xs, Array.swap.go(T, ys, h, U32.sub(i, h), v))

def Array.swap.go(T, a, n, i, v): match a: case ALeaf{x}: (ALeaf{v}, x) case ANode{xs, ys}: +h = U32.shr(n) Array.swap.if(T, xs, ys, h, i, v, U32.is_lt(i, h))

def Array.swap.at(-T: Type, i: U32, v: T, an: Array<T> & U32) -> Array<T> & T: (a, +n) = an Array.swap.go(T, a, n, U32.and(i, U32.sub(n, 1)), v)

def Array.swap(-T: Type, a: Array<T>, +i: U32, v: T) -> Array<T> & T: Array.swap.at(T, i, v, Array.size(T, a))

def Array.clone.node( -T: Data, cx: Array<T> & Array<T>, cy: Array<T> & Array<T> ) -> Array<T> & Array<T>: (xa, xb) = cx (ya, yb) = cy (ANode{xa, ya}, ANode{xb, yb})

def Array.clone(-T: Data, a: Array<T>) -> Array<T> & Array<T>: match a: case ALeaf{+x}: (ALeaf{x}, ALeaf{x}) case ANode{xs, ys}: Array.clone.node(T, Array.clone(T, xs), Array.clone(T, ys))
```

A fork is two handles to one array in O(1). Both read and write the same cells, so racing writes go through the atomics; a match on a handle copies its part, as a clone does. An atomic takes an Array<U32> (fadd an Array<F32>) and returns the array with the old value.

```
@unsafe def Array.fork(-T: Type, +a: Array<T>) -> Array<T> & Array<T>: (a, a)

@unsafe def Array.join(-T: Type, a: Array<T>, b: Array<T>) -> Array<T>: a

law Array.atomic.add: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.min: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.max: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.and: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.or: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.xor: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.exch: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.cas: for a: Array<U32> for +i: U32 for +x: U32 for +v: U32 Array<U32> & U32

law Array.atomic.fadd: for a: Array<F32> for +i: U32 for +v: F32 Array<F32> & F32

def Array.new(-T: Data, +d: Nat, +v: T) -> Array<T>: match d: case 0n: ALeaf{v} case 1n+p: ANode{[v : T^p], [v : T^p]}

def Array.set.fin(-T: Type, r: Array<T> & T) -> Array<T>: (na, old) = r na

def Array.set(-T: Type, a: Array<T>, i: U32, v: T) -> Array<T>: Array.set.fin(T, Array.swap(T, a, i, v))

law Array.get.go: for -T: Data for a: Array<T> for n: U32 for +i: U32 Array<T> & T

def Array.get.if( -T: Data, xs: Array<T>, ys: Array<T>, +h: U32, i: U32, z: Bool ) -> Array<T> & T: match z: case True{}: Array.swap.lo(T, ys, Array.get.go(T, xs, h, i)) case False{}: Array.swap.hi(T, xs, Array.get.go(T, ys, h, U32.sub(i, h)))

def Array.get.go(T, a, n, i): match a: case ALeaf{+x}: (ALeaf{x}, x) case ANode{xs, ys}: +h = U32.shr(n) Array.get.if(T, xs, ys, h, i, U32.is_lt(i, h))

def Array.get.at(-T: Data, i: U32, an: Array<T> & U32) -> Array<T> & T: (a, +n) = an Array.get.go(T, a, n, U32.and(i, U32.sub(n, 1)))

def Array.get(-T: Data, a: Array<T>, +i: U32) -> Array<T> & T: Array.get.at(T, i, Array.size(T, a))

def Array.to_list.go(~T: Type, a: Array<T>, acc: List<T>) -> List<T>: match a: case ALeaf{x}: x <> acc case ANode{xs, ys}: Array.to_list.go(~T, xs, Array.to_list.go(~T, ys, acc))

def Array.to_list(~T: Type, a: Array<T>) -> List<T>: Array.to_list.go(~T, a, Nil{})

def Array.map.go( ~T: Data, ~U: Data, ~f: T -> U, +n: Nat, +i: U32, b: Array<U>, r: Array<T> & T ) -> Array<U>: match n: case 0n: b case 1n+p: (a, x) = r +j = U32.inc(i) Array.map.go(~T, ~U, ~f, p, j, Array.set(U, b, i, f(x)), Array.get(T, a, j))

def Array.map.at( ~T: Data, ~U: Data, ~f: T -> U, v: U, an: Array<T> & U32 ) -> Array<U>: (a, +n) = an Array.map.go(~T, ~U, ~f, U32.to_nat(U32.sub(n, 1)), 1, Array.new(U, U32.log2(n), v), Array.get(T, a, 1))

def Array.map.seed(~T: Data, ~U: Data, ~f: T -> U, r: Array<T> & T) -> Array<U>: (a, x) = r Array.map.at(~T, ~U, ~f, f(x), Array.size(T, a))

def Array.map(~T: Data, ~U: Data, ~f: T -> U, a: Array<T>) -> Array<U>: Array.map.seed(~T, ~U, ~f, Array.get(T, a, 0))
```

Map

```
def Map.bit.u(x: U32, k: Nat) -> Bool: U32.is_ne(U32.and(U32.shrn(x, k), 1), 0)
```

```
def Map.bit.chr(c: Char, off: Nat) -> Char & Bool: match c off: case Chr{x} 0n: (Chr{x}, True{}) case Chr{+x} 1n+b: (Chr{x}, Map.bit.u(x, Nat.sub(31n, b)))

def Map.bit.go.chr(t: String, r: Char & Bool) -> String & Bool: (c2, b) = r (SCon{c2, t}, b)

def Map.bit.go.rec(c: Char, r: String & Bool) -> String & Bool: (t2, b) = r (SCon{c, t2}, b)

def Map.bit.go(key: String, ci: Nat, off: Nat) -> String & Bool: match key: case SNil{}: (SNil{}, False{}) case SCon{c, t}: match ci: case 0n: Map.bit.go.chr(t, Map.bit.chr(c, off)) case 1n+j: Map.bit.go.rec(c, Map.bit.go(t, j, off))

def Map.bit.at(key: String, co: Nat & Nat) -> String & Bool: (ci, off) = co Map.bit.go(key, ci, off)

def Map.bit(key: String, pos: Nat) -> String & Bool: Map.bit.at(key, Nat.divmod(pos, 33n))

law Map.msb.u: for n: Nat for +x: U32 Nat

def Map.msb.u.if(p: Nat, x2: U32, z: Bool) -> Nat: match z: case True{}: 0n case False{}: Nat.add(1n, Map.msb.u(p, U32.shr(x2)))

def Map.msb.u(n, x): match n: case 0n: 0n case 1n+p: Map.msb.u.if(p, x, U32.is_zero(x))

def Map.diff.chr(x: U32) -> Nat: Nat.sub(33n, Map.msb.u(32n, x))

def Map.diff.step(x: Char, y: Char) -> Nat & Bool: match x y: case Chr{+cx} Chr{+cy}: (Map.diff.chr(U32.xor(cx, cy)), U32.is_eq(cx, cy))

law Map.diff: for a: String for b: String Nat

def Map.diff.fin(xt: String, yt: String, rc: Nat & Bool) -> Nat: (r, c) = rc match c: case True{}: Nat.add(33n, Map.diff(xt, yt)) case False{}: r

def Map.diff(a, b): match a b: case SNil{} SNil{}: 0n case SNil{} SCon{h, t}: 0n case SCon{h, t} SNil{}: 0n case SCon{x, xt} SCon{y, yt}: Map.diff.fin(xt, yt, Map.diff.step(x, y))

def Map.new(a, -V: Kind(a)) -> Map<a, V>: MTip{}

law Map.put: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String for x : V Map<a, V>

def Map.put.bit( a, -V: Kind(a), x: V, p2: Nat, lo: Map<a, V>, hi: Map<a, V>, kb: String & Bool ) -> Map<a, V>: (key2, b) = kb match b: case False{}: MNode{p2, Map.put(a, V, lo, key2, x), hi} case True{}: MNode{p2, lo, Map.put(a, V, hi, key2, x)}

def Map.put(a, V, m, key, x): match m: case MTip{}: MLeaf{key, x} case MLeaf{k, v}: MLeaf{k, x} case MNode{+pos, lo, hi}: Map.put.bit(a, V, x, pos, lo, hi, Map.bit(key, pos))

def Map.ins.splice.bit( a, -V: Kind(a), x: V, rest: Map<a, V>, pb: Nat, kb: String & Bool ) -> Map<a, V>: (key2, b) = kb match b: case False{}: MNode{pb, MLeaf{key2, x}, rest} case True{}: MNode{pb, rest, MLeaf{key2, x}}

def Map.ins.splice( a, -V: Kind(a), +p: Nat, key: String, x: V, rest: Map<a, V> ) -> Map<a, V>: Map.ins.splice.bit(a, V, x, rest, p, Map.bit(key, p))

law Map.ins: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String for x : V for +p : Nat Map<a, V>

def Map.ins.deep( a, -V: Kind(a), x: V, lo: Map<a, V>, hi: Map<a, V>, pb: Nat, qb: Nat, kb: String & Bool ) -> Map<a, V>: (key2, b) = kb match b: case False{}: MNode{qb, Map.ins(a, V, lo, key2, x, pb), hi} case True{}: MNode{qb, lo, Map.ins(a, V, hi, key2, x, pb)}

def Map.ins.if( a, -V: Kind(a), key: String, x: V, lo: Map<a, V>, hi: Map<a, V>, +p2: Nat, pb: Nat, t: Bool ) -> Map<a, V>: match t: case False{}: Map.ins.splice(a, V, pb, key, x, MNode{p2, lo, hi}) case True{}: Map.ins.deep(a, V, x, lo, hi, pb, p2, Map.bit(key, p2))

def Map.ins(a, V, m, key, x, p): match m: case MTip{}: MLeaf{key, x} case MLeaf{k, v}: Map.ins.splice(a, V, p, key, x, MLeaf{k, v}) case MNode{+pos, lo, hi}: Map.ins.if(a, V, key, x, lo, hi, pos, p, Nat.is_lt(pos, p))

def Map.lo( a, -V: Kind(a), -R: Type, p2: Nat, hi: Map<a, V>, r0: Map<a, V> & R ) -> Map<a, V> & R: (lo2, r) = r0 (MNode{p2, lo2, hi}, r)

def Map.hi( a, -V: Kind(a), -R: Type, p2: Nat, lo: Map<a, V>, r0: Map<a, V> & R ) -> Map<a, V> & R: (hi2, r) = r0 (MNode{p2, lo, hi2}, r)

law Map.seek: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String Map<a, V> & String & Maybe<&2, String>
```

```
def Map.seek.bit( a, -V: Kind(a), lo: Map<a, V>, hi: Map<a, V>, p2: Nat, kb: String & Bool ) -> Map<a, V> & String & Maybe<&2, String>: (key2, b) = kb match b: case False{}: Map.lo(a, V, String & Maybe<&2, String>, p2, hi, Map.seek(a, V, lo, key2)) case True{}: Map.hi(a, V, String & Maybe<&2, String>, p2, lo, Map.seek(a, V, hi, key2))

def Map.seek(a, V, m, key): match m: case MTip{}: (MTip{}, key, None{}) case MLeaf{+k, v}: (MLeaf{k, v}, key, Some{k}) case MNode{+pos, lo, hi}: Map.seek.bit(a, V, lo, hi, pos, Map.bit(key, pos))

def Map.set.fin.go( a, -V: Kind(a), m: Map<a, V>, key: String, x: V, r: (String & String) & Cmp ) -> Map<a, V>: ((keyb2, k2), c) = r match c: case LT{}: Map.ins(a, V, m, key, x, Map.diff(keyb2, k2)) case EQ{}: Map.put(a, V, m, key, x) case GT{}: Map.ins(a, V, m, key, x, Map.diff(keyb2, k2))

def Map.set.fin( a, -V: Kind(a), m: Map<a, V>, key: String, x: V, keyb: String, k: String ) -> Map<a, V>: Map.set.fin.go(a, V, m, key, x, String.cmp(keyb, k))

def Map.set.go( a, -V: Kind(a), x: V, r: Map<a, V> & String & Maybe<&2, String> ) -> Map<a, V>: (m2, key2, found) = r match found: case None{}: MLeaf{key2, x} case Some{k}: +ka = {key2 : String} Map.set.fin(a, V, m2, ka, x, ka, k)

def Map.set(a, -V: Kind(a), m: Map<a, V>, key: String, x: V) -> Map<a, V>: Map.set.go(a, V, x, Map.seek(a, V, m, key))

def Map.has.leaf(a, -V: Kind(a), v: V, r: (String & String) & Cmp) -> Map<a, V> & Bool: ((key2, k2), c) = r (MLeaf{k2, v}, Cmp.is_eq(c))

law Map.has: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String Map<a, V> & Bool

def Map.has.bit( a, -V: Kind(a), lo: Map<a, V>, hi: Map<a, V>, p2: Nat, kb: String & Bool ) -> Map<a, V> & Bool: (key2, b) = kb match b: case False{}: Map.lo(a, V, Bool, p2, hi, Map.has(a, V, lo, key2)) case True{}: Map.hi(a, V, Bool, p2, lo, Map.has(a, V, hi, key2))

def Map.has(a, V, m, key): match m: case MTip{}: (MTip{}, False{}) case MLeaf{k, v}: Map.has.leaf(a, V, v, String.cmp(key, k)) case MNode{+pos, lo, hi}: Map.has.bit(a, V, lo, hi, pos, Map.bit(key, pos))

def Map.get.leaf(-V: Data, d: V, +v: V, r: (String & String) & Cmp) -> Map<&2, V> & V: ((key2, k2), c) = r match c: case LT{}: (MLeaf{k2, v}, d) case EQ{}: (MLeaf{k2, v}, v) case GT{}: (MLeaf{k2, v}, d)

law Map.get: for -V : Data for d : V for m : Map<&2, V> for key : String Map<&2, V> & V

def Map.get.bit( -V: Data, d: V, lo: Map<&2, V>, hi: Map<&2, V>, p2: Nat, kb: String & Bool ) -> Map<&2, V> & V: (key2, b) = kb match b: case False{}: Map.lo(&2, V, V, p2, hi, Map.get(V, d, lo, key2)) case True{}: Map.hi(&2, V, V, p2, lo, Map.get(V, d, hi, key2))

def Map.get(V, d, m, key): match m: case MTip{}: (MTip{}, d) case MLeaf{k, v}: Map.get.leaf(V, d, v, String.cmp(key, k)) case MNode{+pos, lo, hi}: Map.get.bit(V, d, lo, hi, pos, Map.bit(key, pos))

def Map.pop.lo( a, -V: Kind(a), pos: Nat, hi: Map<a, V>, r0: Map<a, V> & Maybe<a, V> ) -> Map<a, V> & Maybe<a, V>: (lo, r) = r0 match lo: case MTip{}: (hi, r) case lo2: (MNode{pos, lo2, hi}, r)

def Map.pop.hi( a, -V: Kind(a), pos: Nat, lo: Map<a, V>, r0: Map<a, V> & Maybe<a, V> ) -> Map<a, V> & Maybe<a, V>: (hi, r) = r0 match hi: case MTip{}: (lo, r) case hi2: (MNode{pos, lo, hi2}, r)

def Map.pop.leaf(a, -V: Kind(a), v: V, r: (String & String) & Cmp) -> Map<a, V> & Maybe<a, V>: ((key2, k2), c) = r match c: case LT{}: (MLeaf{k2, v}, None{}) case EQ{}: (MTip{}, Some{v}) case GT{}: (MLeaf{k2, v}, None{})

law Map.pop: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String Map<a, V> & Maybe<a, V>

def Map.pop.bit( a, -V: Kind(a), lo: Map<a, V>, hi: Map<a, V>, p2: Nat, kb: String & Bool ) -> Map<a, V> & Maybe<a, V>: (key2, b) = kb match b: case False{}: Map.pop.lo(a, V, p2, hi, Map.pop(a, V, lo, key2)) case True{}: Map.pop.hi(a, V, p2, lo, Map.pop(a, V, hi, key2))

def Map.pop(a, V, m, key): match m: case MTip{}: (MTip{}, None{}) case MLeaf{k, v}: Map.pop.leaf(a, V, v, String.cmp(key, k)) case MNode{+pos, lo, hi}: Map.pop.bit(a, V, lo, hi, pos, Map.bit(key, pos))

def Map.del.fin(a, -V: Kind(a), r: Map<a, V> & Maybe<a, V>) -> Map<a, V>: (m2, x) = r m2

def Map.del(a, -V: Kind(a), m: Map<a, V>, key: String) -> Map<a, V>: Map.del.fin(a, V, Map.pop(a, V, m, key))

def Map.to_list.go( a, -V: Kind(a), m: Map<a, V>, acc: List<a, Sigma<&2, a, String, _ => V>> ) -> List<a, Sigma<&2, a, String, _ => V>>: match m: case MTip{}: acc case MLeaf{k, v}: (k, v) <> acc case MNode{pos, lo, hi}: Map.to_list.go(a, V, lo, Map.to_list.go(a, V, hi, acc))

def Map.to_list(a, -V: Kind(a), m: Map<a, V>) -> List<a, Sigma<&2, a, String, _ => V>>: Map.to_list.go(a, V, m, Nil{})
```

```
def Map.keys.go(a, -V: Kind(a), m: Map<a, V>, acc: List<&2, String>) -> List<&2, String>: match m: case MTip{}: acc case MLeaf{k, v}: k <> acc case MNode{pos, lo, hi}: Map.keys.go(a, V, lo, Map.keys.go(a, V, hi, acc))

def Map.keys(a, -V: Kind(a), m: Map<a, V>) -> List<&2, String>: Map.keys.go(a, V, m, Nil{})

def Map.from_list.go( a, -V: Kind(a), kvs: List<a, Sigma<&2, a, String, _ => V>>, m: Map<a, V> ) -> Map<a, V>: match kvs: case Nil{}: m case (k, v) <> t: Map.from_list.go(a, V, t, Map.set(a, V, m, k, v))

def Map.from_list(a, -V: Kind(a), kvs: List<a, Sigma<&2, a, String, _ => V>>) -> Map<a, V>: Map.from_list.go(a, V, kvs, MTip{})

def Map.union(a, -V: Kind(a), m: Map<a, V>, n: Map<a, V>) -> Map<a, V>: Map.from_list.go(a, V, Map.to_list(a, V, n), m)

def Map.size(a, -V: Kind(a), m: Map<a, V>) -> Nat: match m: case MTip{}: 0n case MLeaf{k, v}: 1n case MNode{pos, lo, hi}: Nat.add(Map.size(a, V, lo), Map.size(a, V, hi))

def Map.values.go(a, -V: Kind(a), m: Map<a, V>, acc: List<a, V>) -> List<a, V>: match m: case MTip{}: acc case MLeaf{k, v}: v <> acc case MNode{pos, lo, hi}: Map.values.go(a, V, lo, Map.values.go(a, V, hi, acc))

def Map.values(a, -V: Kind(a), m: Map<a, V>) -> List<a, V>: Map.values.go(a, V, m, Nil{})
```

Set

```
---

def Set() -> Data: Map<&2, Unit>

def Set.new() -> Set(): Map.new(&2, Unit)

def Set.add(s: Set(), key: String) -> Set(): Map.set(&2, Unit, s, key, Unit{})

def Set.has(s: Set(), key: String) -> Set() & Bool: Map.has(&2, Unit, s, key)

def Set.del(s: Set(), key: String) -> Set(): Map.del(&2, Unit, s, key)

def Set.size(s: Set()) -> Nat: Map.size(&2, Unit, s)

def Set.to_list(s: Set()) -> List<&2, String>: Map.keys(&2, Unit, s)

def Set.from_list.go(keys: List<&2, String>, s: Set()) -> Set(): match keys: case Nil{}: s case h <> t: Set.from_list.go(t, Set.add(s, h))

def Set.from_list(keys: List<&2, String>) -> Set(): Set.from_list.go(keys, Set.new())
```

Image

```
-----

def Image.sink(img: Image) -> Unit: Unit{}

def Image.drop.join(a: Unit, b: Unit, c: Unit, d: Unit) -> Unit: match a b c d: case Unit{} Unit{} Unit{} Unit{}: Unit{}
```

Forks k levels, then a dead parameter sinks the rest in one loop.

```
def Image.free(+k: Nat, img: Image) -> Unit: match k: case 0n: Image.sink(img) case 1n+e: match img: case Pix{c}: Unit{} case Qua{tl, tr, bl, br}: a b c d = Image.free(e, tl) Image.free(e, tr) Image.free(e, bl) Image.free(e, br) Image.drop.join(a, b, c, d)

def Image.drop(img: Image) -> Unit: Image.free(7n, img)
```

App

```
---

def App.next( -S: Type, window: Window, rest: Window -> S -> IO(Unit), next: Maybe<S> ) -> IO(Unit): match next: case None{}: Window.close(window) case Some{state}: rest(window, state)

def App.turn( -S: Type, shown: Window & Image & List<Event>, tick: List<Event> -> S -> IO(Maybe<S>), state: S, rest: Window -> S -> IO(Unit) ) -> IO(Unit): (window, image, events) = shown do IO<Unit>: Unit <- IO.pure(Unit, Image.drop!(image)) IO.bind(Maybe<S>, Unit, tick(events, state), App.next(S, window, rest))
```

def App.draw(-S: Type, window: Window, drawn: S & Image, tick: List<Event> -> S -> IO(Maybe<S>), rest: Window -> S -> IO(Unit)) -> IO(Unit): (state, image) = drawn do IO<Unit>: shown : Window & Image & List<Event> <- Window.frame(window, image) App.turn(S, shown, tick, state, rest)

def App.step(-S: Type, app: App<S>, window: Window, state: S, rest: Window -> S -> IO(Unit)) -> IO(Unit): App{view, tick} = app App.draw(S, window, view(state), tick, rest)

def App.loop(~S: Type, ~app: App<S>, fuel: Nat, window: Window, state: S) -> IO(Unit): match fuel: case 0n: Window.close(window) case 1n+f: App.step(S, app, window, state, w => s => App.loop(~S, ~app, f, w, s))

def App.run(~S: Type, ~app: App<S>, title: String, width: U32, height: U32, state: S) -> IO(Unit): do IO<Unit>: window : Window <- IO.try(Window, Window.open(title, width, height)) App.loop(~S, ~app, U32.to_nat(4294967295), window, state)

def App.more(-S: Type, rest: S -> IO(Maybe<S>), next: Maybe<S>) -> IO(Maybe<S>): match next: case None{}: IO.pure(Maybe<S>, None{}) case Some{state}: rest(state)

def App.fold(-S: Type, app: App<S>, events: List<Event>, state: S, rest: S -> IO(Maybe<S>)) -> IO(Maybe<S>): App{view, tick} = app IO.bind(Maybe<S>, Maybe<S>, tick(events, state), App.more(S, rest))

def App.play(~S: Type, ~app: App<S>, frames: List<List<Event>>, state: S) -> IO(Maybe<S>): match frames: case Nil{}: IO.pure(Maybe<S>, Some{state}) case events <> rest: App.fold(S, app, events, state, s => App.play(~S, ~app, rest, s))

F.13 · bend-runtime/knowledge/02-guides-cli/raw/BASE-TYPES-DUMP.md

type Empty is Data:

type Unit is Data: Unit{}

type Bool is Data: False{} True{}

type Cmp is Data: LT{} EQ{} GT{}

type Either<a, b, -A: Kind(a), -B: Kind(b)> is Kind(a <&> b): Inl{value: A} Inr{value: B}

type Sigma<a, b, -A: Kind(a), -B: @-x: A -> Kind(b)> is Kind(a <&> b): Tuple{fst: A, snd: B(fst)}

type Nat is Data: Zero{} Succ{pred: Nat}

type Maybe<a, -A: Kind(a)> is Kind(a): None{} Some{value: A}

type Result<a, b, -E: Kind(a), -A: Kind(b)> is Kind(a <&> b): Fail{error: E} Done{value: A}

type List<a, -A: Kind(a)> is Kind(a): Nil{} Con{head: A, tail: List<a, A>}

law Word: for n: Nat Data

type Word.Nil is Data: WNil{}

type Word.Con<-p: Nat> is Data: WCon{head: Bool, tail: Word(p)}

type U32 is Data: U32{data: Word(32n)}

type F32 is Data: F32{data: Word(32n)}

type Char is Data: Chr{code: U32}

type String is Data: SNil{} SCon{head: Char, tail: String}

type Array<-T: Type> is Type: ALeaf{value: T} ANode{xs: Array<T>, ys: Array<T>}

type Image is Data: Pix{color: U32} Qua{tl: Image, tr: Image, bl: Image, br: Image}

type Event is Data: Key{code: U32, down: Bool} Mouse{x: U32, y: U32, button: U32, down: Bool} Move{x: U32, y: U32} Close{}

type Map<a, -V: Kind(a)> is Kind(a): MTip{} MLeaf{key: String, val: V} MNode{pos: Nat, lo: Map<a, V>, hi: Map<a, V>}

law File: Type

law Socket: Type

law Listener: Type

law Window: Type

law Audio: Type

law Chan: for -A: Type Data

type IO.OP<-R: Type> is Type: Emit{value: R} Halt{code: U32, message: String}

law Pair: for -A: Type for -B: Type Type

law IO: for -A: Type Type

type App<-S: Type> is Type: App{view: S -> Pair(S, Image), tick: List<Event> -> S -> IO(Maybe<S>)}

F.14 · bend-runtime/knowledge/02-guides-cli/raw/CLI-HELP-DUMP.txt

Bend 2.0.25: check, run, build and publish Bend programs.

usage: bend <file.bend> [args] check the file, then run main with args bend <file.bend> -o <out> build a binary; <out>.c emits C, <out>.js JS bend <file.bend> --check-only check the file and its imports; run nothing bend <file.bend> --publish publish the file and its imports to the hub bend <page.html> -o <dir> bundle a page that imports .bend files bend base [--types|<name>] print Base, its types, or a name and subnames bend guide print the Bend guide bend update install the latest bend (curl | sh, shown first) bend version print the version

Read the guide (`bend guide`) before writing Bend code.

F.15 · bend-runtime/knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md

Effects in Bend

An AI wrote this text from the runtime's source (`bend2/comp.ts` and `bend2/effs/`). A human will revise it later. Report anything wrong on GitHub.

AN EFFECT DEF

An effect is a def of type `IO(R)` whose body is two imports:

```
def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"
```

The host function is named after the def: lowercased, dots to underscores (`clock_now`). The `.c` file serves `bend x.bend -o x` ; the `.js` file serves `bend x.bend -o x.js` and `bend x.bend` . Every effect in Base is built this way: `bend2/effs/*.c` and `*.js` are the reference.

THE C SIDE

The compiler splices your `.c` into the program's C source after the runtime, so every runtime symbol is in scope. Define the effect and register it in a constructor:

```
Term clock_now_run(Env e, Term* f, IoWork* w);

static void __attribute__((constructor)) clock_now_use(void) {
  io_eff(CID_CLOCK_NOW, clock_now_run, 0);
}
```

`CID_X` is the def's name uppercased, dots to underscores. `f` holds the def's arguments in order: a `U32` is the word (`(u32)f[0]`), a `String` is taken with `io_cstr(e, f[0], &len)` (a `malloc` ed copy you free), a handle with `io_hand_v(f[0])` . The last argument of `io_eff` is the need: `0` runs the effect at once; `IO_READ` parks it until the handle in `f[0]` is readable; `IO_TIME` parks it for `f[0]` milliseconds. Then the loop calls the effect.

The effect returns a Term: a `U32` is `(Term)n` , `Unit` is `term_pak(CID_UNIT, 0)` , a `String` is `io_str(e, p, n)` , a two-field constructor is `io_node(e, CID_K, a, b, 0)` . A `Result` is `io_done(e, v)` or `io_fail(e, code, text)` (`text` NULL prints `strerror(code)`). `io_sys_end(w, n)` stores `errno` in `w->code` when a call fails.

A handle is a descriptor or pointer packed in one word by `io_hand(v)`. Its type must be one of Base's handle laws (`File`, `Socket`, ...): a user handle type is a WONTFIX entry for now, see `WONTFIX.txt`. An effect on a handle hands it back beside its result: `io_tup(e, io_hand(h), r)`, also on failure.

Blocking work leaves the loop in two ways. `io_work(w, call, pack)` runs `call(w)` on a helper thread, then `pack(e, w)` on the loop; `pack`'s value is the answer. `call` has no `Env`: it may only touch `w`. `io_wait_on(w, fd, POLLIN, more)` parks until `fd` is ready (`POLLIN` or `POLLOUT`), then runs `more(e, w)` on the loop; `more` may call `io_wait_on` again. Both return `IO_PARK`, which the effect returns.

`w` is the effect's scratch space: `hand`, `made`, `word`, `size`, `data`, `text`, `code`. The runtime owns `w` and `f`. The effect owns `w->data`: allocate it in the run function and free it in `pack`. In `bend2/effs/`, `file_read.c` is the pattern for `io_work` and `tcp_recv.c` for `io_wait_on`.

THE JS SIDE

The `.js` file is a plain script, wrapped in a closure; the compiler finds the function by name. It takes the def's arguments as JS values: a `U32` is a number, a `Nat` a `BigInt`, a `String` a string, a constructor `{$: "Name", field: value}`, a handle its host value (a descriptor). It returns the answer the same way: `io_done(v)`, `io_fail(code)`, `io_tup(handle, result)`, `{$: "Unit"}`.

A need is a second function, `clock_now_need`, that returns `{read: true}` or `{time: true}`. A blocking effect takes one more argument, `k`, and parks with `io_park_on(fd, out, k, more)`: it returns `undefined`, and the loop calls `more()` when `fd` is ready; `more` answers the value, or `undefined` to park again. `io_sys()` is `libc` through `bun:ffi` (`read`, `recv`, `poll`, `errno`); `tcp_recv.js` shows the full shape.

A COMPLETE EXAMPLE

`main.bend`:

```
import Base

def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"

def main() -> IO(Unit):
  do IO<Unit>:
    t : U32 <- Clock.now()
    IO.print("ms since boot: " ++ U32.show(t))
```

`clock.c`:

```
Term clock_now_run(Env e, Term* f, IoWork* w) {
  return (Term)(uint32_t)(io_tick() / 1000000ull);
}

static void __attribute__((constructor)) clock_now_use(void) {
  io_eff(CID_CLOCK_NOW, clock_now_run, 0);
}
```

`clock.js`:

```
function clock_now() {
  return Math.floor(performance.now()) >>> 0;
}
```

`bend main.bend -o main && ./main` prints a line; so do `bend main.bend -o main.js && bun main.js` and `bend main.bend`. The output name must not be one of the effect's files.

COMPATIBILITY

The C side tracks the exact compiler version: the names above are the runtime's internals, and a release may rename any of them. There is no ABI promise. Rebuild your effects with every update.

F.16 · bend-runtime/knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md

Bend

Bend is a new programming language that combines Lean-like formal proofs with C-like speeds and CUDA-like parallelism. It gives humans an ambiguity-free language to communicate their intents to AIs, a compiler capable of mechanically checking that the AI implemented these intents to unquestionable mathematical correctness, and a compiler that runs that code fast on CPUs and GPUs.

HELLO, WORLD!

Bend's syntax is Python-shaped, but its semantics are closer to Haskell / Lean, while being resource-aware like Rust (if less annoyingly). A hello world is just a typed definition returning an IO block:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

To run it, install Bend (`curl -fsSL https://bend-lang.com/install.sh | sh`) and type `bend hello.bend` .

Bend is a *pure language*, with effects denoted via a Haskell-inspired IO Monad (https://wiki.haskell.org/Introduction_to_IO). It comes with a list of built-in effects for files, networking, audio, graphics, input, and more, and the user can extend them with foreign C and JS imports; more on that later.

CORE FEATURES

TYPES AND FUNCTIONS

A typical Bend program is a set of datatypes and functions over them:

```
import Base

type Shape is Data:
  Circle{r: U32}
  Square{s: U32}

def area(x: Shape) -> U32:
  match x:
    case Circle{+r}:
      (3 * r * r : U32)
    case Square{+s}:
      (s * s : U32)

def main() -> U32:
  area(Square{5})
```

Bend, by default, is *affine*, meaning variables must be used, at most, once.

Here, `is Data` declares that `Shape` may be copied, while `is Type` keeps it non-copiable. The `+` annotation before a variable name allows using it more than once, if the variable is Data-kinded.

Bend does almost no inference, meaning it requires more annotations than similar languages. This is what allows Bend's checker to be significantly faster than other provers, and its error messages more precise, at the expense of programs and proofs being more verbose. When the checker can't decide the type of an expression, just annotate it, as in `{3 : U32}` .

CLOSURES

Functions are values and can be stored, passed, and returned from other functions.

```
import Base

def adder(k: U32) -> U32 -> U32:
  x => (x + k : U32)

def main() -> U32:
  add2 = adder(2)
  add5 = adder(5)
  add5(add2(1))
```

A closure is affine: it can be called at most once, even when everything it captures is `Data`. Only top-level definitions can be called freely. Partial applications like `U32.add(2)` are closures too.

RECURSION AND TERMINATION

Bend uses recursion to repeat work. Tail calls compile to loops.

```
import Base

def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}:
      acc
    case Con{h, t}:
      sum(t, (acc + h : U32))

def main() -> U32:
  sum([1, 2, 3, 4], 0)
```

Here, `t` has one fewer element than `xs`, so `sum` eventually reaches the empty list. Bend verifies termination by requiring recursive calls to use smaller parts of their inputs, obtained through pattern matching. The check reads the arguments of a recursive call from left to right: each must be passed unchanged until one is a smaller part of its parameter, and the ones after it are free. So, put the parameter that shrinks first. Also, is no `if` syntax yet. Use `match` on `True{}` and `False{}` instead.

Termination is mandatory and mutual recursion is not allowed. Both restrictions keep Bend's proofs sound, as a function that never returns could otherwise prove anything. A loop bounded by the outside world, like a server's, counts down a `Nat` fuel argument instead, and two mutually recursive functions become one def with an extra argument selecting which to run. A `def` marked `@unsafe` recurses freely, but falls outside Bend's proof guarantees.

A `match` inspects a parameter or a variable bound by a pattern, never a computed value: `match sum(xs, 0):` is rejected. Scrutinees follow binder order, and a `let` may not precede a `match` on a parameter. To match on a computed value, pass it to a helper that matches on its parameter.

PARALLELISM

Bend's parallelism primitive is the parallel call notation:

```
import Base

def pow2(+n: Nat) -> U32:
  match n:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p) # parallel call
      (a + b : U32)

def main() -> IO(Unit):
  IO.print(U32.show(pow2!(20n))) # `!` runs on GPU
```

A parallel call promises the compiler two things:

1. The calls are independent.
2. They run in roughly the same time.

Since Bend is pure and affine, the first point always holds. The second is yours to keep: if one call finishes before the other, the speedup will be sub-ideal. Bend's current scheduler is a contention-free, binary fork-join machine: every task is handed to a core exactly once and never moved afterwards. That makes it fast and GPU-friendly, but you must keep the workload balanced.

A `!` after a function name marks a parallel call: `pow2!(20n)` hands that call, and every parallel call inside it, to the GPU. When compiled to a native executable, `pow2(20n)` runs in parallel on the CPU, while `pow2!(20n)` runs on the GPU. The heap is fully unified, so, if your chip has unified memory (as in Apple M-series processors), moving data from the CPU to the GPU is a zero-cost operation. The GPU shines on uniform numeric work like mandelbrot or nbody; divergent work like n-queens stays faster on the CPU. A machine without a GPU runs `!` on the CPU (still in parallel). What the lanes share also sets the speed: a `+` value read by every lane costs an atomic per read. Read `bend guide shaders` before you write a parallel app.

The JavaScript target ignores all that and just runs sequentially.

ARRAYS

Arrays give Bend in-place mutation without giving up purity:

```
import Base

def main() -> Array<U32> & U32:
  a = [0 : U32*8n] # new array with 8 copies of 0
  a[5] <- 42       # performs an in-place rewrite
  a[5]             # reads index 5
```

An `Array<T>` is a `Type`, so it has exactly one owner at all times. That is what lets `a[5] <- 42` overwrite the slot and hand back the same array, with no copy. A read hands the array back next to the element for the same reason: if it returned only the element, the array would be gone. Indexes wrap around. A write followed by another statement re-binds its array: `a[5] <- 42` on its own line is `a = a[5] <- 42`. As the last statement it is the written array. The slot count after `*` is a power of two; `[0 : U32^3n]` names the depth instead.

The `a[i]` sugar assumes `Array<U32>`. For other element types, call `Array.get` (`Data` elements; else `Array.swap`) and `Array.set` directly, and `Array.clone` when you need two copies. Read Bend's Base for reference. This will be generalized soon!

QUANTITIES

A quantity says how many times a variable may be used.

```
import Base

# -A: erased (gone at runtime)
#  n: affine (used at most once)
# +x: reusable (requires A to be Data)
def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n:
      Nil{}
    case 1n+p:
      x <> replicate(A, p, x)

def main() -> List<U32>:
  replicate(U32, 3n, 7)
```

Erased variables can only appear in types and proofs: the checker sees them, the compiler deletes them. Affine variables are the default, and dropping one is always free. Reusable variables require `Data`: functions, arrays and IO handles are `Type`, so they can never be copied. Note that `main` marks nothing: you write `+` where you need the copies, and `replicate` pays for them with a reference count at runtime. Matching a `+` value hands out `+` fields; on a plain one, write `+r` in the pattern to make a field reusable.

KINDS

Every type has a kind, which caps how many times its values may be used.

```
import Base

#  a: a quantity (&0, &1 or &2)
# -A: a type whose values may be used a times
def length(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{h, t}:
      1n+length(a, A, t)

def main() -> Nat:
  length(&2, U32, [1, 2, 3])
```

`Type` is short for `Kind(&1)` and `Data` for `Kind(&2)`, so a `Kind(a)` parameter accepts both: `length(&1, U32 -> U32, fs)` counts a list of closures just as well. A bare `a` in a parameter list is short for `-a: Quant`. Base declares `type List<a, -A: Kind(a)> is Kind(a)`, making a list exactly as reusable as its elements: `List<U32>` is short for `List<&1, U32>`, and `+List<U32>` for `List<&2, U32>`. A type holding two element types combines their quantities with `a <&> b`, the smaller of the two.

TEMPLATES

A template receives its argument as syntax and inlines it at compile time.

```
import Base

# ~f: substituted at compile time, not passed at runtime
def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

Template parameters come first in the parameter list, and a `~` argument must be closed: it may mention top-level defs, but no local variable of the caller. Each distinct set of `~` arguments compiles to its own copy of `twice`, so `f` costs nothing at runtime and, unlike a closure, may be called as many times as you like. A template may call only templates declared above it. This is how `List.map` is written in Base.

LAWS AND PROOFS

A law states a fact that must hold. It must be proven inside a paired def.

```
import Base

# LAW: "for every x, x plus 0 equals x"
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

# PROOF: case analysis:
# - base case: reflexivity
# - step case: induction, rewrite, reflexivity
def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> {Nat.add(2n, 0n) == 2n : Nat}:
  add_zero(2n)
```

Laws are a critical feature in Bend, as they provide an ambiguity-free language on which humans can state precise specs for AI's to implement. That is, instead of writing a natural language prompt such as "implement a function that sorts a list", users can write precise laws like "implement a function F such that, for every list of numbers, `F(list)` returns the same numbers in ascending order". Models are then guaranteed to respond with bug-free code, since Bend will demand that they provide an actual proof.

> We envision that "law-driven development" will eventually become the way humans use AI to write and maintain large codebases, as it is the perfect middle point between having to code everything manually (laborious) and letting AI do it all via prompts without auditing a line of code (error/ambiguity-prone, unsecure).

By convention, a project keeps its laws in two files at its root. `LAWS.bend` imports the code and states the laws, each an open claim: the human writes it, the AI does not touch it. `PROOF.bend` imports `LAWS.bend` and proves each law with a def of the same name (`law sorted` is proven by `def Laws.sorted`): the AI writes it, along with the code. `bend PROOF.bend` is the gate: it fails while any law is open or false, and prints "All terms check." once every law holds. bend refuses a `PROOF.bend` that sits beside a `LAWS.bend` without importing it.

Bend has no tactics: a proposition is a type, and a proof is a def of that type. `{a == b : T}` is an equality; `{==}` proves it when both sides compute to the same term. Matching refines the goal in each case, a recursive call is the induction hypothesis, and `%e : P` rewrites with `e : {a == b : T}` : `P` is the goal with `_` marking `b`, and the goal becomes `P` with `a` there. `exs y: T` in a law asks for a witness, returned as `(y, proof)`. A failed step prints the expected and observed terms; `?name` prints the goal, `?TODO` leaves it open, and a law with no def is an open claim.

Since types are terms, a def may return a `Type`, like `def IsEven(n: Nat) -> Type:`, which is all dependent types are. A `match e:` with no cases closes a branch where `e : Empty`. To refute a clash like `e : {1n == 0n : Nat}`, rewrite it through a motive `disc(_)`, where `disc` sends `0n` to `Empty` and `1n+p` to `Unit`, and answer `Unit{}`. `{a != b : T}` is `{a == b : T} -> Empty`, and `Equal.sym`, `Equal.trans` and `Equal.cong` are in Base.

IO AND CONCURRENCY

Effects live in the `IO` type and are sequenced with `do` blocks:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1000)             # a step
    return "Hello, " ++ name   # return: wraps a pure value

def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER")) # <-: binds a result
    chan : Chan(String) <- IO.fork(String, greet(name)) # runs concurrently
    IO.print("Waiting...")
    text : String <- IO.join(String, chan)
    IO.print(text)
```

Every bind is annotated, and `x : T = v` binds a pure value in the middle of a block. A fallible effect answers `Result<&1, &1, U32 & String, A>`: `IO.try` unwraps it or exits with the error, and `IO.die` exits with your own. `IO.args` answers the command line, less the runtime's own options (a `--` ends them). A handle (`File`, `Socket`, `Window`) is an affine, opaque value, so every effect on one hands it back beside its result, and no program can forge or reuse one.

A Bend program is a set of computations interleaved by one event loop, as in Node.js: each runs its pure code (in parallel, on every core) up to its next effect, and one that waits on a socket, a sleep or a channel steps aside for the others. `IO.fork` starts a computation and returns the channel its result will arrive on; `IO.join` waits for it. Underneath are `IO.spawn`, `Chan.new`, `Chan.send`, `Chan.recv` and `Chan.close`. The program ends when every computation is done, or reports a deadlock when the remaining ones all wait.

Every effect in Base is a def whose body is `import "./x.js"` plus a `.c` twin, implemented by a host function named after the def, lowercased, dots to underscores. You can add your own effects the same way. Only the event loop runs them, so proofs, termination and the GPU never touch host code. In the other direction, a JS file may `import Game from "./game.bend"` (with `bend2/main.ts` preloaded) and call every non-IO def, with constructors as `{$: "Name", field: value}` and `Nat` as `BigInt`. A value crosses without a copy: an `Array` argument is the caller's own array, updated in place, so copy it first if you keep it.

MONADS

The `do` notation works for any monad, not just IO.

```
import Base

def add_strs(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a) # a None here ends the block with None
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> Maybe<&2, U32>:
  add_strs("40", "2")
```

A `do M<xs.., R>:` block desugars each `x : A <- v` into `M.bind(xs.., A, R, v, x => ..)` and each `return e` into `M.pure(xs.., R, e)`, so any type with those two defs works: `IO`, `Maybe`, `Result`, or your own. The leading arguments (here, the `&2` quantity) are passed along to both.

APPS

Graphics in Bend are pure: a frame is an `Image`; `App` maps states to images.

```
import Base

# tick: folds a frame's events into the next state; None quits the app
def tick(events: List<Event>, color: U32) -> IO(Maybe<U32>):
  match events:
    case Nil{}:
      IO.pure(Maybe<U32>, Some{color})
    case Con{Close{}, rest}:
      IO.pure(Maybe<U32>, None{})
    case Con{e, rest}:
      tick(rest, (color + 1 : U32))

def main() -> IO(Unit):
  # view: returns the state beside its image; Pix paints the whole frame
  App.run(~U32, ~App{+s => (s, Pix{s}), tick}, "Hello", 256, 256, 0)
```

An `Image` is a quadtree: `Pix{color}` paints a square, and `Qua{tl, tr, bl, br}` splits it in four, so a frame is drawn by recursion like everything else, in parallel if you want. Events are `Key`, `Mouse`, `Move` and `Close`. `App.run` opens a window and calls `view` then `tick` once per frame, until `tick` answers `None`. Since the state is affine, `view` must hand it back next to the image. Underneath are `Window.open`, `Window.frame` and `Window.close`, and `Audio.open`, `Audio.write` and `Audio.close` for sound. See `demos/app_pong_game_2d` for a complete one.

THE BASE LIBRARY

Base is small, and its names follow a scheme, so you can guess most of it:

```
import Base

def main() -> String:
  +a = (6 * 7 : U32)         # sugar for U32.mul(6, 7)
  b  = U32.to_nat(a)         # conversions are T.to_x and T.from_x
  U32.show(a) ++ " = " ++ Nat.show(b)
```

Every def is named `Type.verb`, and the same verbs recur across `Nat`, `U32` and `F32`: `add sub mul div mod` for arithmetic, `and or xor not shl shr` for bits (`U32` only), `cmp` (returning `Cmp`, not on `F32`) and `is_eq is_ne is_lt is_le is_gt is_ge` (returning `Bool`) for comparisons, `show` to `String` and `read` back from it (answering a `Maybe`). Operators and `<`-style comparisons are just sugar for these. Beyond numbers there are `Bool`, `Cmp`, `Maybe`, `Result`, `List`, `Array`, a string-keyed `Map` (`new set get has del keys`; `get` takes a default, and `get` and `has` hand the map back beside their result), `Set` on top of it, the `Equal` lemmas, and the effects. `bend base` prints all of it, `bend base --types` only the types, and `bend base Map` one name and everything under it.

MODULES

A module is a file, and an import gives it a local name:

```
# math.bend
import Base

def square(+x: U32) -> U32:
  (x * x : U32)
```

```
# main.bend
import Base
import ./math.bend as M  # M.x now names every def of math.bend

def main() -> U32:
  M.square(7)
```

The alias is local to the importing file, and dots inside a name are just characters: `U32.show` needs no module. A law left open in one file may be filled in another as `def M.name(..)`, so a proof can ship separately from its claim. `import 0x<hash>/main.bend as P` imports a package by content hash, fetched from the hub and checked against it; `bend main.bend --publish` uploads a file with everything it imports and prints that line.

TOOLING

Bend is a single command:

```
bend file.bend            # check; run main (IO compiled; a value normalized)
bend file.bend -o file    # compile to a native binary (clang 14+; 19+ with `!`)
bend file.bend -o file.c  # emit the C source instead
bend file.bend -o file.js # emit the JS source instead
bend page.html -o dist    # bundle a web page that imports .bend files
./file --threads 8        # run a native binary on 8 CPU threads
./file --gpu off          # run ! calls on the CPU (the GPU is on by default)
./file --gpu 4GB          # cap the GPU's heap at 4GB
```

A `main` that returns `IO` runs compiled; one that returns a value is normalized by the checker (slow for big work) and printed; a file with no `main` just checks. A binary that uses `!` builds its GPU program too, as `file.gpu`, which must stay beside it: on macOS it needs Metal, on Linux CUDA 12 at `/usr/local/cuda`. On Linux a program with a Window needs `libx11-dev`, one with Audio `libasound2-dev`. `bend guide` prints this text, `bend base` prints the Base library (`bend base Map` prints one name and everything under it), and `bend --help` lists the other commands.

SYNTAX REFERENCE

Every form of the language, grouped by where it appears. Operators, literals and brackets are sugar for names in Base.

```
# Top level
import Base                              # the prelude
import ./file.bend as M                  # a module; its defs are M.x
type D<a, -A: Kind(a)> is Kind(a):       # a datatype and its kind
  K{x: A, xs: List<a, A>}                # one constructor per line
def f(x: A, -y: B, +z: C) -> T:          # a def; the body follows
def f(x, y):                             # fills the law named f
def t(~g: A -> B, x: A) -> B:            # a template
law f:                                   # a claim, proven by def f
  for x: A                               # a parameter (also for -x, for +x)
  for y: B where P(y)                    # y is then the pair (y, P(y) proof)
  exs z: C                               # a witness the proof must return
  T                                      # the claim
@unsafe def f(x: A) -> T:                # skips the termination check
def e(x: A) -> IO(B):                    # a foreign effect
  import "./e.c"
  import "./e.js"
```

```
# Types
Type  Data  Kind(q)                      # kinds; Type = Kind(&1), Data = Kind(&2)
Quant  &0  &1  &2  a <&> b               # quantities and their minimum
A -> B  @x:A -> B  @-x:A -> B            # functions: plain, dependent, erased
A & B  &x:A -> B  A | B                  # pairs, dependent pairs, sums
D<A>  +D<A>  D<&2, A>                    # a datatype; + makes it reusable
{a == b : T}  {a != b : T}               # equality and its negation
```

```
# Terms
42  1.5  3n  'c'  "s"                  # U32, F32, Nat, Char, String
[a, b]  h <> t  (a, b)                 # a list, a cons, a tuple
K{a, b}  x => e  +x => e               # a constructor, a lambda
f(a, b)  f!(a)  t(~g, a)               # a call, on the GPU, of a template
(a + b * c : T)  {x : T}               # operators over T; an annotation
[v : T*n]  [v : T^d]  a[i]  a[i] <- v  # an array of n or 2^d slots; a read, a write
{==}  %e : P; e2  %e@E : P; e2         # reflexivity, a rewrite, a named one
?name  ?TODO                           # print the goal; leave it open
```

```
# Statements (a def, case, lambda or parenthesized body)
x = v  +x = v  -x = v                  # a let: affine, reusable, erased
(a, b) = v  K{x, y} = v                # a destructuring let
a b = f(x) g(y)                        # a parallel let
match a b:                             # a match on one or more values
  case K{x, _} 1n+p:                   # patterns nest; _ catches the rest
do M<xs.., R>:                         # a monadic block over M.bind, M.pure
  x : A <- m                           # bind
  x : A = v                            # let
  m                                    # a Unit step
  return v                             # the result
```

Inside `(.. : T)`, `+ - * / %` call `T.add` through `T.mod`, `.&. .|. .^.` the bit operations, `<< >>` the shifts (by a `Nat`), and `< <= > >=` the `T.is_lt` family; without a `: T` they belong to `Nat`. `&& ||` work on `Bool` and `++` on `String` anywhere. Operators need spaces on both sides. Equality of values is a call, `T.is_eq(a, b)`; `==` is only the type. A `Nat` literal past `256n` is `U32.to_nat(n)` underneath, up to `4294967295n`.

UNDER THE HOOD

Bend's compiler emits one C file, and that file is both the CPU program and the GPU kernel. clang compiles it for the CPU. Metal (on Apple) or CUDA (on NVIDIA) compiles the same file for the GPU. So a `!` call runs the same code on whichever chip it lands on.

A term is one 64-bit word: small values are stored inline, everything else is a pointer into a single heap shared by every core and by the GPU. There is no garbage collector. Since values are affine, a `match` frees the node it opens on the spot, and only `+` values carry a reference count. There is no C stack either: each def compiles to a segment of a flat state machine, a call is a jump, and a parallel call creates a join task plus one task per call, which the scheduler deals across CPU or GPU lanes. `paper/BendRT.pdf` has the design and the benchmarks.

Bend's theory has one universe and no positivity check: `Type : Type` holds, and a datatype may recurse on the left of an arrow. What keeps this consistent is a wall between two checking modes. Code that runs is checked *live*; types, erased arguments and equations are checked *dead*. Dead code may loop forever or inhabit `Empty`, but nothing dead ever counts as live evidence, and live recursion must terminate. `bend2/bend.lean` mechanizes this, though it lags `bend.ts`; `paper/BendTT.pdf` is the paper.

FURTHER READING

- `demos/`: complete programs, including the game and its proof from the video.
- `bend2/base.bend`: the Base library, also printed by `bend base`.
- `paper/BendTT.pdf` and `paper/BendRT.pdf`: the type theory and the runtime.

EXTRA

`bend guide shaders` prints "Shaders in Bend", a tutorial written by AIs for AIs on how to write efficient shaders in Bend. It distills what building `demos/app_slash_boss_3d` (120 FPS in pure Bend) taught. Read it before you write a graphical or parallel app in Bend.

`bend guide effects` prints "Effects in Bend", an AI-written note (to be revised by a human) on the C and JS side of custom effects. Read it before you write one.

F.17 · bend-runtime/knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md

Shaders in Bend

AIs wrote this tutorial, for AIs, from the logs of building `demos/app_slash_boss_3d`. A human will revise it later. Report anything wrong on GitHub.

A fast shader in Bend is one fork tree that ends in flat loops. The reference is `demos/app_slash_boss_3d/bend3d.bend`, a rasterizer that draws 1920x1200 at 120 FPS (8.0 ms a frame) on a 10-core M4 mini. Every ms below is that machine and that frame unless stated.

THE MACHINE

A `!` call (a bang) ships one call to the GPU, which runs a cube of 16384 lanes (128 x 128). A parallel let `a b = f(x) g(y)` creates one task per call plus a join task. A kernel iteration grows the fork tree breadth-first until the frontier fills the cube, then every lane runs its task to the end: ~0.4-0.5 ms plus the slowest lane. A continuation that forks again lands in the next iteration. A def is flat when it has no parallel let, no bang, no closure apply, no non-flat call and no non-tail self call: it compiles to one native tail loop (`spin_N` in the C). A non-tail self call becomes a segmented continuation: a frame per call. Memory is one heap shared by the CPU and the GPU: unified on Apple; managed on CUDA, where each page faults across PCIe on the first touch by the other side. A machine with no GPU, or `--gpu off`, runs bangs on the CPU pool. Divergent per-lane work (a search, a tree walk per ray) is faster there than on the GPU. Aim for 4^7 leaves per bang, one per lane, and sweep one fork level either side on the device: the CPU pool is flat across that sweep, so it cannot tune it. Fewer leaves leave lanes idle (256 of 16384 busy: 234 ms for a frame that takes 28 ms at 4096); more cost an iteration each.

A FRAME

1. Host: build the scene (mesh, project, light, cull) into a quadtree of 64-px cells, one cons list of flat screen triangles per cell. 1.6 ms.
2. One bang: fork four ways per node, five levels to the cells, two more over each cell's list, to 16-px tiles: 4^7 = 16384 = the lane cube. A tile filters the cell's list into its own and walks it in flat loops. Draw ~6 ms.
3. The last frame rides down the same tree as `Four` quadrants; each tile leaves its quadrant unused, so the runtime sinks it there. Drop 0 ms.
4. Host: free the last scene as one branch of the next build. 0.35 ms.

```
 2048 root   Frame.node, k=5   fork 4   levels 1-5  -> 64-px cells (32 x 32)
64-px cell  Cell.fork, k=2    fork 4   levels 6-7  -> 16-px tiles (16384)
16-px tile  Tile.b16 = 4 Tile.b8 = 4 Tile.b4       straight-line, no fork
4-px square Blk.go            one flat loop over the list, four Blk
2-px block  Blk.four          one Pix, or four
```

The root is the power of two over the frame (2048 over 1920x1200); a region off the frame, or empty, answers one `Pix` before any fork (`Frame.sky`). A fork per pixel drowns in scheduling; a fork per cell leaves lanes idle. A fold per level plus a fold per pixel: 600 ms and 31 iterations a frame, against 8 ms and 1.

THE CODE

The list is the one shape a flat loop walks with no stack. One list per cell, not per pixel: a list shared by pixels is contended atomics.

```
type Tris is Data:
  TNil{}
  TCons{+tri: Tri, next: Tris}

type Cells is Data:
  CNil{}
  CLeaf{+n: U32, +ts: Tris}
  CQua{tl: Cells, tr: Cells, bl: Cells, br: Cells}
```

The fork. `Frame.show` bangs once and returns the tree beside the image, so the host still owns it. `Image.open(old) -> Four` splits the last image at the call (a `Pix` opens into four); each tile receives `old: Four` and ignores it.

```
def Frame.node(+k: Nat, c: Cells, old: Four) -> Image:
  match k c:
    case _ CLeaf{_, ts}:
      Cell.fork(2n, ts, 0, old)
    case 1n+e CQua{tl, tr, bl, br}:
      Four{oa, ob, oc, od} = old
      a b c d = Frame.node(e, tl, Image.open(oa)) Frame.node(e, tr, Image.open(ob))
        Frame.node(e, bl, Image.open(oc)) Frame.node(e, br, Image.open(od))
      Qua{a, b, c, d}
    case _ _:
      Pix{0}

def Frame.show(+cells: Cells, old: Image) -> Image & Cells:
  (Frame.node!(5n, cells, Image.open(old)), cells)
```

`Cell.fork(+k, +cs, ..)` has the same shape over the cell's list and calls `Tile.go` at `0n`. The tile: one flat loop reads each entry once for a block of `Hit{w, c, i}`, then frees its list after the last read.

```
def Blk.go(ts: Tris, +i: U32, +h0: Hit, +h1: Hit, +h2: Hit, +h3: Hit) -> Image:
  match ts:
    case TNil{}:
      Blk.four(h0, h1, h2, h3)
    case TCons{t, n}:
      Blk.go(n, (i + 1 : U32), Hit.step(t, i, h0), Hit.step(t, i, h1),
        Hit.step(t, i, h2), Hit.step(t, i, h3))

def Tile.go(cs: Tris, +x: U32) -> Image:
  +h = {Hit{0.0, 0, 0} : Hit}
  +ts = Tris.filter(cs, x, TNil{})
  Tile.fin(Blk.go(ts, 1, h, h, h, h), ts)
```

`Blk.four` answers `Pix{k0}` when the four hits are one triangle, else a `Qua` of four `Pix`; `Tile.fin` walks the list to `TNil` and returns the image. Transparency: the filter fills a second list (`Filt{+n, ts, bs}`) and `Blend.go` walks it in order over the blocks after the opaque loop. Text and HUD: strokes as opaque quads nearer than the world (`Text.put`, `Quad.put` at inverse depth 3000), layered by depth, order-free. The host: one parallel let per def, the last scene freed as a branch by `Cells.drop(4n)`: four forked levels in `Frame.node`'s shape, then `Tris.free` loops.

```
def Scene.build(+t: F32, used: Cells) -> Cells:
  a b c = Hero.build(t) Stage.build(t) Cells.drop(4n, used)
  Cells.m4(a, b, c, CNil{})
```

A turn keeps the last image and scene as one pair: the build takes the old scene, the bang the old image. An `IO.now()` step sits between them: a bang that follows the host's forks in one evaluation runs on the CPU pool. `Window.frame(window, image)` shows the image.

```
def Turn.go(+t: F32, last: Image & Cells) -> IO(Image & Cells):
  (old, used) = last
  do IO<Image & Cells>:
    cells : Cells = Scene.build(t, used)
    now : Nat <- IO.now()
    IO.pure(Image & Cells, Frame.show(cells, old))
```

ANY SCENE

The pixel never sees the scene. The fork tree opens each node once on the way down and hands each tile one short flat list; the pixel loop marches that list. For triangles, voxels, distance fields and sprites alike, the host walks the world once per frame into per-tile candidates (the primitives, columns or spans whose screen bounds meet the tile); the device reads only those. A field with no scene data (a procedural distance field) is the best case: one flat loop per pixel on a U32 step counter. A per-ray tree walk fails three ways: a fold over a tree is a non-tail recursion (a frame per node), the lanes of a SIMD group diverge, and a `+` tree pays an atomic per node per pixel. A reported case: a voxel world as a `+Data` quadtree of 32x32 columns, read per ray: 176 ms a frame, against 24 ms for the same picture computed per pixel with no scene data (the reporter's machine). The fix: keep the editable map on the host, and hand each tile a flat list of the column words its rays cross (a 32-high column is one U32 of bits) each frame. An `Array` has one owner, so it cannot go down a fork tree: build lists.

DATA

- Records of scalars ride in registers through every call and fork; a boxed record shared by every vertex is an atomic count per use. `Tri` is 24 flat words (barycentric, depth and colour planes, mode, alpha, box): one F32 per 64-bit word, ~200 B loaded per entry per walk.
- A wide record round-trips the stack: a 52-word record per list entry made the draw 6.3 -> 12.1 ms; a 22-word camera on every mesh call, +2-4% build. Build wide data where used (`Hook.cam(F)`).
- A def or continuation takes at most 255 words ("an arity over 255"): four 15-field records in one parallel let fail to compile. Wrap a big header in a constructor of the tree it heads.
- Use the typed `pick(c, a: F32, b: F32)` and `word(c, a: U32, b: U32)` ; the generic `Bool.pick` boxes its words.
- Tables are `word` chains and Bool-column matches. A `match` on U32 literals is a bit-by-bit tree with the default arm copied per branch: +21% host C, +6.4% `.gpu` , 1-4% slower.
- Pass a constant as a `~` template argument (a def like `Screen.w()`), not as a parameter: a parameter rides in every task.
- Tessellate in screen space: ~20-px patches; a circle gets `max(4, ceil(pi sqrt(r/2)))` segments. Slivers and sub-pixel triangles pass every box test and pile 100-400 entries on a tile (42 ms frames); exact triangle-vs-cell overlap holds the worst tile at 80-110. Length past that is not the cost: a cap of 16/32/64 entries per tile changed nothing.

OWNERSHIP

The compiler decides borrows; `+` does not. The checker needs `+` for a second use. The compiler borrows a boxed parameter (not an `Array`) that the def only matches or passes to a borrower: every read is `term_peek` , no count. It owns one that the def returns, stores in a constructor, or passes to an owner. An owned use of a value that is used later shares it: a count on the value and, since sealing is per type, on every node of that type, silently. So `+ts` above borrows: `Blk.go` only matches, `Tile.fin` consumes last. So the scene is host-owned, borrowed by the bang, and returned (`Frame.show`). Read through counts, the frame costs 8.0 -> 16.5 ms (draw 6.0 -> 14.1).

- Two consumers of one list share it: 45 ms frames and a 448 s Metal compile. A `+` where one use would do heats its type (symreg 2.83 -> 3.30 s).
- A def that returns its argument shares it: `Bool.pick(Cloth, c, a, b)` on a tree gave 14 keeps and a hot type. Match the selector and recurse into one field (`Cells.put` has eight arms for this). Refolding a matched `+` record at a call is an owned argument (8-16 keeps): pass the fields.
- The last reader of a borrowed list needs a later use, else it becomes the owner: `Tile.fin` walks the lists after the pixels read them (as dead parameters the runtime's sink is 0.55 ms slower).
- A bang inside a `do` continuation flips its callee to owner (the tree is a captured local): 2x reads. Bang in a pure def that returns the pair.
- Drop the last image inside the fork tree (a dead `Four` per tile) and the last scene on the host by a walk (35% faster than the runtime sink); never inside a bang: 12.9 ms. `_ = x` drops a value on the spot.
- Never let the device own the scene: draw 6.2 -> 12.0 ms.

Verify in the emitted C (`bend x.bend -o x.c`): in a def's body, `term_peek` is a borrow; `term_keep` , `ctr_take` and `rfc_seal` are counts. A new keep in a device def is a regression.

COST MODEL

- Mesh, project, light and cull on the host: 0.6-1.6 ms in parallel lets; on the GPU 14-33 ms (a P-core is ~75x a lane on serial work).
- Leaves are flat loops over cons lists; joins are constructors. Unroll the squares: `Tile.b16` = 4 x `Tile.b8` = 4 x `Tile.b4` ; folded into one recursive def they made the draw 5.5 -> 12.6 ms.
- Walk the list once per 4x4 square, not per pixel or per 2x2 block: 6.6 -> 5.6 ms. Anything that depends on x alone, once per column pair: sky per pixel 3.8 ms, per block 1.6, per column pair ~1.0 less.
- The draw is throughput-bound, not tail-bound: splitting heavy tiles +0.03 ms, 32-px cells +0.85. A per-lane early-out saves nothing while a SIMD sibling runs the body.
- Drops: `Image.drop` forked at every `Qua` (~100 ops of task machinery per node), 2.6 ms; base's `Image.free(7n)` , seven levels then a sink, 1.6 ms; inside the draw's tree, 0; a whole image sunk on one lane, 19.8.
- Host: one parallel let per def (a second costs an iteration: build 2.32 -> 1.93 ms); every extra branch or level ~0.1 ms; a 0.3 ms job loses to the pool wake-up; a heavy branch serializes a 16-ring chunk (1.2-1.6 ms): split it.
- CUDA: a scene built by the host and read by the device faults each page over PCIe, both ways: 17-19 FPS on an RTX 4090 (25 ms `gpu_pass` + ~21 ms host turn) for the frame the M4 draws in 8. Keep a whole frame on one side of the bus.

MEASURING

Probe as the game does (`SLASH_PROBE=1200`): scripted frames headless, each stage's microseconds as medians after warm-ups. Single-run medians vary 0.3-0.5 ms across days, and the second binary of a GPU pair runs ~7% faster: compare by 3-4 interleaved 1200-frame pairs in both orders, plus an A-vs-A pair. Check frames pixel-identical (`SLASH_DUMP=T` prints one as a tree) before trusting a gain. Count the keeps in the C and the `.gpu` bytes (+6% was a real 1-4% slowdown). Probe without audio: it wakes `audiomxd` at 70-100% of a core.

DO NOT

1. Read the scene through counts: 2x per read, 8.0 -> 16.5 ms.
2. Fold per level, or bang twice a frame: 600 ms against 8.
3. Recurse non-tail in the bang, or fold a tree per pixel: draw 5.5 -> 12.6.
4. Mesh on the GPU: 14-33 ms against 0.6-1.6.
5. Fork per pixel or per `Qua` : drop 2.6 vs 1.6 ms.
6. Leave slivers and sub-pixel triangles: 42 ms frames.
7. Wide records through frames and joins: 2x per entry.
8. `match` on U32 literals: +21% C, +6.4% `.gpu` .
9. Generic `Bool.pick` on words or trees: boxed words, hot types.
10. Free the scene in a bang (12.9 ms) or the image on the host (milliseconds).
11. Trust one run, or the CPU pool (the 600 ms frame took 11 ms there), as a device measurement.

REFERENCE

`demos/app_slash_boss_3d/bend3d.bend` is the library; its comments are these rules at their sites. `main.bend` beside it is the app, with `Play.loop` and the probes.

F.18 · `bend-runtime/knowledge/03-hvm-inet-theory/01-interaction-nets-foundations.md`

01 — Interaction Nets Foundations

Sources fetched 2026-09-28:

- Wikipedia "Interaction nets" (https://en.wikipedia.org/wiki/Interaction_nets `(https://en.wikipedia.org/wiki/Interaction_nets)`)
- Interaction Calculus (https://github.com/VictorTaelin/Interaction-Calculus `(https://github.com/VictorTaelin/Interaction-Calculus)`)
- HVM1 HOW.md (https://github.com/HigherOrderCO/HVM1/blob/master/guide/HOW.md `(https://github.com/HigherOrderCO/HVM1/blob/master/guide/HOW.md)`)
- HVM2 `src/hvm.rs` (https://github.com/HigherOrderCO/hvm/blob/main/src/hvm.rs `(https://github.com/HigherOrderCO/hvm/blob/main/src/hvm.rs)`)

1. ORIGIN: LAFONT 1989

Interaction nets were devised by French mathematician Yves Lafont in 1989 (Lafont, "Interaction Nets", POPL '90, pp. 95–108) as a generalisation of the proof structures of linear logic. The goal was a graphical, local, resource-aware model of computation where each rewrite step touches only a bounded neighbourhood of the graph.

Unlike Turing machines (global tape + head) or term rewriting (nested substitution + traversal), interaction nets reduce by purely local graph rewrites. This is why HVM1's HOW.md calls the interaction net a model that "supersedes the Turing Machine and the Lambda Calculus" for parallel functional evaluation: there is no global environment, no renaming pass, no garbage-collector root scan in the core rule.

2. SIGNATURE, AGENTS, PORTS, ARITY

An interaction system is a pair `(Σ, R)` :

- `Σ` is a **signature**: a set of agent types (symbols), e.g. `Σ = {ε, δ, γ}` for interaction combinators.
- `ar : Σ → ℕ` assigns each agent an **arity**: the number of auxiliary ports.
- Every agent has exactly one **principal port** plus `ar(α)` auxiliary ports.

Example: $ar(\varepsilon) = 0$, $ar(\delta) = ar(\gamma) = 2$. Graphically an agent is drawn as a triangle/circle with one distinguished (often arrow-marked) principal port and n auxiliary legs.

Ports obey a linearity invariant:

1. Every port is incident to at most one edge.
2. Every edge connects exactly two ports.
3. Ports incident to no edge are **free ports**; together they form the **interface** of the net.

This is an affine/linear discipline at the graph level. It is the reason HVM can be "linear like Rust" (HOW.md TL;DR): every object logically exists in one place.

3. NETS, WIRINGS, TREES, INTERFACES

Three primitive structures (cf. Wikipedia "Primitives of Interaction Nets" diagram):

- **Edge/wire**: a direct connection between two ports.
- **Wiring** ω: a net consisting solely of edges (i.e. a permutation
  - partial pairing of free ports). Wirings translate to bare names in the calculus.
- **Tree** t with root x: inductively either an edge x, or an agent α whose free principal port is x and whose auxiliary ports x_i are connected to roots of subtrees t_i.

Any net N can be redrawn as a set of trees joined by a wiring layer. That decomposition is exactly what makes the textual **interaction calculus** possible: trees become terms, wiring becomes name sharing / equations.

4. ACTIVE PAIRS AND INTERACTION RULES

When two agents are connected **principal-port to principal-port** they form an **active pair**. Only active pairs may rewrite. No other redex notion exists.

For each unordered pair $(\alpha, \beta) \in \Sigma \times \Sigma$ there is at most one **interaction rule**:

```
α[...] ⋈ β[...]  →  N
```

where N is an arbitrary net with the same interface as the redex (free ports preserved). Graphically the two principal ports annihilate and the auxiliaries are rewired into N.

A net with no active pair is in **normal form**. Reduction is the iterated replacement of active pairs by their reducts.

Key consequences:

- **Locality**: only active pairs rewrite.
- **Linearity**: each rule fires in O(1) — bounded work, no traversal.
- **No synchronisation needed**: disjoint active pairs never interfere.

5. INTERACTION CALCULUS (FERNÁNDEZ–MACKIE)

The textual form is the **interaction calculus** (Fernández & Mackie, PPDP 1999). Grammar:

```
t ::= α(t₁,…,tₙ) | x            (terms: trees or names)
c ::= (t₁,…,tₘ | v₁=w₁,…,vₙ=wₙ) (configuration: interface + equations)
```

Each name must occur exactly twice (linearity). $\langle \ldots | \ldots \rangle$ separates the observable interface (left) from the unordered multiset of equations (right). ω becomes names; trees become nested terms.

Two reduction relations:

- **Interaction**: if $\alpha[v_1 \ldots v_m] \bowtie \beta[w_1 \ldots w_n]$ is a rule, then $\langle \vec{t} \mid \alpha(t_1 \ldots t_m)=\beta(u_1 \ldots u_n), \Delta \rangle \to \langle \vec{t} \mid t_1=v_1, \ldots, u_n=w_n, \Delta \rangle$.
- **Indirection** (substitution): $\langle \ldots t \ldots \mid x=u, \Delta \rangle \to \langle \ldots t[x:=u] \ldots \mid \Delta \rangle$ when x occurs once in t. Variants handle $t=w$ equations.

Plus structural α-conversion: both occurrences of a name may be renamed fresh. $c \downarrow c'$ means c reduces to equation-free normal form.

An equation $x = t$ where x occurs in t is a **deadlock**; theory usually restricts to deadlock-free nets.

6. WHY THIS MATTERS FOR HVM AND BEND

HVM1/HVM2 and Bend all compile to this substrate:

- HVM1 HOW.md core language (`λ` , application, constructors, `dup`) is syntactic sugar over nets: `dup` is a floating duplicator agent, lambdas/applications are binary agents, constructors are n-ary `CON` .
- HVM2 `src/hvm.rs` generalises agents to 8 tags (`VAR,REF,ERA,NUM,CON,DUP,OPR,SWI`) with an 8×8 rule table — a direct mechanisation of "at most one rule per principal-principal pair".
- Interaction Calculus (Taelin) refines the calculus with affine globals, `SUP` / `DUP` , and labelled commutation, preserving the same configuration/equation intuition but optimised for machine layout.
- Bend's compiler inserts `dup` wherever a variable is used >1, restoring the "every name twice" invariant automatically so user code never sees equations.

Understanding agents/ports/active-pairs is therefore prerequisite to reading `Port::get_rule` , `interact_anni` , `interact_comm` , and Bend's `dup` elaboration in the following notes.

7. MINIMAL EXAMPLES

- Wire alone: `⟨x | ø⟩` — identity net, interface of one free port.
- Eraser applied: `ε ⋈ α[ε…ε]` — deleting a tree (see note 02).
- Self-loop deadlock: `x = α(x)` — excluded; would be a cyclic equation.
- Two disjoint redexes: both can fire in either order or in parallel — preview of strong confluence (see note 03).

8. FURTHER READING

- Lafont 1989 POPL; Lafont 1997 "Interaction Combinators" (Inf. Comp.).
- Asperti & Guerrini, *The Optimal Implementation of Functional Programming Languages* (CUP 1998), Ch. 1–3.
- Fernández (2009), "Interaction-Based Models of Computation".
- INL, INblobs, iNet playground, BOHM for executable counterparts.

F.19 · `bend-runtime/knowledge/03-hvm-inet-theory/02-symmetric-combinators-rules.md`

02 — Symmetric Combinators and Interaction Rules

Sources fetched 2026-09-28:

- Wikipedia "Interaction nets" § Interaction combinators
- Lafont 1997 "Interaction Combinators" (Inf. Comp. 137(1):69–101)
- Interaction Calculus (https://github.com/VictorTaelin/Interaction-Calculus `(https://github.com/VictorTaelin/Interaction-Calculus)`)
- HVM1 HOW.md §§ Rewrite Rules (Dup-Lam, Dup-Sup, App-Sup)
- HVM2 `src/hvm.rs` rule table + `interact_eras/anni/comm`

1. THE UNIVERSAL SYSTEM: Σ = {ε, δ, γ}

One of the simplest universal interaction systems uses only three agents (Lafont 1997):

- `ε` (ERASE, arity 0): deletes resources.
- `δ` (DUPLICATE, arity 2): copies resources.
- `γ` (CONSTRUCT, arity 2): builds/application node; takes two resources, returns a combined one (binary op, pair, or `f x`).

Universality means any other interaction system can be encoded into `{ε,δ,γ}` . Wikipedia states this explicitly: combinators "can simulate any other interaction system". HVM exploits this: instead of compiling each datatype to bespoke agents, it keeps generic `CON/DUP/ERA` plus `OPR/SWI/NUM` for arithmetic — a pragmatic extension of the triad.

2. THE THREE RULE FAMILIES

2.1 ERASING: E ⋈ A[E…E]

```
ε ⋈ α[ε,…,ε]
```

When `ε` meets any agent `α`, `α` is deleted and an `ε` is propagated to each of `α`'s auxiliary subtrees. Cost O(arity), i.e. O(1) for fixed signature. In HVM2 this is `ERAS` (`interact_eras` in `hvm.rs`): pop node `b`, link `a` to both children `b1,b2`. In HVM1 HOW.md the twin rules are `Dup-U32` / `Dup-Ctr` specialised to numbers vs constructors.

Intuition: garbage collection without tracing — unreachable subtrees are erased lazily, layer by layer, only when demanded.

2.2 DUPLICATION (COMMUTATION OF Δ WITH A)

```
δ[α(x₁…xₙ), α(y₁…yₙ)] ⋈ α[δ(x₁,y₁),…,δ(xₙ,yₙ)]
```

A duplicator meeting a constructor pushes through it: one `δ` on the outside becomes `n` `δ`s on the inside, rebuilding `α` twice. This is the workhorse of lazy cloning. HOW.md § "Constructor Duplication" spells it identically with `dup x y = (Foo a b…)` producing `dup a0 a1 = a`, `dup b0 b1 = b`, `x <- (Foo a0 b0…)`.

2.3 ANNIHILATION: Δ⋈Δ AND Γ⋈Γ

```
δ[x,y] ⋈ δ[x,y]
γ[x,y] ⋈ γ[y,x]   (note swap on γ)
```

Like-meets-like principal ports cancel, wiring auxiliaries straight through. The `γ` swap reflects orientation (constructor symmetry). In HVM2 this is `ANNI` (`interact_anni`): take both nodes, link `a1~b1`, `a2~b2`. `ANNI` needs no fresh allocation — cheapest rule.

Together erasing + duplication + annihilation are complete: any net built from the triad normalises using only these.

3. ANNIHILATION VS COMMUTATION IN HVM2

HVM2 generalises the triad to 8 tags. `Port::get_rule` encodes the full 8×8 dispatch table in `hvm.rs`:

```
      VAR REF ERA NUM CON DUP OPR SWI
VAR   LINK ...
REF   LINK VOID VOID VOID CALL ...
ERA   LINK VOID VOID VOID ERAS ...
NUM   LINK VOID VOID VOID ERAS OPER SWIT
CON   LINK CALL ERAS ERAS ANNI COMM ...
DUP   LINK CALL ERAS ERAS COMM ANNI ...
OPR   LINK CALL ERAS OPER COMM COMM ANNI ...
SWI   LINK CALL ERAS SWIT COMM COMM COMM ANNI
```

- Diagonal (`CON–CON`, `DUP–DUP`, …) = **annihilation** family.
- Off-diagonal node–node (`CON–DUP`, `CON–OPR`, …) = **commutation** family (`interact_comm`): allocate 4 fresh vars + 4 fresh nodes, cross-wire copies. This is the expensive rule (4 nodes + 4 vars).
- `ERAS/OPER/SWIT/CALL/VOID/LINK` are typed specialisations (erase, numeric fold, pattern switch, global unfold, no-op, var link).

`should_swap` normalises orientation so the smaller tag is first; `is_high_priority` puts `LINK/CALL/VOID` on the `hi` bag for prompt handling.

4. WORKED TRACE: THE NON-TERMINATING LOOP

Wikipedia gives the classic cyclic combinator net `δ(ε,x) = γ(x,ε)` which reduces to itself. In calculus:

```
(ø | δ(ε,x) = γ(x,ε))
→ (ø | ε=γ(x₁,x₂), x=γ(y₁,y₂), x=δ(x₁,y₁), ε=δ(x₂,y₂))   (commutation)
→* (ø | δ(ε,x) = γ(x,ε))                                  (erasures)
→ …
```

This is the interaction-net analogue of `Ω = (λx.x x)(λx.x x)`: finite net, infinite reduction, no normal form. It proves the system is Turing complete even with 3 agents, and warns Bend authors that termination is not guaranteed by typing in untyped HVM.

5. FROM Δ/Γ TO SUP/DUP WITH LABELS

Plain combinators cannot distinguish *which* duplicator should annihilate with *which* copy — nested dups collapse incorrectly. The Interaction Calculus fix (Taelin repo, fetched 2026-09-28) labels superpositions and duplications:

```
Term ::= VAR | ERA("*") | LAM λx.t | APP (f a)
       | SUP &L{a,b} | DUP !&L{x,y}=v; b
```

Core rules:

- `APP-LAM` (beta): `(λx.f a) → x<-a; f`.
- `APP-SUP`: `(&L{a,b} c) → !&L{c0,c1}=c; &L{(a c0),(b c1)}`.
- `DUP-LAM`: project a lambda through a dup, superposing the bound var.
- `DUP-SUP equal labels`: annihilate (`x<-a; y<-b`).
- `DUP-SUP different labels`: commute with nested dups (`x<-&R{a0,b0}; y<-&R{a1,b1}; !&L{a0,a1}=a; !&L{b0,b1}=b`).

Labels recover confluence for nested sharing (the "clone of a clone" problem in HOW.md). HVM2 carries the label in the high bits of the `DUP` port value; IC32 exposes `SP0–SP7/CX0–CY7` tag families.

6. COLLAPSING AND STRICT EXTENSIONS

IC adds **collapse** rules (`ERA-LAM`, `ERA-APP`, `SUP-LAM`, `SUP-APP`, `DUP-VAR`, `DUP-APP`) to project a superposed tree back to pure λ-terms, plus `DUP` permutations (`APP-DUP`, `DUP-DUP`) to float nested dups out of redex position. **Labeled lambdas** (`&Lλ`) extend `APP-LAM` to handle differing labels via extra superposition. HVM2's `OPER/SWIT` are the strict-number analogues: `(+ {a0,a1} b)` duplicates `b` and superposes results — see HOW.md "Superposed Operation".

7. PRACTICAL READING GUIDE

- To predict allocation: `ANNI` =0 alloc, `ERAS` =0 alloc +2 links, `COMM` =4 nodes+4 vars+4 links, `OPER/SWIT` =1–2 nodes.
- To debug Bend output: look for `DUP` fans (`let x0,x1 = dup x`) and matching `SUP` constructors (`&L{…}`); mismatched labels mean a commutation storm is coming.
- To extend the system: adding a new agent means adding one row/column to `get_rule` plus one `interact_*` function — locality guarantees no other rule changes.

8. REFERENCES

- Lafont 1997, §3–4 (erasing/duplication/annihilation diagrams).
- Wikipedia "Interaction combinators" + "Examples of Interaction Nets".
- IC README §§ Specification, Collapsing, DUP Permutations, Labeled Lambdas, IC32.
- HVM1 HOW.md §§ Number/Constructor/Lambda Duplication, Superposed Application/Operation/Duplication.
- HVM2 `src/hvm.rs` consts `VAR…SWI`, `LINK…SWIT`, `interact_*`.

F.20 · `bend-runtime/knowledge/03-hvm-inet-theory/03-confluence-optimal-reduction.md`

03 — Confluence, Optimal Reduction, and Parallelism

Sources fetched 2026-09-28:

- Wikipedia "Interaction nets" §§ Properties, Non-deterministic extension
- Interaction Calculus (https://github.com/VictorTaelin/Interaction-Calculus `(https://github.com/VictorTaelin/Interaction-Calculus)`)
- HVM1 HOW.md (TL;DR, "What makes it fast", Lambda Duplication, Bonus sections on beta-optimality/parallelism, safe-pointer discussion)
- HVM2 `src/hvm.rs` (RBag hi/lo, `enter`, atomics, evaluator loop)
- Asperti–Guerrini, *The Optimal Implementation of Functional Programming Languages* (TOIOFPL); Lambdascope; BOHM

1. STRONG CONFLUENCE: THE ONE-STEP DIAMOND

Interaction nets enjoy three properties (Wikipedia § Properties):

1. **Locality** — only active pairs rewrite.
2. **Linearity** — each rule fires in constant time.
3. **Strong confluence** (one-step diamond): if $c \to c_1$ and $c \to c_2$ then there exists c' with $c_1 \to c'$ and $c_2 \to c'$ in at most one step each.

Proof sketch: two distinct redexes are either disjoint (commute trivially) or overlap on at most one agent — but principal ports pair uniquely, so overlap implies identity. Hence no critical pairs. There is no need for a strategy: any reduction order reaches the same normal form if one exists, with the same *number* of interactions up to permutation.

Consequence for runtimes: a global work queue with atomic push/pop is sufficient. HVM2's `RBag { lo, hi }` plus `TMem::interact` popping any redex is sound precisely because of the diamond. No locks around rule selection; only allocation and `vars_exchange` need atomics.

2. MASSIVE PARALLELISM FOR FREE

Because redexes never interfere, "computations can take place simultaneously in many parts of an interaction net, and no synchronisation is needed" (Wikipedia lede). HVM1 benchmarks this: Bubble Sort (inherently sequential) shows no speedup, while Radix Sort (tree merge, inherently parallel) goes ~2.5× faster than GHC on 8 cores with zero annotations. The parallelism is *automatic*: the compiler does not insert `par`; the runtime just feeds disjoint active pairs to different workers.

HVM2 mechanises this with:

- `GNet { node: &[APair], vars: &[APort] }` — flat atomic buffers.
- `TMem { tid, tids, tick, itrs, nput, vput, nloc, vloc, rbag }` per worker; `node_alloc/vars_alloc` stripe allocation by thread id.
- `link()` with `vars_exchange + enter()` path compression; only dup nodes and substitution need synchronisation (HOW.md FAQ "Why is HVM so parallelizable?").
- `is_high_priority` steering `LINK/CALL/VOID` to the `hi` bag so administrative steps drain fast and expose more parallel `ANNI/COMM`.

Caveat: `COMM` allocates 4+4 cells; parallel `COMM` storms can contend on the allocator. Bend mitigates by linearising hot paths and keeping `def.safe` (dup-free) definitions on the fast `CALL→ERAS` path.

3. LÉVY OPTIMALITY AND WHY GHC DUPLICATES WORK

Optimal reduction (Lévy 1978) means: never duplicate a redex that will later be shared; never re-reduce the same family. Naive β-reduction copies lambdas wholesale, re-evaluating `(+ 2 2)` inside each copy. Lamping (1990) showed interaction nets with sharing nodes (fans, brackets/croissants) achieve Lévy-optimal λ-reduction.

HOW.md's centrepiece trace makes this concrete:

```
dup f g = ((λx λy (Pair (+ x x) y)) 2)
(Pair (f 10) (g 20))
→* (Pair (Pair 4 10) (Pair 4 20))
```

`(+ 2 2)` fires *once* even though it sits under two copied binders. Haskell thunks cannot share inside lambdas (free variables break memoisation), so GHC evaluates it twice. Repeated composition (`Add` as `App xs λx(Inc x) ys`, `Mul` on Scott/λ-encoded bits) turns this constant-factor win into an *exponential* win: HVM multiplies λ-encoded numbers exponentially faster than GHC (HVM1 README chart). HOW.md "Bonus: Abusing Beta-Optimality" generalises to deforestation: `map f . map g → map (f . g)` happens at runtime, even when `funcs` is unknown statically (`foldr (.) id funcs`).

4. SHARING, ORACLES, AND THE COST OF PERFECTION

Full Lévy optimality needs an **oracle** to match fans correctly (brackets, croissants, delimiters, safe pointers — TOIOFPL pp. 40+). That machinery costs ~10× on β-reduction (HVM1 FAQ). HVM1 therefore implements the *bookkeeping-free* fragment (TOIOFPL pp. 14–39): fast, covers Y-combinators, Church numerals, Scott datatypes, recursion, but rejects the pathological self-cloning shape:

```
let g = λf λx (f (f x)) in (g g)   -- clone cloning its own clone: UB
```

Fix by eta-splitting (`(g h)` with `g,h` identical bodies) or by EAL inference (Elementary Affine Logic) converting full lambdas to light lambdas. The IC's labelled `DUP-SUP` (equal → annihilate, different → commute with nested dups) is the lightweight oracle: labels decide locally, no global bracket stack. IC32's `SP/CX/CY` tag families and HVM2's 29-bit `Lab` field carry those labels at runtime.

Worst-case note (HVM issue #60, Hacker News post-mortem cited in FAQ): some programs are *quadratic* on HVM but linear on GHC. Optimality is about shared higher-order families, not every first-order loop. Safe pointers + linearity analysis + optional mutability are the planned mitigations.

5. TERMINATION, DEADLOCKS, AND NON-DETERMINISM

- Confluence ≠ termination: the `δ(ε,x)=γ(x,ε)` loop (note 02) is confluent but diverges. Bend must rely on recursion measures, not the net model, for termination.
- Deadlock (`x = t(x)`) is excluded by construction in HVM (acyclic allocation + `enter` loop guard); a cyclic equation would spin `enter` forever.
- Pure nets are deterministic. Fernández–Khalil `amb` (two principal ports, Wikipedia § Non-deterministic extension) adds choice and `ParallelOr`. HVM/Bend omit `amb`; `SWI + OPR` cover deterministic branching. Anyone adding search to Bend should reintroduce `amb` explicitly rather than abusing `DUP` labels.

6. FROM THEORY TO `HVM.RS` PRIORITIES

Theory concept	Runtime reflex
Diamond / disjoint redexes	lock-free `RBag::pop_redex`, thread pool
Administrative vs real work	`hi` bag for `LINK/CALL/VOID`, `lo` for `ANNI/COMM/OPER/SWIT`
Sharing families	`Lab` on `DUP/CON`, `DUP-SUP` label compare
Optimal vs fast fragment	`Def.safe` flag; safe globals `CALL→ERAS` shortcut
Oracle cost	labels instead of brackets; documented UB on clone-of-clone
Deforestation	`APP-SUP` / `SUP-APP` pushing applications through superpositions

7. WHAT BEND INHERITS

- Order-independence: Bend's parallel `bend run --threads` needs no program annotations; any schedule is correct.
- Cost model: count `ANNI < ERAS < OPER/SWIT < COMM`; minimise `COMM` by keeping functions linear and labels distinct.
- Fusion for free: pipeline `map` / `fold` chains on Church/Scott lists fuse at runtime; prefer λ-lifted shared prefixes (HOW.md `Not` / `Inc` pattern) to keep normal forms small.

- Escape hatch: when optimal sharing hurts (quadratic clones), rewrite to accumulator/loop form or native `U24` ops — the same advice as HOW.md "Abusing Parallelism" TODO extended.

8. REFERENCES

- Lévy 1978 (optimal families); Lamping 1990 (abstract algorithm).
- Asperti–Guerrini TOIOFPL, Ch. 10 (complexity bounds apply 1-to-1 to HVM's higher-order machinery per HOW.md TL;DR).
- van Oostrom et al. 2010 "Lambdascope"; Mackie 2008 "Closed Reduction".
- IC README §§ "IC = Lambda Calculus ∪ Interaction Combinators", "Collapsing", IC32 runtime.
- HVM2 `src/hvm.rs`: `get_rule`, `is_high_priority`, `enter`, `link`, `evaluator`, `Numb::operate`.

F.21 · `bend-runtime/knowledge/03-hvm-inet-theory/04-hvm-runtime-tags-memory.md`

04 — HVM Runtime: Tags, Memory Layout, and Reduction Engine

Sources fetched 2026-09-28:

- HVM2 `src/hvm.rs` (full file via webfetch: types, `Port` / `Pair`, `Numb`, `RBag`, `GNet`, `TMem`, `Def` / `Book`, `interact_*`, `evaluator`, serialization, `show`, `test_f24`)
- HVM1 HOW.md §§ Low-level Implementation, Rewrite Rules
- Interaction Calculus README § IC32: a 32-Bit Runtime + Parsing + Stringifying IC32

1. DESIGN GOAL: GRAPHS WITHOUT POINTER-CHASING OVERHEAD

Early optimal-reduction implementations stored λ-terms as heap graphs with per-node pointers, paying indirection on every step. HVM2 (and IC32) instead pack the net into flat atomic buffers with 32-bit ports and 64-bit pairs, exploiting the fact that inputs are known to be λ-terms/constructors (HOW.md TL;DR: "50% lower memory usage … 50× faster"). A lambda is 2×64-bit pointers — "about as lightweight as it gets" (HVM1 FAQ).

2. PORT, PAIR, NUMB: THE THREE WORDS

2.1 `PORT(PUB VAL)` — 32 BITS: 29-BIT VAL + 3-BIT TAG

```
pub type Tag = u8; pub type Lab = u32; pub type Val = u32;
impl Port {
  pub fn new(tag: Tag, val: Val) -> Self { Port((val << 3) | tag as Val) }
  pub fn get_tag(&self) -> Tag { (self.0 & 7) as Tag }
  pub fn get_val(&self) -> Val { self.0 >> 3 }
  pub fn is_nod(&self) -> bool { self.get_tag() >= CON }
  pub fn is_var(&self) -> bool { self.get_tag() == VAR }
}
```

Tags (`hvm.rs` consts):

```
VAR 0x0  variable (points into vars buffer / substitution slot)
REF 0x1  global reference (function id + par flag bit 28)
ERA 0x2  eraser
NUM 0x3  number (payload is Numb-encoded, see §3)
CON 0x4  constructor (points into node buffer)
DUP 0x5  duplicator (points into node buffer, carries Lab)
OPR 0x6  operator (points into node buffer)
SWI 0x7  switch (pattern match on NUM)
```

`FREE = 0x0`, `ROOT = 0xFFFFFFF8`, `NONE = 0xFFFFFFFF` are sentinels. `is_nod` distinguishes heap nodes (`CON/DUP/OPR/SWI`) from inline atoms (`VAR/REF/ERA/NUM`).

2.2 PAIR(PUB U64) — TWO PORTS, ONE NODE

```
pub struct Pair(pub u64);
Pair::new(fst, snd) = (snd.0 as u64 << 32) | fst.0 as u64
```

A node is exactly one `Pair`: the two auxiliary ports of a `CON/DUP/ OPR/SWI` agent. The principal port is the *reference* to the node (`Port::new(tag, loc)`). `get_fst/get_snd`, `adjust_pair` (relocate during `CALL` copy), `set_par_flag/get_par_flag` (parallel-call bit on `REF`) complete the API.

2.3 NUMB(PUB VAL) — NUMBERS INSIDE A PORT

HVM2 packs U24/I24/F24 + symbolic ops into the 29-bit `NUM` payload:

```
TY_SYM 0x00, TY_U24 0x01, TY_I24 0x02, TY_F24 0x03,
OP_ADD…OP_SHR, FP_* reversed variants (see hvm.rs consts)
```

Key methods: `new_u24/new_i24/new_f24` (F24 uses round-ties-even + NaN-preserving shift), `get_typ/is_num/is_cast`, `partial` (curried operator), `cast` (Rust-like saturating semantics), `operate` (wrapping int arithmetic, float `+ - * / %`, comparisons returning `U24` booleans, `atan2/log/powf/sin/tan` overloaded on `AND/OR/XOR/ SHL/SHR` for F24). `test_f24` in `hvm.rs` pins round-trip, ties-even, and NaN≠∞ behaviour.

IC32 (Taelin README) is the 32-bit cousin: 1-bit `sub` + 5-bit `tag` (`VAR/LAM/APP/ERA/NUM/SUC/SWI/TMP/SP0–7/CX0–7/CY0–7`) + 26-bit `val`; `DUP` floats on the heap, reachable only via `CX/CY` vars — same idea, tighter packing.

3. RULE DISPATCH: THE 8×8 TABLE

```
pub fn get_rule(a: Port, b: Port) -> Rule {
  const TABLE: [[Rule;8];8] = [
    [LINK,…],               // VAR
    [LINK,VOID,VOID,VOID,CALL,…], // REF
    [LINK,VOID,VOID,VOID,ERAS,…], // ERA
    [LINK,VOID,VOID,VOID,ERAS,ERAS,OPER,SWIT], // NUM
    [LINK,CALL,ERAS,ERAS,ANNI,COMM,COMM,COMM], // CON
    [LINK,CALL,ERAS,ERAS,COMM,ANNI,COMM,COMM], // DUP
    [LINK,CALL,ERAS,OPER,COMM,COMM,ANNI,COMM], // OPR
    [LINK,CALL,ERAS,SWIT,COMM,COMM,COMM,ANNI], // SWI
  ];
  TABLE[a.get_tag()][b.get_tag()]
}
pub fn should_swap(a,b) -> bool { b.get_tag() < a.get_tag() }
pub fn is_high_priority(rule) -> bool { (0b00011101 >> rule) & 1 != 0 }
```

Rules: `LINK 0, CALL 1, VOID 2, ERAS 3, ANNI 4, COMM 5, OPER 6, SWIT 7`. `should_swap` canonicalises orientation; `ROOT` forces `CALL`. High-priority (`LINK/CALL/VOID`) go to `RBag.hi`, rest to `lo`, so `pop_redex` drains administrative steps first.

4. GLOBAL NET AND THREAD MEMORY

```
pub struct GNet<'a> {
  pub nlen: usize, pub vlen: usize,
  pub node: &'a mut [APair], pub vars: &'a mut [APort],
  pub itrs: AtomicU64,
}
pub struct TMem {
  pub tid: u32, pub tids: u32, pub tick: u32, pub itrs: u32,
  pub nput: usize, pub vput: usize,
  pub nloc: Vec<usize>, pub vloc: Vec<usize>,
  pub rbag: RBag,
}
```

- `GNet::new` allocates raw `APair/APort` slices with `alloc` / `Layout`; `node_create/vars_create/load/store/exchange/take`, `is_node_free/is_vars_free`, and `enter` (VAR-chain path compression with `vars_exchange … NONE + vars_take`) are all `Relaxed` atomics. `Drop` frees both buffers.
- `TMem::new`, `node_alloc/vars_alloc` (stripe by `nput/vput`, reserving index 0 for `FREE`), `get_resources` (admission check before each interaction — if allocation fails the redex is pushed back and the worker yields).

- `Def { name, safe, root, rbag, node, vars }`, `Book { defs }` hold compiled functions; `safe` = dup-free, enabling the `CALL→ERAS` shortcut when a safe global meets `DUP` (else abort with "attempt to clone a non-affine global reference" — the clone-of-clone UB guard).

5. THE EIGHT INTERACTIONS (`TMEM::INTERACT_*`)

- `link(a,b)`: if `a` is node and `b` is var, swap; if `a` is node, push `Pair(a,b)` to rbag; else `enter(b)` then `vars_exchange(a,b)` with ownership dance (`NONE` check + `vars_take`). `link_pair` wraps it. `interact_link` = pure link.
- `interact_call(a,b,book)`: `fid = a.val & 0xFFFFFFF`; DUP+unsafe guard; `get_resources(rbag+1, node.len, vars)`; `vars_create(NONE)`, `node_create(adjust_pair)`, replay `def.rbag`, link `def.root ~ b`. Copies the function body into the net.
- `interact_void`: no-op (REF/ERA/NUM erasure resolved at link time).
- `interact_eras(a,b)`: `take(b) → (b1,b2)`; link `a~b1`, `a~b2`.
- `interact_anni(a,b)`: `take(a)`, `take(b)`; link `a1~b1`, `a2~b2`.
- `interact_comm(a,b)`: needs 4 rbag + 4 node + 4 vars; `take` both; create 4 fresh vars + 4 cross nodes `(v0,v1), (v2,v3),(v0,v2),(v1,v3)`; link `B-tag(n0)~a1`, `B-tag(n1)~a2`, `A-tag(n2)~b1`, `A-tag(n3)~b2`.
- `interact_oper(a,b)`: `a` must be `NUM`; `take(b) → (b1, enter(b2))`; if `b1` is `NUM`, `Numb::operate` and link result; else allocate partial-op node and link `b1 ~ OPR(n0)`.
- `interact_swit(a,b)`: `av = u24(a)`; `take(b) → (b1,b2)`; if `av==0` build `(b2, ERA)` linked via `CON`; else build `(ERA, CON(NUM(av-1), b2))` — Peano-style case split.
- `interact(net,book)`: pop redex, `get_rule`, handle `REF~ROOT` override + `should_swap`, dispatch, on failure push back, on success bump `itrs` (except `LINK`). `evaluator` loops until rbag empty, then `itrs.fetch_add`.

6. PARSING, STRINGIFYING, SERIALIZING

- IC32 parsing (README): two-phase global linking — `lcs` (name→loc)
  - `vrs` (name→var term); `parse_var/lam/app/sup/dup` record uses, then backpatch. Handles forward refs like `λt.((t x) λx.λy.y)`.
- IC32 stringifying: pre-pass assigns ids to binders + dedups floating `DUP`s, then emits `! &{x0 x1} = tᵢ` preamble + body — dups never nest inside ASTs on the wire.
- HVM2 `Book::to_buffer`: `u32` def count, then per-def `fid/name[256]/safe/rbag-len/node-len/vars-len/root/rbag/node` — the exact image Bend's backend loads before `CALL`.

7. PERFORMANCE READING

- Zero-alloc: `LINK/VOID/ANNI(fast path)/ERAS`.
- One-node: `OPER/SWIT`.
- Heavy: `COMM` (4+4), `CALL` (def-size copy).
- Contention points: `vars_exchange` in `link` / `enter`, allocator cursor `nput/vput`. Keep nets linear, dups labelled, safe defs dup-free to stay on the fast path.

8. REFERENCES

- HVM2 `src/hvm.rs` ll. types → `test_f24` (1207 lines, 35.4 KB).
- IC README §§ IC32, Parsing IC32, Stringifying IC32, pseudocode `app_lam/app_sup/dup_lam/dup_sup/whnf/normal`.
- HVM1 HOW.md §§ Rewrite Rules, Low-level Implementation (`runtime/base/memory.rs` pointer).

F.22 · bend-runtime/knowledge/03-hvm-inet-theory/05-bend-compilation-fuel.md

05 — Bend Compilation to HVM Nets, Fuel, and Parallel Execution

Sources fetched 2026-09-28:

- Interaction Calculus (https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus))
- HVM1 HOW.md (Core Language, Clones, Bonus: Copatterns/Beta-Optimality)
- HVM2 `src/hvm.rs` (`Def.safe`, `RBag`, `TMem::evaluator`, `get_resources`, par-flag on `REF`)
- Wikipedia "Interaction nets" (locality/linearity/confluence)

1. WHERE BEND SITS IN THE STACK

```
Bend (high-level: ADTs, match, folds, comprehensions)
  ↓  elaboration + affine analysis + dup insertion
HVM core (λ + CON + DUP + NUM/OPR/SWI + REF globals)
  ↓  Book { defs } serialization (to_buffer image)
INET runtime (GNet node/vars buffers + RBag + TMem workers)
```

Bend is the ergonomic surface; HVM is the net assembler; `hvm.rs` is the parallel graph machine. Every Bend function becomes one `Def` (`name/safe/root/rbag/node/vars`); every call becomes `REF~arg` (`CALL`); every `match` becomes `SWI` / `OPER` cascades; every shared variable becomes an explicit floating `DUP`.

2. ELABORATION: FROM SUGAR TO NETS

1. **ADTs → CON tags.** `data List = Nil | Cons(head, tail)` assigns consecutive `CON` ids; nullary `Nil` is `(Nil)` — matches HOW.md "parentheses can be omitted from unary constructors".
2. **Functions → REF globals.** `(Map f Nil) = Nil` style equations (HOW.md § User-defined Rules) compile to a `Def` whose `root` is the RHS tree and whose `rbag` holds pattern-match redexes. Currying is preserved: `(f a b c)` = nested applications; constructors vs applications distinguished by case (`(F x)` vs `(f x)`).
3. **Match → SWI + OPR + nested CALL.** Numeric_match lowers to `SWI` chains (zero vs succ with `NUM(av-1)` rebuild, cf. `interact_swit`); structural match lowers to `CON–CON` annihilation attempts plus fallback `COMM` for polymorphic projections.
4. ** lambdas → LAM/CON nodes.** Bend closures are binary agents; `APP-LAM` is β-reduction (`x<-arg; body`) — 200M+ β/s per core per HOW.md because substitution is 1–2 array writes (each var occurs once).
5. **Copatterns allowed.** Because functions and constructors share the agent representation, Bend can express HOW.md-style corecursion (`Head (Map f xs) = …`) — eliminator/introducer roles swap freely.

3. AFFINENESS AND `DUP` INSERTION (THE BORROW-CHECKER REPLACEMENT)

The pre-processor enforces the "every runtime variable occurs once" invariant:

```
(Foo a) = (+ a a)
  ↓
(Foo a) = dup x y = a; (+ x y)
```

- Use-count 0 → `ERA` (`*`); use-count 1 → direct wire; use-count N → chain of `DUP`s with fresh labels.
- `dup`s float on the global scope, never nested inside terms on the wire (IC32 stringify invariant) — same reason `hvm.rs` stores dup bodies via `CX/CY` vars pointing at the floating node.
- `Def.safe` is set iff the body contains no `DUP`. Safe globals take the `CALL→ERAS` shortcut when cloned (`interact_call` early return) instead of faulting; unsafe globals cloned by `DUP` abort unless a `--unsafe` /EAL pass has split light vs full lambdas (HVM1 FAQ plan).

Bend authors should therefore: keep hot functions linear (move, don't clone, large structures); hoist shared lambdas (`Not` / `Inc` lifting in HOW.md) so normal forms stay small; avoid `(g g)` self-clone shapes — eta-split into `(g h)`.

4. LABELS, FUSION, AND KEEPING NORMAL FORMS SMALL

Labels (`Lab`, 29 bits in HVM2; `SP0–7/CX0–7/CY0–7` in IC32) decide `DUP-SUP`: equal → annihilate (2 substitutions, zero alloc); different → commute (nested dups + 2 superpositions). Bend's labeller:

- Assigns distinct labels per source-level `dup` site so independent copies annihilate promptly.
- Threads the same label through copies of one value so they meet and cancel rather than explode into `COMM` storms.
- Powers runtime deforestation: `APP-SUP` pushes `( &L{a,b} c )` to `&L{(a c0),(b c1)}`, so `map f . map g` fuses even when `funcs` is dynamic (`foldr (.) id funcs` — HOW.md). Prefer Church/Scott lists in pipelines to maximise this.

Collapse rules (`SUP-LAM`, `SUP-APP`, `DUP-VAR`, `DUP-APP`, `ERA-*`) project intermediate superpositions back to pure terms before I/O.

5. FUEL: BOUNDING REDUCTION

HVM reduction is Turing-complete (Ω-loop exists), so Bend exposes fuel:

- **Interaction count** (`GNet.itrs: AtomicU64 + TMem.itrs: u32`, flushed via `fetch_add` in `evaluator`): `bend run --fuel N` stops after N non-`LINK` interactions. Used for gas metering, termination testing, and differential benchmarking (interactions ≫ wall-clock for asymptotic claims).
- **Allocation quota** (`get_resources(need_rbag, need_node, need_vars)` admission): each `interact_*` declares its maximum footprint (`COMM` =4/4, `CALL` =def-size); on exhaustion the redex is re-queued and the worker grows/yields instead of corrupting the net. Bend maps OOM to "increase heap" rather than unsound rewrite.
- **Tick/depth guard** (`TMem.tick`): watchdog for diverging `SWI` loops (e.g. `lop` / `fun` demo in `hvm.rs` comments); Bend's `timeout` layers on top.
- **Static fuel**: linearity/EAL analysis rejects or rewrites clone-of-clone shapes at compile time, so fuel is rarely consumed by pathological sharing.

Debugging recipe: `RBag.show/GNet.show/Book.show` dumps + `itrs` counter bisect superlinear blowups to a specific `COMM` site; relabel or linearise that site.

6. PARALLEL EXECUTION: WHAT BEND GETS FOR FREE

- **No annotations.** Any two entries in `RBag.lo/hi` are disjoint redexes by the diamond property; the thread pool (`tid/tids`, `nput/vput` striping, work-stealing `pop_redex`) scales them across cores. Radix-sort-style tree merges speed up; bubble-sort-style folds do not — algorithm, not runtime, decides.
- **Two sync points only** (HOW.md FAQ): dup-node locks (threads passing through the same lazy clone) and atomic substitution (`vars_exchange` moving ownership across threads). Everything else in `reduce()` is thread-safe.
- **Priority matters:** `is_high_priority` keeps `LINK/CALL/VOID` on `hi` so administrative steps don't head-of-line-block `ANNI/COMM`. Bend's scheduler inherits this: spawn-heavy programs (many `CALL`s) drain fast and expose width.
- **`REF` par-flag** (`set_par_flag/get_par_flag`, bit 28): marks calls safe to steal across workers. Bend sets it on `safe` defs and data-parallel maps; clears it around `OPER` on shared `NUM` cells where ordering matters.

7. NUMBERS AND EFFECTS

- `U24/I24/F24` via `Numb` + `OPR` partial application + `SWI` case split cover Bend's `u24/i24/f24` primitives; wrapping/saturating semantics match `operate` / `cast` (see note 04 + `test_f24`).
- Purity preserved: no `amb` agent, no global mutation — `SWI` / `OPER` are deterministic. Bend effects (IO, random) sit *outside* the net: `main` reduces to a value tree, then the driver interprets it.

8. AUTHORING CHECKLIST FOR FAST BEND

1. Favour linear data flow; `dup` only cold values.
2. λ-lift shared prefixes (`case_e/case_o/case_i` pattern) to keep compositions constant-space.
3. Use distinct `dup` sites for independent copies; share labels only for true siblings.
4. Prefer `match` on constructors over `SWI` on numbers in hot loops (`ANNI` < `SWIT` in cost).
5. Mark leaf helpers `safe` (no clone) to unlock `CALL→ERAS`.
6. Measure in interactions (`itrs`), not just ms, when claiming optimality wins.

9. REFERENCES

- Bend → HVM lowering: HOW.md Core Language + Rewrite Rules + Copatterns.
- Runtime: HVM2 `src/hvm.rs` (`Def/Book`, `RBag`, `GNet/TMem`, `interact_*`, `evaluator`, `to_buffer`, par-flag).

- Theory: notes 01–04 in this directory; IC README §§ Specification, Collapsing, DUP Permutations, Labeled Lambdas, IC32.

F.23 · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/00-overview-architecture.md`

00 — BendRT Overview & Architecture (Bend2 runtime)

Sources (fetched 2026-09-28): `bend guide` (Bend 2.0.25, "Under the Hood"

- "Tooling" sections); `bend guide shaders` ("The machine", "A frame"); `bend guide effects` ("An effect def", "Compatibility"); GitHub `bendlang/bend` README (References: `paper/BendRT.pdf` , `paper/BendTT.pdf` , `demos/` , `bend2/base.bend`); `demos/` listing (16 demos incl. `pure_hvm5_mini` , `pure_par_sort` , `app_slash_boss_3d` , `proof_insertion_sort`); `paper/` listing (`BendRT.pdf` , `BendTT.pdf`). Local: `bend2-knowledge/04-runtime-bendrt-hvm5/bendrt-architecture.md` (seed, 8 lines — expanded here).

1. WHAT BENDRT IS

BendRT is the Bend2 parallel runtime. The Guide's "Under the Hood" states the design in three sentences:

1. **One C file = CPU program + GPU kernel.** The compiler (`bend2/comp.ts`) emits a single C file. `clang` compiles it for the CPU; Metal (Apple) or CUDA (NVIDIA) compiles *the same file* for the GPU. A `!` call runs that code on whichever chip it lands on.
2. **A term is one 64-bit word.** Small values inline; everything else is a pointer into a single heap shared by every core and the GPU (see `01-memory-model-no-gc.md`).
3. **No GC, no C stack.** Affinity frees on `match` ; only `+` values carry a reference count. Each def compiles to a segment of a flat state machine; a call is a jump; a parallel call creates a join task plus one task per call (see `02-schedulers-cpu-gpu.md`).

`paper/BendRT.pdf` holds the full design and benchmarks; `paper/BendTT.pdf` holds the affine dependent type theory. The compiler itself is ~99% AI-written and unaudited (README Limitations); the *kernel* behind `--verdict` is mechanized separately in `bend2/bendtt.lean` (lags `bend.ts`).

2. COMPONENT MAP (REPO LAYOUT)

Piece	Path in `bendlang/bend`	Role
Language front-end	`bend2/bend.ts`	Human-written sources of truth — **do not edit**
Compiler + emitters	`bend2/comp.ts`	Check, optimize, emit C / Metal / CUDA / JS
CLI + module loader	`bend2/main.ts`	`bend` command, Bun/Node `.bend` import preload
Theory kernel	`bend2/bendtt.lean`	Mechanized kernel for `--verdict` (lags `bend.ts`)
Prelude	`bend2/base.bend`	`Base` library (`bend base` prints it)
Effects (C twins)	`bend2/effs/*.c`	Reference foreign-effect implementations
Effects (JS twins)	`bend2/effs/*.js`	Same, for `bend x.bend and -o x.js`
Papers	`paper/BendRT.pdf` , `paper/BendTT.pdf`	Runtime design; type theory
Executable specs	`demos/` (16 dirs)	Apps, servers, proofs, each with `LAWS.bend`
Guide source	`guide/GUIDE.md`	Printed by `bend guide`

Targets today: **C, Metal, CUDA, JavaScript**. Planned: Lua, Luau, Python. One GPU per program, one event loop, no multi-machine execution yet (README Limitations). One C file per program: no separate compilation, no incremental builds; native compile is slow (clang/CUDA/Metal), so use the JS target for fast iteration.

3. THEORY WALL (WHY THE RUNTIME CAN BE THIS SIMPLE)

BendTT has **one universe** (`Type : Type`) and **no positivity check** — a datatype may recurse left of an arrow. Consistency comes from a checking wall (Guide, "Under the Hood"):

- **Live** code runs: must terminate (structural recursion, smaller part of a matched input; mutual recursion rejected; `@unsafe / def f?(..)` opts out and forfeits proof guarantees).
- **Dead** code (types, erased `-x` args, equations/proofs) may loop or inhabit `Empty` , but dead evidence never counts as live evidence.
- Erased variables exist only for checker/proofs; the compiler deletes them → zero runtime values (memory-file consequence).

This is why the runtime needs no runtime type tags for erased content and no termination fallback at execution: termination was already proven (or explicitly waived with `@unsafe`).

4. EXECUTION SHAPE (PREVIEW OF FILES 01–05)

- **Memory** (`01`): 64-bit words; unified Apple heap (zero-copy) vs CUDA managed/paged heap (PCIe faults both ways — shaders guide reports 17–19 FPS on RTX 4090 for a frame an M4 draws in 8 ms when a whole frame crosses the bus every turn).
- **Scheduling** (`02`): fixed grid of rings, tasks assigned once, **no work stealing**; programmer promises balanced forks; `!` marks the GPU subtree; JS runs strictly sequentially.
- **Effects** (`03`): only the event loop touches host C/JS; proofs, termination, GPU never do; determinism rules isolate consensus `transition` from RNG/time/net/foreign code.
- **HVM5-mini** (`04`): `demos/pure_hvm5_mini` (`main.bend` , `LAWS.bend` , `PROOF.bend`) is the only in-repo `HVM5` artifact — a pure Interaction Calculus evaluator written in Bend itself; use as executable spec sketch.
- **Laws gate** (`05`): every demo pairs `LAWS.bend` (human-owned claims) with `PROOF.bend` (AI-written defs); `bend PROOF.bend` fails while any law is open/false — the commit gate.

5. REFERENCE WORKLOAD (ANCHORS)

- **Bitonic sorter** (`demos/pure_par_sort/main.bend`): `sort!(16n, …)` over a `Tree(16n)` of 2^{16} numbers; Guide-era numbers 12.15 s (1-thread M3 Max) → 0.96 s (16-thread) → 0.21 s (RTX 4090). Quote BendRT numbers only with revision + date.
- **Rasterizer** (`demos/app_slash_boss_3d/bend3d.bend` , via shaders guide): 1920×1200 @ 120 FPS (8.0 ms/frame) on a 10-core M4 mini; one bang per frame, fork-4 × 7 levels = 16 384 tiles = the lane cube; host builds the scene (1.6 ms), device draws (~6 ms), host frees last scene (0.35 ms).

6. HOW TO USE THIS DIRECTORY

Read in order: `00` (this file) → `01` memory → `02` schedulers → `03` FFI/determinism → `04` HVM5-mini port notes → `05` laws/proof/CLI. For spec work, map each runtime fact to `08-roadmap-hvm5-spec/spec-todo.md` (API, domain types, metering, fork/upgrade hooks) and keep consensus state a `Data` tree with total, law-checked migration functions.

SOURCES

- `bend guide` output, Bend 2.0.25, fetched 2026-09-28 (esp. "Under the Hood", "Tooling", "Limitations").
- `bend guide shaders` , fetched 2026-09-28 ("The machine", "A frame", "Cost model").
- `bend guide effects` , fetched 2026-09-28 ("An effect def", "Compatibility").
- https://github.com/bendlang/bend `(https://github.com/bendlang/bend)` (README References + Limitations), fetched 2026-09-28.
- https://github.com/bendlang/bend/tree/main/demos `(https://github.com/bendlang/bend/tree/main/demos)` (16-demo listing), fetched 2026-09-28.
- https://github.com/bendlang/bend/tree/main/paper `(https://github.com/bendlang/bend/tree/main/paper)` (`BendRT.pdf` , `BendTT.pdf` listing), fetched 2026-09-28.

F.24 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/01-memory-model-no-gc.md

01 — Memory Model: 64-bit Words, Unified Heap, No GC

Sources (fetched 2026-09-28): `bend guide` ("Under the Hood": "A term is one 64-bit word… There is no garbage collector…"; "Quantities", "Arrays", "Kinds" sections); `bend guide shaders` ("Memory…", "Data", "Ownership" sections); `bend guide effects` (C-side `Term` constructors); `bend base --types` (datatype/kind listing); seed `memory-64bit-heap-no-gc.md`. Historical contrast: HVM2 64-bit word note in seed (`schedulers-cpu-gpu.md` era: every port 32 bits, binary node = 2 ports).

1. THE WORD

A BendRT term is **one 64-bit word** (Guide, "Under the Hood"):

- **Small values inline** — numbers (`U32` as 32-bit word payload, `Nat` literals, `F32` bits), narrow constructors.
- **Everything else is a pointer** into a single heap shared by every core and the GPU.

Historical HVM2 encoding (for contrast only, not the current ABI): every port 32 bits; a binary node stores 2 ports (main implicit by nesting) → one 64-bit word → atomic ops. Tag decides the reading: heap address (CON/DUP/OPR/SWI), vfn address (REF), unboxed 29-bit NUM, var name (VAR), 0 (ERA). NUM/ERA/VAR inline; others indirect; REF fast-copies unless it contains DUPs. The IC32 variant used 1-bit sub + 5-bit tag + 26-bit val with CX/CY float-dups. **Do not bake these layouts into a new spec** — they document the lineage; BendRT's word packs constructors, `U32` / `F32` words, and heap pointers per `comp.ts`, verified via emitted C (`bend x.bend -o x.c`), not via HVM2 notes.

2. ONE HEAP, TWO REALITIES

- **Apple (unified memory):** CPU↔GPU transfer is zero-cost. The shaders guide's reference rasterizer (`demos/app_slash_boss_3d`) depends on this: host builds per-frame cell lists, one bang reads them on the GPU.
- **CUDA (managed memory):** each page faults across PCIe on first touch by the other side. Shaders guide reports 17–19 FPS on an RTX 4090 (25 ms `gpu_pass` + ~21 ms host turn) for the frame an M4 draws in 8 ms — caused by building on the host and reading on the device every frame. **Spec rule: keep a whole frame on one side of the bus per turn.**
- **JS target:** single heap in JS semantics, sequential only.
- Cap the GPU heap explicitly: `./file --gpu 4GB`; disable the GPU with `./file --gpu off` (bangs run on the CPU pool instead).

3. NO TRACING GC — AFFINITY + MATCH-FREE + COUNTED +

There is **no garbage collector** (Guide). Three mechanisms replace it:

1. **Affinity (default).** Variables are used *at most once*. Dropping an affine value is always free. A `match` **frees the node it opens on the spot** — deallocation is structural and immediate.
2. **`+` (reusable) pays a reference count at runtime.** `+x` requires the type to be `Data` (`is Data` declares copyability; `is Type` keeps values linear). Matching a `+` value hands out `+` fields; on a plain value, `+r` in the pattern makes one field reusable. `main` marks nothing — you write `+` where copies are needed and the compiler inserts share/borrow.
3. **Types / erased args / proofs → zero runtime values.** `-A: Data` parameters, `Kind` / `Quant` witnesses, and all equation proofs are deleted by the compiler. They cost memory only in the checker.

Kinds recap (`bend base --types`, Guide "Kinds"): `Type = Kind(&1)`, `Data = Kind(&2)`; `type List<a, -A: Kind(a)> is Kind(a)` — a list is exactly as reusable as its elements. Functions, arrays, IO handles are `Type`: never copied. Quantities combine with `a <&> b` (minimum).

4. ARRAYS: IN-PLACE MUTATION WITHOUT LOSING PURITY

`Array<T>` is a `Type` — exactly one owner at all times (Guide "Arrays"):

- `a = [0 : U32*8n]` — 8 slots; count after `*` is a power of two (`[0 : U32^3n]` names depth instead).
- `a[5] <- 42` rewrites **in place**, hands back the same array (re-bind: `a = a[5] <- 42`; as last statement it *is* the written array).
- `a[5]` **reads by returning array + element** (else the array would be gone). Indexes wrap. `Array.get` / `Array.swap` for non-`U32` elements, `Array.set` for writes, `Array.clone` for a second copy.
- **An `Array` has one owner, so it cannot go down a fork tree** (shaders guide, "Any scene"): build per-tile **lists** on the host, not shared arrays. Atomic arrays shared across threads are experimental, need `@unsafe`.

HVM5-mini applies the same discipline: its env/book are `Array<Cell>` / `Array<Tpl>` threaded linearly through `S{env, book, …}` states (see `04-hvm5-mini-reference-port.md`).

5. OWNERSHIP INFERENCE (WHAT + REALLY COSTS)

The **compiler decides borrows; `+` does not** (shaders guide, "Ownership"):

- The checker needs `+` for a second use. The compiler then borrows a boxed parameter (not `Array`) that the def only matches or passes to a borrower (`term_peek` in emitted C — no count), vs owns one it returns, stores, or passes to an owner.
- An owned use of a later-used value *shares* it: a count on the value and — sealing is per type — on every node of that type, silently.
- Measured costs on the M4 rasterizer: reading the scene through counts 8.0 → 16.5 ms/frame; two consumers of one list → 45 ms frames + 448 s Metal compile; device-owned scene 6.2 → 12.0 ms draw.
- **Verify in emitted C** (`bend x.bend -o x.c`): `term_peek` = borrow; `term_keep` / `ctr_take` / `rfc_seal` = counts. A new keep in a device def is a regression.

Rules of thumb: lists consumed once; wide records built where used (a 52-word record per list entry took draw 6.3 → 12.1 ms); pass constants as `~` template args, not parameters; typed `pick` / `word` over generic `Bool.pick`; `match` on `U32` literals costs (+21% host C, +6.4% `.gpu`).

6. CONSEQUENCES FOR A RUNTIME SPEC

- **Meter heap growth (`bits`) separately from steps (`mana`)** — cf. the Kindelia model in `07-smart-contracts-dynamic/`. Affinity makes peak-live words predictable; `+` counts and array clones are the metered events.
- Keep `transition` state a **`Data` tree**; migration functions total + law-checked (dead code never leaks into live state).
- Never let consensus state own GPU/CPU-crossing pages per turn; batch transfers or keep frames device-resident.
- Quote word layout from **emitted C + `comp.ts`**, never from HVM2 notes.

SOURCES

- `bend guide` (Bend 2.0.25), fetched 2026-09-28 — "Under the Hood", "Quantities", "Arrays", "Kinds", "Closures" (affine closures).
- `bend guide shaders`, fetched 2026-09-28 — "Memory…", "Data", "Ownership" (term_peek/term_keep audit rule).
- `bend guide effects`, fetched 2026-09-28 — C-side `Term` shapes (`term_pak`, `io_node`, `io_tup`).
- `bend base --types` output, fetched 2026-09-28 (Empty…Map kinds).
- https://github.com/bendlang/bend/tree/main/demos (`https://github.com/bendlang/bend/tree/main/demos`) (`pure_hvm5_mini` env/book arrays), fetched 2026-09-28.

F.25 · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/02-schedulers-cpu-gpu.md`

02 — Schedulers: CPU Threads + GPU (BendRT fork-join)

Sources (fetched 2026-09-28): `bend guide` ("Parallelism": parallel-call promises, contention-free binary fork-join, `!`, unified heap, JS sequential); `bend guide shaders` (full: "The machine", "A frame", "The code", "Cost model", "Measuring", "Do not"); GitHub demos `pure_par_sort/main.bend`, `pure_par_sum`, `app_slash_boss_3d`, `app_ray_tracer_3d` (listing); seed `schedulers-cpu-gpu.md`. Historical contrast: HVM2 work-stealing note in seed (active pairs, atomics, CUDA active-pair list) — superseded by BendRT's fixed grid.

1. THE PRIMITIVE: PARALLEL LET + `!`

```
a b = pow2(p) pow2(p)   # parallel call: one task per call + join task
result = pow2!(20n)     # `!`: this call + every parallel call inside → GPU
```

The Guide ("Parallelism") states the two promises a parallel call makes:

1. **The calls are independent** — always true: Bend is pure + affine.
2. **They run in roughly the same time** — *yours to keep.* One early finisher = sub-ideal speedup. Split 1M items as halves, never 1 vs 999 999.

`!` hands a subtree to the GPU when compiled native (Metal/CUDA if `--gpu`, else CPU threads); nested parallel calls stay on the device; one GPU per program. A machine with no GPU — or `--gpu off` — runs bangs on the CPU pool (still parallel). **The JS target ignores all of it and runs sequentially.** What lanes share sets the speed: a `+` value read by every lane costs an atomic per read — read `bend guide shaders` before writing a parallel app (Guide says so verbatim).

2. BENDRT SCHEDULER (CURRENT): FIXED GRID, NO WORK STEALING

- **Fixed grid of rings; work grows along rows, drains along columns; each task assigned once, never moved** (seed + shaders "The machine"). Contention-free and GPU-friendly — but no rebalancing. **Programmer promises balanced forks.**
- **No C call stack per Bend call.** Each def compiles to a flat state-machine segment; a call is a jump; a tail call is a loop; a sequential continuation is explicit; a parallel fork creates a join task. Pending work is explicit data → the same binary runs on CPU and GPU.
- A kernel iteration grows the fork tree breadth-first until the frontier fills the **lane cube (128×128 = 16 384 lanes)**, then every lane runs its task to the end (~0.4–0.5 ms + slowest lane). A continuation that forks again lands in the next iteration.
- **Flat def** = no parallel let, no bang, no closure apply, no non-flat call, no non-tail self-call → compiles to one native tail loop (`spin_N` in the C). Non-tail self-recursion becomes a segmented continuation (a frame per call) — fine on host, poison in a bang.
- HVM2 contrast (historical): active pairs as work items, wire atomics, light C procs / CUDA kernels, lock-free; DUP-traversal lock + atomic substitution as the only sync; CPU work-stealing pool; GPU over the active-pair list; any order valid (confluence). Divide-and-conquer parallelizes; chains don't. BendRT keeps the confluence insight, drops work stealing for determinism + GPU-friendliness.

3. REFERENCE WORKLOADS

3A. BITONIC SORT (`DEMOS/PURE_PAR_SORT/MAIN.BEND`)

Tree of 2^{16} numbers (`Tree(16n)` = nested `Sigma` pairs down to `Nat` leaves); `gen` fills descending; `sort` sorts halves in opposite directions then `flow` / `mix` merges leaf-by-leaf; **every fork is a parallel let**; `sort!(16n, …)` runs GPU-or-all-cores; `main` prints "sorted"/"unsorted" via `ascending`. Guide-era numbers: 12.15 s (1-thread M3 Max) → 0.96 s (16-thread) → 0.21 s (RTX 4090). **BendRT numbers must be quoted with revision + date** (seed rule) — the 2.0.25 checker/compiler moves fast. Its `LAWS.bend` (`mix_leaves`, `sort_leaves`) proves leaf-sum preservation — the scheduler story and the proof story in one demo (see `05`).

3B. RASTERIZER FRAME (`DEMOS/APP_SLASH_BOSS_3D`, VIA SHADERS GUIDE)

1920×1200 @ 120 FPS (**8.0 ms/frame**, 10-core M4 mini):

1. **Host** builds scene → quadtree of 64-px cells, one flat `Tris` cons list per cell (mesh/project/light/cull). ~1.6 ms, one parallel let per def.
2. **One bang**: fork-4 per node × 5 levels → cells, × 2 more over each cell's list → 16-px tiles: 4^7 = 16 384 = the lane cube. Each tile filters its cell list, walks it in flat loops (`Blk.go` over 4×4 squares, `Blk.four` → `Pix` / `Qua`). ~6 ms.
3. Last frame rides down as `Four` quadrants; tiles leave quadrants unused → runtime sinks them. 0 ms.
4. **Host** frees last scene as one branch of next build (`Cells.drop(4n)`). 0.35 ms.

Shape table: 2048-root → `Frame.node` k=5 → 64-px cells → `Cell.fork` k=2 → 16-px tiles → `Tile.b16 = 4×b8 = 4×b4` straight-line → `Blk.go` flat loop → `Blk.four` pixels. Off-frame/empty answers one `Pix` before forking (`Frame.sky`). Fold-per-level + fold-per-pixel alternatives measured 600 ms / 31 iterations vs 8 ms / 1 — the fork-tree shape *is* the performance.

4. COST MODEL (SHADERS GUIDE, M4 + FRAME UNLESS STATED)

- Host serial work (mesh/project/cull): 0.6–1.6 ms parallel vs 14–33 ms on GPU (a P-core ≈ 75× a lane on serial work) — **mesh on host, never GPU**.
- Aim **4^7 leaves per bang, one per lane**; sweep one fork level either side on-device (CPU pool is flat across that sweep, can't tune it). 256/16 384 busy measured 234 ms vs 28 ms at 4 096 leaves.
- Walk each list **once per 4×4 square** (6.6 → 5.6 ms vs per-pixel); x-only work once per column pair; per-lane early-out saves nothing while a SIMD sibling runs the body (throughput-, not tail-bound).
- Drops: `Image.drop` forked per `Qua` ~2.6 ms; `Image.free(7n)` (7 levels then sink) 1.6 ms; inside draw tree 0; whole image on one lane 19.8 ms.
- Host: one parallel let per def (a second costs an iteration: build 2.32 → 1.93 ms); each extra branch/level ~0.1 ms; sub-0.3 ms jobs lose to pool wake-up; a heavy branch serializes a 16-ring chunk — split it.

- Keep per-tile lists short: slivers/sub-pixel triangles pile 100–400 entries/tile (42 ms frames); exact triangle-vs-cell overlap holds worst tile at 80–110; tessellate ~20-px patches in screen space.
- CUDA bus rule (above): whole frame on one side per turn.

5. MEASURING (SHADERS "MEASURING" + "DO NOT")

Probe like the game: `SLASH_PROBE=1200` scripted headless frames, medians after warm-ups. Single-run medians drift 0.3–0.5 ms across days; second binary of a GPU pair runs ~7% faster → compare 3–4 interleaved 1200-frame pairs in both orders + an A-vs-A pair. Verify pixel-identical (`SLASH_DUMP=T`) before trusting a gain. Count keeps in the C + `.gpu` bytes (+6% was a real 1–4% slowdown). Probe without audio (`audiomxd` wakes at 70–100% of a core). The "Do not" list (11 items) is a review checklist: no counted scene reads, no fold-per-level, no non-tail bang recursion, no GPU meshing, no per-pixel forks, no slivers, no wide records, no U32-literal matches, no generic `Bool.pick` on words, no in-bang frees, no single-run trust.

6. SPEC CONSEQUENCES

- Require balanced-fork justification for every parallel let in consensus or proving code; unbalanced trees are a perf bug by construction.
- `!` placement is part of the design doc, with `--gpu off` fallback behavior stated (CPU pool) and JS-sequential behavior stated.
- One GPU / one event loop today: no multi-device or multi-machine claims.

SOURCES

- `bend guide` ("Parallelism"), Bend 2.0.25, fetched 2026-09-28.
- `bend guide shaders` (all sections), fetched 2026-09-28.
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/main.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/main.bend), fetched 2026-09-28.
- https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) (par_sum, ray_tracer, slash_boss listings), fetched 2026-09-28.

F.26 · bend-runtime/knowledge/04-runtime-bendrt-hvm5/03-ffi-determinism-io.md

03 — FFI, Effects, Determinism & IO

Sources (fetched 2026-09-28): `bend guide` ("IO and Concurrency", "Monads", "Apps" sections); `bend guide effects` (complete: C side, JS side, worked `Clock.now` example, Compatibility); `bend guide shaders` ("A frame": `Turn.go` host/bang interleaving, `IO.now` barrier); GitHub demos `io_hello_world` , `io_http_fetch` , `io_http_server` , `io_rollback_netcode` , `io_tcp_echos` , `app_pong_game_2d` (listing); seed `ffi-determinism.md` .

1. EFFECT DEFS: ONE BEND SIGNATURE, TWO HOST TWINS

```
def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"
```

Guide + effects note, condensed:

- **Naming:** host function = def name, lowercased, dots → underscores (`Clock.now` → `clock_now` ; `Window.frame` → `window_frame`). Every effect in `Base` is built this way; `bend2/effs/*.c + *.js` are the reference implementations.
- **C twin** serves `bend x.bend -o x` (native). The compiler splices your `.c` into the program's C source *after* the runtime — every runtime symbol is in scope. Define the effect + register in a constructor: `io_eff(CID_CLOCK_NOW, clock_now_run, need)` where `CID_X` is the def name uppercased (dots → underscores). `need` : `0` = run at once, `IO_READ` = park until handle `f[0]` readable, `IO_TIME` = park `f[0]` ms. Then the loop calls the effect.
- **JS twin** serves `bend x.bend` and `-o x.js` . Plain script in a closure; compiler finds the function by name. Needs become a second function (`clock_now_need` → `{read: true}` / `{time: true}`); blocking effects take `k` and park via `io_park_on` , answering later through `more()` . `io_sys()` is libc through `bun:ffi` .

- **Argument/return marshaling (C):** `U32` = word (`(u32)f[0]`); `String` via `io_cstr(e, f[0], &len)` (malloc'ed copy — free it); handles via `io_hand_v(f[0])`. Return: `U32` as `(Term)n`, `Unit` as `term_pak(CID_UNIT, 0)`, `String` via `io_str`, 2-field ctors via `io_node`, `Result` via `io_done` / `io_fail` (`text=NULL` prints `strerror(code)`; `io_sys_end` stores errno). Handles pack with `io_hand(v)` and are **always handed back beside the result, even on failure** (`io_tup(e, io_hand(h), r)`).
- **Marshaling (JS):** `U32` = number, `Nat` = BigInt, `String` = string, ctors = `{$: "Name", field: value}`, handles = host values; answers via `io_done(v)` / `io_fail(code)` / `io_tup(handle, result)` / `{$: "Unit"}`.
- **User handle types are WONTFIX for now** (`WONTFIX.txt`): new effects must reuse Base handle laws (`File`, `Socket`, …).
- **No ABI promise:** the C side tracks the exact compiler version; releases may rename internals — **rebuild effects with every update**.

Worked example (`Clock.now` in effects note): `main.bend` + `clock.c` + `clock.js`; `bend main.bend -o main && ./main`, `bend main.bend -o main.js && bun main.js`, and `bend main.bend` all print the line. Output name must not collide with an effect file. Reference patterns: `file_read.c` for `io_work` (helper-thread blocking: `call(w)` off-loop, `pack(e, w)` on-loop; `call` has no `Env`, touches only `w`; effect owns `w->data`), `tcp_recv.c` / `tcp_recv.js` for `io_wait_on` / `io_park_on` (fd parking, re-parkable `more`).

2. THE CONCURRENCY MODEL (ONE EVENT LOOP)

Guide "IO and Concurrency":

- A program = computations interleaved by **one event loop** (Node-like): each runs pure code (in parallel, every core) to its next effect; one waiting on socket/sleep/channel steps aside. Ends when all done, or reports **deadlock** when the rest all wait.
- `IO.fork` starts a computation → channel for its eventual result; `IO.join` waits. Underneath: `IO.spawn`, `Chan.new/send/recv/close`.
- `do IO<T>:` desugars binds to `IO.bind` / `IO.pure`; every bind annotated (`x : T <- effect`); `x : T = v` binds pure values mid-block; fallible effects answer `Result` → `IO.try` unwraps or exits, `IO.die` exits deliberately; `IO.args` = CLI args minus runtime options (`--` ends them).
- Handles are **affine opaque values**: every effect hands them back; no forging, no reuse. `do M<…>:` generalizes to any monad with `bind` + `pure` (`Maybe`, `Result`, own types).
- **Purity boundary (hard rule):** *only the event loop runs host code, so proofs, termination and the GPU never touch host code.* Reverse direction: JS may `import Game from "./game.bend"` (with `bend2/main.ts` preloaded) and call every non-IO def — ctors as `{$:…}`, `Nat` as BigInt, **`Array` crosses without a copy** (caller's own array, updated in place — copy first if you keep it).

3. IO × GPU INTERLEAVING (SHADERS "A FRAME")

`Turn.go` keeps `(oldImage, usedScene)`: build takes old scene, bang takes old image, with an **`IO.now()` step between them** — a bang following the host's forks *in one evaluation* runs on the CPU pool, so the barrier matters. `Window.frame(window, image)` presents. Audio via `Audio.open / write / close`; windows via `Window.open / frame / close`. Graphics stay pure: a frame is an `Image` quadtree (`Pix` / `Qua`); `App.run` calls pure `view` then effectful `tick` per frame until `tick` answers `None` (Guide "Apps"; full sample `demos/app_pong_game_2d`).

4. DETERMINISM RULE FOR CONTRACTS (D ON S)

Consensus `transition` sees **only** `(state, msg, height/hash/caller)`:

1. **No RNG / time / net / foreign C/JS inside** `transition`. Effects (`IO.now`, env, sockets, custom foreigns) live outside consensus — inputs, never in-transition calls. Proofs/termination/GPU never touch host code, so keep it that way by construction.
2. **Metered + bounded, revert-on-exceed** (Kindelia mana/size model): count steps *and* heap words; exceed → revert, never partial-commit.
3. **Canonical encodings** for states/txs/receipts (byte-exact; no pointer-dependent ordering, no float nondeterminism — note `F32` is axiomatic: *nothing about floats can be proven*, Guide Limitations).
4. **Rules immutable; upgrades = new hashes + registry pointer.** Old in-flight frames finish old rules; new calls use new rules (see `07-smart-contracts-dynamic/`).
5. Keep `transition` state a `Data` tree; migration functions total + law-checked; `F32` excluded from consensus hashing (or bit-normalized with explicit law).

JS-import caveat for off-chain relayers: imported `.bend` values cross without copy for `Array` — clone before retaining relayer-side buffers.

5. DEMO MAP (WHAT TO READ FOR WHAT)

Demo	Teaches
`io_hello_world`	Minimal `IO.print` program shape
`io_tcp_echos / io_http_server / io_http_fetch`	Socket/file parking patterns (`io_wait_on`)
`io_rollback_netcode`	Deterministic sim + IO boundary (model for contracts)
`app_pong_game_2d`	Complete `App.run` (view/tick, events, `Close{}` quit)
`app_slash_boss_3d`	Host-build → bang → present pipeline + probes

SOURCES

- `bend guide` ("IO and Concurrency", "Monads", "Apps"), Bend 2.0.25, fetched 2026-09-28.
- `bend guide effects` (C side, JS side, `Clock.now` example, Compatibility), fetched 2026-09-28.
- `bend guide shaders` ("A frame": `Turn.go` , `IO.now` barrier), fetched 2026-09-28.
- https://github.com/bendlang/bend/tree/main/demos `(https://github.com/bendlang/bend/tree/main/demos)` (io_* + app_*), fetched 2026-09-28.

F.27 · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/04-hvm5-mini-reference-port.md`

04 — HVM5-mini Reference & Port Notes (`demos/pure_hvm5_mini`)

> Sources (fetched 2026-09-28): raw `demos/pure_hvm5_mini/main.bend` (~1916 lines, header comment = port manual), `LAWS.bend` (8 laws), `PROOF.bend` (full proofs); `bend guide` ("Arrays": single-owner env/book discipline; "Under the Hood": no GC); seed `hvm5-mini-reference.md` (7 lines — expanded here). Status rule from seed: the **only** `HVM5` artifact in the live repo; a minimal **pure** runtime model in Bend itself — executable spec sketch, **not** a release. No standalone HVM5 repo/paper verified — do not cite one.

1. WHAT IT IS

`demos/pure_hvm5_mini/` = **a port of `hvm5.c`** : an Interaction Calculus evaluator with the same syntax, semantics, and interaction counting as the C reference — written in Bend, running on Bend's own memory:

- `main.bend` — types, parser, evaluator, instantiator, printer, driver.
- `LAWS.bend` — 8 claims about the pure logic under the evaluator.
- `PROOF.bend` — proofs for all 8 (eat-your-own-dogfood gate demo).

"Terms are Bend constructors, so term memory is Bend's own: erasing a term is dropping it" (header comment). A run of `#0/#1` is **one word** (24 bits + count, outermost bit lowest, as `hvm5.c` packs them); a run over `()` is the word alone (`Won`) — a bit chain costs no heap at all.

2. TERM SYNTAX (HEADER GRAMMAR)

```
Term ::= x | &{} | &L{a,b} | ! &L{a,b} = v; k | λx.f | λ- f | (f x)
       | () | λ() f | #0{x} | #1{x} | λ{#0:f; #1:g} | #(a,b) | λ! f
       | @ref | @ref(a,b,..)
```

Runtime `Term` type: `Var Co0 Co1 Era Sup Lam Drp Cal One Use Pak Won Tup Get Mat Ref` + normal-form-only `Abs Nam` . Template `Tpl` (parser output, binders as ids) adds `TDup/TDux` (both/one side live — copies turn them into cells), `TSup` , `TApp` , `TUse/TGet/TMat` , `TRef` . `Sh1/Sh2` describe 1-child / 2-child node shapes for the copier, eliminator-sup splits, and `inst` rebuilds.

3. MACHINE STATE

```
type S is Type:
 S{env: Array<Cell>, book: Array<Tpl>, m: Mode, k: Kont, +c: U32, +fh: U32, +nx: U32}
```

- **`env : Array<Cell>` by binder id.** Lambda cell holds its arg (`Sub`) once applied; dup cell holds the value until a side asks (`Sha` , or `Haf` when the twin is dead), then the twin's copy. A cell read by its variable joins the free list (stores old head as `Num` ; `fh` / `nx` ride the loop). `Cell ::= Sub Sha Haf Num Nul Ded` .
- **`book : Array<Tpl>` by definition index** (parser numbers names + binders as met). A reference **moves its template out, copies in place, writes back** — the copier only *reads* the template, so the compiler lends it and no node is rebuilt (borrow, not count — cf. file `01`).
- **`Mode`** : `Go{t}` (evaluate), `Kn{w}` (deliver whnf to top frame), `Sk{w}` (stuck, unwinding — `hvm5.c` dies here).
- **`Kont` stack** (as `hvm5.c` 's): `Arg` (pending arg), `Usk/Mtk/Gtk` (eliminators awaiting scrutinee), `Dpk{l,i,sd,fl}` (dup cell awaiting value; which side asks; twin alive?), `Rfk{r}` (deferred `@ref`), `NfK + N*` rebuild frames (strong normalization in the same loop).
- **`Out`** : `Fin{env,book,t,c}` (normal form + interactions), `Exp{… r}` (expand def `r`), `Dsp{… r, w}` (resolve deferred `@r` vs whnf `w`).
- **`c` = interaction count** — the same counter the C reference keeps; every helper answers the next `S` ; `run` matches stack × mode.

Arrays are `Type` (single owner) so `S` threads `env` / `book` linearly — the affordance Bend gives `transition` -style state machines for free.

4. EVALUATOR HIGHLIGHTS (PORT MANUAL, CONDENSED)

- **Collection (`free` / `free_run`):** dropped terms return cells to the free list (as `hvm5.c` 's `collect`). Lambda binder reads `Era` henceforth (its variable may live in a dup copy). Dropped dup side: twin sole (`Haf`), or free the parked copy, or mark `Ded` (value under evaluation — settling side frees the copy).
- **Instantiation (`inst` , phase `IM`):** copies a *borrowed* template over `fh/nx` , moving each binder's occurrence out of its cell. Under fusion (`FRoot` at a reference root, `FLam` under fused binders — `hvm5.c` 's `L_RF0` loop) leading lambdas/drops meet pending `Arg` frames instead of copying; a root constructor-eliminator meeting a non-bit defers to its scrutinee's whnf (`K_DISP` : copy answers scrutinee under `Rfk → IEnd`). `FOff` = no fusion (whole-template copies).
- **References (`ref_fin` , `dsp_*`):** `K_DISP` — deferred `@ref` meets whnf: a bit selects the branch to copy fused with the rest; any other whnf meets a copy of the whole template.
- **Interaction rules (`app/use/mat/get/dup/settle_*`):** β (`Lam`), erase-drop (`Drp` frees arg, counts), sup-duplication through fresh cells (`Sha`), eliminator-vs- `Sup` commutations (`eli_sup` , `mat_sup`), bit selection with loser-branch free (`mat_bit`), dup settling with twin-park/free (`settle_go` : left/right × dead/alive), `Pak` duplication charging `(p >> 24)` interactions.
- **Parser:** continuation monad in `do` -notation over `St{head-form, src, scope, next-id, names}` ; error prints and stops; lookahead-heavy forms resume as phases `Ph` (`Trm/PAt/PArg/PCls/PSep/PDup`). Names/labels are codes (6 bits/char, first highest); run words via `run_word/run_push/run_pop/bit0` ; `#b{x}` grows the run or starts a new one (`term_pak` , full at 24 bits).
- **Driver:** program = `src` constant (a Bend binary takes no args); output = `hvm5 -s` of the same source (booleans-as-bit-runs `@not` , Church-2 `@two` sharing via dup, `@main` applying `two(two(not))` to both bits at once). `λ` is code point 955, emitted as UTF-8 bytes `206,187` (`lam()`).

5. LAWS & PROOFS (THE GATE PATTERN AT FULL SCALE)

`LAWS.bend` (8 laws — lexer, cursor, printer, closed sanity facts):

Law	Claim
`word_joins`	word+rest re-form the source (lexer loses nothing)
`skip_is_drop`	blank/comment skip = `str_drop` by witness `n` (`exs`)
`pak_braces`	`Pak{p,x}` display opens `run_len(p)` more braces than `x`
`drop_drop` / `at_drop`	cursor composes / reads compose

Law	Claim
`run_push_pop`	push-then-pop round-trips (sanity)
`name_round_trip`	`name_code("main"/"F")` prints back (sanity)
`book_depth_fits`	book depth = least `d` with 2^d slots (sanity: 1→0, 5→3, 8→3, 9→4)

`PROOF.bend` proves all: `word_joins` by source induction (`go_cat` head lemma + IH rewrite); `skip_is_drop` by witness-building induction (`skip_step`: skipped head = 1 + rest witness); `pak_braces` via `opens_cat` (braces distribute over `++`), `bits_open_opens` (per-bit `pick_opens` induction), `bits_close_opens`, `add_zero`; cursor laws by source×position induction; sanity laws compute (`{==}`). Helper `law`s (`go_cat`, `skip_step`, `opens_cat`, `pick_opens`, …) are proven inline — the file is the style manual for proof-bearing runtime code.

6. HOW TO USE IT FOR SPEC WORK

1. Read the three files line-by-line (header comment first — it *is* the port manual).
2. Map each def to `08-roadmap-hvm5-spec/spec-todo.md`: API surface, domain types, metering (`c` counter → mana), fork/upgrade hooks (`Exp` / `Dsp` = the deferral points upgrades must preserve).
3. Treat it as **sketch, not release**: no perf claims, no wire format; the shipped runtime is `comp.ts` + papers, not this demo.
4. Copy its *discipline* (linear env/book, interaction counting, free-list threading, fusion-aware copying) into any new pure-model prototypes.

SOURCES

- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/main.bend (`https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/main.bend`), fetched 2026-09-28.
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/LAWS.bend (`https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/LAWS.bend`), fetched 2026-09-28.
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/PROOF.bend (`https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/PROOF.bend`), fetched 2026-09-28.
- `bend guide` ("Arrays", "Under the Hood"), Bend 2.0.25, 2026-09-28.

F.28 · `bend-runtime/knowledge/04-runtime-bendrt-hvm5/05-laws-proof-gate-cli.md`

05 — Laws, Proof Gate & CLI Workflow

Sources (fetched 2026-09-28): `bend guide` ("Laws and Proofs", "Tooling", "Syntax Reference" top-level/terms statements); local `bend-runtime/LAWS.bend` + `PROOF.bend` (double/sum/pow2 laws); GitHub raw `demos/proof_insertion_sort/LAWS.bend` (`sort_sorted`, `sort_perm`), `demos/pure_par_sort/LAWS.bend` (`mix_leaves`, `sort_leaves`), `demos/pure_hvm5_mini/LAWS.bend` + `PROOF.bend` (8-law suite); `bendlang/bend` README ("Bend BLOCKS mistakes", `AGENTS.md` snippet); `bend --help` + `bend base --types` (Bend 2.0.25). Workspace rule (`bend-runtime/AGENTS.md`, mirrors upstream `AGENTS.md`): run `bend guide` to learn; keep rules in `LAWS.bend`; run `bend PROOF.bend` before committing; parallelize whenever possible.

1. THE CONVENTION (README + GUIDE)

> "`LAWS.bend` is `AGENTS.md` backed by **proof**." — README

1. **Human** formalizes app rules in `LAWS.bend` (project root). The AI does not touch it.
2. **AI** writes code *and* `PROOF.bend`, which imports `LAWS.bend` and proves each law with a same-named def (`law sorted` ← `def Laws.sorted`). Bend refuses a `PROOF.bend` beside a `LAWS.bend` it doesn't import.
3. **Gate:** `bend PROOF.bend` fails while any law is open or false; prints "All terms check." only when every law holds. **Run it before every commit** (workspace `AGENTS.md`).

Demo proof: the "winning is impossible" game (`demos/app_win_is_bug_2d`, README lab) — without laws the wrap-around feature merged a bug; with `LAWS.bend` the AI had to retry (added a wall) until the law held.

2. LAW/PROOF SYNTAX (GUIDE "LAWS AND PROOFS" + "SYNTAX REFERENCE")

```
import Base

# CLAIM: for every nat x, x + 0 equals x.
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

# PROOF: induction on x — one rewrite (%) per step.
def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}
```

- `law name: + for x: T` params (`for -x` erased, `for +x` reusable), `for y: B where P(y)` (y arrives with its proof), `exs z: C` (proof returns a witness pair), then claim `T`. Propositions are types; proofs are defs of that type.
- `{a == b : T}` equality; `{==}` reflexivity (both sides compute equal). `match` refines the goal per case; recursion = induction hypothesis; `%e : P` rewrites with `e : {a == b}` (`_` marks `b` in `P`). `%e@E : P` named rewrite. `Equal.sym/trans/cong` in Base; `{a != b} = {a == b} -> Empty` (refute via motive to `Empty`, answer `Unit{}`).
- `?name` prints the goal; `?TODO` leaves it open; a law with no def is an **open claim** (gate fails). Failed steps print expected vs observed.
- `F32` is axiomatic: **nothing about floats can be proven** — keep floats out of laws (Guide Limitations).
- Cross-file: an open law in one file is filled as `def M.name(..)` in another — proofs ship separately from claims.

3. LAW CATALOG (WHAT GOOD LAWS LOOK LIKE)

Local (`bend-runtime/LAWS.bend`): `double_zero`, `double_succ`, `double_is_nat_double` (our `App.double = Nat.double` everywhere), `sum_nil`, `pow2_zero` — proved in `PROOF.bend` by `{==}` + one induction (`%Laws.double_is_nat_double(p)` rewrite). Minimal template for new code.

`proof_insertion_sort`: `sort_sorted` (`Sorted(sort(xs))` — output property) + `sort_perm` (count-based permutation: every `x` occurs in `sort(xs)` as often as in `xs`). The two-law shape every sorter needs.

`pure_par_sort`: `mix_leaves` (lesser+greater leaves conserve the sum)

- `sort_leaves` (sort preserves leaf sum, any direction). Scheduler demo *and* conservation-law demo — mirror for any fork-join balance claim.

`pure_hvm5_mini` (8 laws): lexer/cursor/printer + sanity suite (see `04`, §5) — the pattern for runtime-metatheory laws.

LAW style guide: state *over all inputs* (`for` binders), name the observable (`Sorted`, `count`, `sum`, `word_cat`), keep witnesses explicit (`exs n` for cursors), keep floats out, keep each law one idea.

4. CLI WORKFLOW (BEND 2.0.25, `BEND --HELP` + GUIDE "TOOLING")

```
bend guide                    # learn (start here; then `guide shaders|effects`)
bend base [--types|<name>]    # prelude: all / types-only / one name + subnames
bend file.bend --check-only   # check file + imports; run nothing
bend file.bend                # check, then run main (IO compiled; value normalized+printed)
bend PROOF.bend               # ★ commit gate: every law proven or it fails
bend file.bend -o file        # native binary (clang 14+; 19+ with `!`; needs file.gpu beside it)
bend file.bend -o file.c      # emit C (audit borrows: term_peek vs term_keep/ctr_take/rfc_seal)
bend file.bend -o file.js     # emit JS (sequential; fast iteration)
bend page.html -o dist        # bundle web page importing .bend files
./file --threads 8            # run on 8 CPU threads
./file --gpu off              # bangs on CPU pool (default: GPU on)
./file --gpu 4GB              # cap GPU heap
bend main.bend --publish      # upload file + imports; prints 0x<hash> import line
```

Notes: no-`main` file just checks; value-`main` normalizes via checker (slow for big work); binary with `!` also builds `file.gpu` (Metal on macOS; CUDA 12 at `/usr/local/cuda` on Linux; X11/ALSA headers for Window/Audio on Linux); no Windows (WSL works); needs clang 14+. This `bend` ignores project `bunfig.toml` / `.env`; `bun bend2/main.ts` from a checkout reads them — check untrusted code with `bend` (README).

5. COMMIT CHECKLIST (WORKSPACE LAW)

- <input disabled="" type="checkbox"> `bend guide` read (at minimum "Laws and Proofs" + target-relevant `shaders` / `effects` note) before writing Bend code.
- <input disabled="" type="checkbox"> New behavior has a `LAWS.bend` entry (human-owned wording).
- <input disabled="" type="checkbox"> `PROOF.bend` fills every law; `bend PROOF.bend` → "All terms check."
- <input disabled="" type="checkbox"> Parallel lets justified balanced (file `02`); `!` placement stated with `--gpu off` + JS fallback behavior.
- <input disabled="" type="checkbox"> New `+` uses audited in emitted C (file `01`, §5).
- <input disabled="" type="checkbox"> No host/RNG/time/net inside consensus `transition` (file `03`, §4).
- <input disabled="" type="checkbox"> Parallelize wherever the fork is balanced (AGENTS.md rule).

SOURCES

- `bend guide` (Bend 2.0.25), fetched 2026-09-28 — "Laws and Proofs", "Tooling", "Syntax Reference".
- Local `bend-runtime/LAWS.bend` + `PROOF.bend`, read 2026-09-28.
- https://raw.githubusercontent.com/bendlang/bend/main/demos/proof_insertion_sort/LAWS.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/proof_insertion_sort/LAWS.bend), fetched 2026-09-28.
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/LAWS.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/LAWS.bend), fetched 2026-09-28.
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/LAWS.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/LAWS.bend)
  - `PROOF.bend`, fetched 2026-09-28.
- https://github.com/bendlang/bend (https://github.com/bendlang/bend) (LAWS.bend game demo, AGENTS.md), fetched 2026-09-28.

F.29 · `bend-runtime/knowledge/05-ecosystem/bendlang-bend-repo.md`

bendlang/bend — Live Bend2 Repo Survey

Survey date (UTC): 2026-09-28. Sources: https://github.com/bendlang/bend (https://github.com/bendlang/bend), https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) (Bend2 stub), https://github.com/VictorTaelin (https://github.com/VictorTaelin), https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md).

1. REPO IDENTITY

- URL: https://github.com/bendlang/bend (https://github.com/bendlang/bend) — "Bend 2: a fast language that blocks AI mistakes via proof."
- Tagline (README hero): post-AGI intent language — laws > natural language, proofs verify AI code, fast compiler runs it at peak compute.
- Scale (2026-09-28): 23.0k★, 707 forks, 125 watching, 42 issues, 19 PRs, 3,088 commits on `main`.
- License: Apache-2.0 (`LICENSE`).
- Topics: `bend cuda dependent-types gpu metal parallel programming-language proof theorem-proving`.
- Install: `curl -fsSL https://bend-lang.com/install.sh | sh`.
- Active Bend2 moved here: https://github.com/HigherOrderCO/Bend2 (https://github.com/HigherOrderCO/Bend2) is stub (10★, 0 code) stating "Bend 2 moved to github.com/bendlang/bend" — always cite `bendlang/bend`.
- Pitch triple (README GIFs): runs FAST (vs C/TS/Lean, 1-core/16-core/GPU), checks FAST (vs Isabelle/Agda/Lean/Rocq), is PARALLEL (`pow2(20)` over 4,096 GPU cores), BLOCKS mistakes (LAWS+PROOF).

2. TOP-LEVEL LAYOUT (AS RENDERED 2026-09-28)

```
bendlang/bend/
  .github/ISSUE_TEMPLATE/
  bench/            # runtime/ + checker/ — charts backing media/*.gif
  bend2/            # compiler: base.bend bend.lean bend.ts comp.ts main.ts docs/ effs/ pack/
  demos/            # 16 demos, each with LAWS.bend + PROOF + main pattern
  evals/            # evaluation harness
  gates/            # acceptance gates
  guide/            # GUIDE.md EFFECTS.md SHADERS.md (also via `bend guide`)
  media/            # hero.gif runtime.gif checker.gif parallel.gif game_*.gif
  paper/            # BendTT.pdf BendRT.pdf
  tests/            # test suite
  tools/bend-fmt-lsp/  # formatting-only language server
  AGENTS.md CHANGELOG.md WONTFIX.txt LICENSE README.md
  flake.nix .gitattributes .gitignore
```

File-nav notes: README + Apache-2.0 license blocks; Activity/Custom properties; Resources/Stars/Watchers/Forks; Releases/Packages/Used-by/Contributors/Languages sections exist but empty via webfetch — use `gh` locally.

3. BEND2/ — COMPILER CORE

- `bend2/bend.ts` — language core. Human-written. DO NOT EDIT (agent rule). Source of truth for syntax/types.
- `bend2/comp.ts` — compiler + runtimes (C/Metal/CUDA/JS codegen, parallelism lowering).
- `bend2/main.ts` — CLI + Bun/Node loader. `bun bend2/main.ts` from checkout respects `bunfig.toml` / `.env`; installed `bend` ignores them (security note in README — audit untrusted code with installed binary).
- `bend2/base.bend` — base library. Printed by `bend base`. Small by design (see Limitations).
- `bend2/bend.lean` (`bendtt.lean` in References) — kernel of `--verdict` + proofs, in Lean. Proven checker path vs unaudited compiler path (see §9).
- `bend2/docs/` — compiler docs (enumerate by cloning; webfetch hides file list).
- `bend2/effs/` — effect definitions (print/env/time/sleep/spawn/channels/files/TCP/UDP surface).
- `bend2/pack/` — hash packages. No names/versions/accounts/search yet — hub is content-hash only.
- Commands: `bend guide` (print guide), `bend base` (print base), `bend PROOF.bend` (verify laws), `bend run/check/build` (see `main.ts --help` locally).
- Agent contract (`AGENTS.md`, also recommended for user `AGENTS.md`):

```
When using Bend:
- run `bend guide` to learn it
- use `LAWS.bend` to keep important rules
- run `bend PROOF.bend` before committing
- parallelize the code whenever possible
```

4. DEMOS/ — 16 DEMOS, LAWS+PROOF+MAIN PATTERN

- Each demo: `LAWS.bend` (rules) + `PROOF.bend` (AI-written proofs) + main app code.
- Canonical example (README): game with law "winning is impossible":

```
# LAWS.bend
law you_cant_win:                          # "winning is impossible"
  for moves: List<Game.Move>               # any sequence of moves
  board = Game.replay(Game.start(), moves)  # replayed from the start
  {Game.is_won(board) == False{} : Bool}    # never leads to victory
```

```
# PROOF.bend
def Laws.you_cant_win(moves):
  # ... written by the AI
```

- Workflow: (1) formalize rules in `LAWS.bend` (e.g. sum balances zero, no wall pass-through, `list_sort` ascending, `array_set` in-bounds, winning impossible); (2) `bend PROOF.bend` after edits; (3) compiler guarantees laws hold — retry until green. Without LAWS, wrap-around bug merged; with LAWS, AI added wall (or moves flag/kills room — any fix preserving law).
- Media: `game_law.gif` (bump wall) / `game_bug.gif` (wrap takes flag) / `game_law_kept.gif` (far wall).
- Interactive: https://bend-lang.com/#lab (`https://bend-lang.com/#lab`) — "Skeptical? Edit the demo's code and break the law!"
- Also: `demos/pure_hvm5_mini` — HVM5 minicore study (only HVM5 artifact; not a runtime release).

- Reuse: copy one demo dir as `LAWS.bend` template for every new app; require `bend PROOF.bend` in CI.

5. `GUIDE/` — DOCS

- `guide/GUIDE.md` — canonical guide; also printed by `bend guide`. Agent entry point (run first).
- `guide/EFFECTS.md` — effect system (CPU effects, `IO`, `do` blocks).
- `guide/SHADERS.md` — GPU shader path (Metal/CUDA).
- Syntax teaser (Python + dependent types):

```
import Base
def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER"))
    IO.print("Hello, " ++ name)
```

- Parallelism teaser (divide-and-conquer, `!` = GPU dispatch):

```
import Base
def pow2(+d: Nat) -> U32:
  match d:
    case 0n: 1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)
def main() -> IO(Unit):
  result = pow2!(20n)
  IO.print(U32.show(result))
```

- Theorem teaser (law + inductive proof with `%` rewrites):

```
import Base
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}
def add_zero(x):
  match x:
    case 0n: {==}
    case 1n+xp:
      %add_zero(xp) : {1n+Nat.add(xp, 0n) == 1n+_ : Nat}
      {==}
```

6. `PAPER/` — FORMALIZATION

- `paper/BendTT.pdf` — "BendTT: An Affine Dependent Type Theory" (type theory).
- `paper/BendRT.pdf` — "BendRT: A Parallel Runtime for CPUs and GPUs" (runtime).
- `bend2/bend.lean` (`bendtt.lean`) — Lean kernel of `--verdict` + proofs.
- Chain: Kindelia WHITEBOOK (mana/memory/serialization) → HVM2 paper (interaction combinators) → IC spec (optimal reduction) → BendTT (affine dependent types) → BendRT (parallel codegen).
- No tactics/proof search in language — proofs manual; `--verdict` is the trusted path.

7. `BENCH/` + `EVALS/` + `TESTS/` + `GATES/`

- `bench/` — every bench behind the charts:
  - `bench/runtime/` — Bend vs C, TypeScript, Lean on 1 core / 16 cores / GPU (`media/runtime.gif`).
  - `bench/checker/` — Bend vs Isabelle, Agda, Lean, Rocq (`media/checker.gif`).
  - README caveat: "We don't have as many benchmarks as we'd like yet, especially for the checker."
- `evals/` — evaluation harness (list by cloning; wire LamBench kernels here).
- `tests/` — test suite (run before commit alongside `bend PROOF.bend`).
- `gates/` — acceptance gates (promote to CI required checks).
- `tools/bend-fmt-lsp/` — formatting-only Bend2 language server (official).
- Community LSP: https://github.com/don2e4/bend2-lsp (`https://github.com/don2e4/bend2-lsp`) (formatting + diagnostics + hover).

- Method: re-run `bench/` for numbers (GIFs are status, not tables); add HVM sieve/FFT/SAT ports + IC 16-interaction smoke term to `bench/runtime/`; add Kind cases to `bench/checker/`.

8. HUB (`BEND2/PACK/` + HUB NOTE)

- Packages are content hashes. Explicit non-features: no names, versions, accounts, search.
- Implication: pin by hash; vendor critical deps; no `npm search` equivalent.
- Roadmap: "Most of these limitations are being addressed" — watch `CHANGELOG.md` + releases.
- Local check: `ls bend2/pack/ && cat CHANGELOG.md | head -100`.

9. LIMITATIONS (FULL, VERBATIM FROM README 2026-09-28 — KEEP INTACT)

```
- Bend 2 is a new language. Bend 1 programs and HVM do not carry over.
- Everything is annotated and nothing is inferred, so code is verbose.
- No type classes, no traits, and no macros beyond compile-time templates.
- Bend has no tactics or proof search; proving theorems takes extra effort.
- Values are affine: closures and arrays cannot be shared.
- Recursion must terminate; `@unsafe` or `def f?(..)` turns the check off.
- Computed matches (`match f(x)`) aren't supported. Must split it manually.
- There is no syntax for if-then-else: a branch is a match on True and False.
- Numbers are Nat, U32 and F32 only: no U64, I64 or F64 (Metal has no f64).
- F32 is axiomatic: nothing about floating point can be proven.
- Strings are linked lists of characters, so text processing is slow.
- Base is small: expect to write helpers other languages ship built in.
- Effects are few: print, env, time, sleep, spawn, channels, files, TCP, UDP.
- No TLS, HTTP library, JSON or regex for now (you can add them as foreigns).
- Targets are C, Metal, CUDA and JavaScript; Lua, Luau and Python are planned.
- The JavaScript target runs on one core and has no graphics or audio.
- Parallelism requires balanced calls. Flexible parallelism will be added later.
- Atomic arrays shared across threads are experimental and need `@unsafe`.
- One GPU per program, one event loop, and no multi-machine execution yet.
- One C file per program: no separate compilation, no incremental builds.
- Compiling to native is slow (clang/CUDA/Metal). For fast development, use JS.
- The compiler is young and has blind spots (unusually slow programs). Report.
- We don't have as many benchmarks as we'd like yet, especially for the checker.
- The compiler (not kernel) is 99% AI-written and not yet fully audited.
- The checker has no proof and may have bugs; `--verdict` uses a proven kernel.
- A binary needs clang 14+; ! needs 19+, Metal or CUDA 12.
- No Windows (WSL works); on Linux, Window and Audio need X11 and ALSA headers.
- The hub has no names, versions, accounts or search yet. Packages are hashes.
- Error messages are terse; no debugger, profiler or REPL.
- The editor tool only formats; the community bend2-lsp adds errors and hover.
- No test framework and no documentation beyond the guide.
- And more that escape me. Be patient, report bugs and request features!
Most of these limitations are being addressed and will improve over time!
```

Plus: "BEND IS YOUNG. EXPECT BUGS AND REPORT THEM: https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues)." `WONTFIX.txt` + `CHANGELOG.md` + `flake.nix` at root — read all three before filing issues.

10. COMMUNITY

- Website: https://bend-lang.com (https://bend-lang.com) (install script, `#lab` interactive law-breaker).
- Discord: https://discord.bend-lang.com (https://discord.bend-lang.com).
- X: https://x.com/bendlang (https://x.com/bendlang).
- Reddit: https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/).
- Issues: https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues) (README: "Bend is young. If anything goes wrong, ask it to open an issue. <3" — best on backend, Linux/macOS).
- LSP: official `tools/bend-fmt-lsp` (format only) + community https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp).
- Credits: Victor Taelin + Lorenzo W Battistela, Paulo J Cavalcanti, Nico, Vanessa Ostroski, Vitor Chiarelli Neves, Alex Van de Sande. Name thanks: Ayush Somani (reserved `bend-lang` name).

11. ACTION LIST (DO THIS WEEK)

- <input disabled="" type="checkbox"> Install: `curl -fsSL https://bend-lang.com/install.sh | sh && bend guide | head -200`.
- <input disabled="" type="checkbox"> Clone: `git clone https://github.com/bendlang/bend && ls bend2 demos guide paper bench evals tests gates`.
- <input disabled="" type="checkbox"> Read: `AGENTS.md`, `CHANGELOG.md`, `WONTFIX.txt`, `guide/GUIDE.md`, `bend2/base.bend` header.
- <input disabled="" type="checkbox"> Run: one `demos/*` end-to-end (`bend PROOF.bend + bend run`); try breaking its law via `#lab`.
- <input disabled="" type="checkbox"> Copy: `LAWS.bend` template into your app; add `bend PROOF.bend + tests/ + gates/` to CI.

- <input disabled="" type="checkbox"> Bench: run `bench/runtime/ + bench/checker/` baselines; record CPU/GPU/checker deltas.
- <input disabled="" type="checkbox"> Target check: `clang --version` (≥14, ≥19 for `!`), Metal/CUDA 12 presence, WSL if on Windows.
- <input disabled="" type="checkbox"> Hub: resolve deps by hash in `bend2/pack/`; vendor what you can't re-fetch.
- <input disabled="" type="checkbox"> Report: file blind spots/slow programs at https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues) with repro.
- <input disabled="" type="checkbox"> Track: `Kindelia WHITEBOOK` mana model + `paper/BendTT.pdf|BendRT.pdf` + `bend2/bend.lean` for proof work.

12. OPEN QUESTIONS / TO VERIFY BY CLONING

- Full `bend2/docs/`, `bend2/effs/`, `bend2/pack/` inventories (webfetch hides trees).
- `demos/` 16-name roster + per-demo `LAWS.bend` complexity.
- `evals/` runner CLI + `gates/` pass/fail criteria + `tests/` runner (no test framework per Limitations?).
- `bench/runtime|checker` repro commands behind `media/*.gif`.
- `tools/bend-fmt-lsp` vs `don2e4/bend2-lsp` feature parity; `flake.nix` dev shell contents.

F.30 · `bend-runtime/knowledge/05-ecosystem/higherorderco-org.md`

HigherOrderCO Org — Map, HVM Lineage, Bend1/Kind, CLI, Benches, Limits

Survey date (UTC): 2026-09-28. Sources: https://github.com/HigherOrderCO (https://github.com/HigherOrderCO), https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind), https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus), https://github.com/bendlang/bend (https://github.com/bendlang/bend), https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md).

1. ORG SNAPSHOT

- Handle: `HigherOrderCO` — tagline (verbatim): "we are getting to the very core of what makes computers capable of reasoning."
- Followers: 2.8k (org header, 2026-09-28).
- Links: https://bend-lang.com (https://bend-lang.com), X `@HigherOrderComp` (https://twitter.com/HigherOrderComp (https://twitter.com/HigherOrderComp)), `contact@higherorderco.com`.
- People (visible, 2): `nicolas-abril`, `VictorTaelin`.
- Top languages (badge order): Haskell, Cuda, C, Rust.
- Repo count: 7 total (all public, listed §2). No private/hidden count exposed.
- Pinned (4 slots rendered): HVM1, HVM4, Bend1, Bend2 (note: HVM2/HVM3/Kind unpinned but present).
- Mission reading: Victor README gists — optimal computer (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b)) + lore (https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1 (https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1)).

2. ORG MAP (ALL 7 REPOS, FETCHED 2026-09-28)

Repo	URL	Desc (verbatim/short)	Lang	Stars / forks	Updated	License (badge)
Bend2 (stub)	https://github.com/HigherOrderCO/Bend2 (https://github.com/HigherOrderCO/Bend2)	"Bend 2 moved to github.com/bendlang/bend"	—	10 / 0, 0 issues, 0 PRs	2026-09-16	—
HVM2	https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)	"HVM2 (2024): massively parallel, optimal functional runtime in Rust" (forked from HVM1)	Cuda	41 / 449	2026-09-16	Apache-2.0

Repo	URL	Desc (verbatim/short)	Lang	Stars / forks	Updated	License (badge)
HVM1	https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1)	"HVM1 (2022): massively parallel, optimal functional runtime in Rust"	Cuda	11,346 / 437	2026-09-16	MIT
Bend1	https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1)	"Bend 1: original massively parallel, high-level language (superseded by github.com/bendlang/bend)"	Rust	5 / 1	2026-09-16	Apache-2.0
HVM4	https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)	(no desc) — C single-file runtime	C	121 / 28, 10 issues, 14 PRs	2026-05-30	—
HVM3	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3)	"HVM3" — Haskell IC implementation	Haskell	305 / 29, 7 issues	2026-01-29	MIT
Kind	https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind)	"A modern proof language"	Haskell	3,761 / 152, 3 issues, 4 PRs	2025-01-22	MIT

Observations:

- HVM1 dominates stars (11.3k ≈ 73% of org-visible stars); HVM2 fork count (449) exceeds HVM1 (437) — reflects Bend1-era forking, not popularity (HVM2 only 41★).
- Bend1 nearly unstarred (5★) — superseded marker is intentional; do not benchmark Bend1 as live.
- Bend2 stub has zero code; sole function is redirect. Always cite https://github.com/bendlang/bend (https://github.com/bendlang/bend) for Bend2.
- No HVM5, Kindelia, awesome-list, or standalone benchmark repos in org — HVM5-mini lives at `bendlang/bend/demos/pure_hvm5_mini` ; benches live at `bendlang/bend/bench/` .
- Freshness gradient: HVM1/HVM2/Bend1/Bend2 touched 2026-09-16 (likely sweep), HVM4 May 2026, HVM3 Jan 2026, Kind Jan 2025 (stable/stale).

3. HVM LINEAGE (HVM1 → HVM4)

HVM1 (2022, RUST + CUDA, MIT, 11.3K★)

- Role: first optimal parallel runtime; cultural anchor (most-starred, pinned #1 on both Victor + org).
- Key doc: `HOW.md` — intuitive optimal-evaluation explainer, pinned in Victor README at `HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md` .
- Research threads: deforestation via β-optimality (https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474 (https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474)).
- Reuse: HOW.md is the onboarding doc; HVM1 numbers are historical, not Bend2 numbers.

HVM2 (2024, C/CUDA, APACHE-2.0, 41★ / 449 FORKS)

- Role: IR-focused successor; paper `paper/HVM2.pdf` — "A Parallel Evaluator for Interaction Combinators".
- Listed as "Forked from HigherOrderCO/HVM1" — continuity, but separate paper/benchmarks.
- Gists tied here: prime sieve (https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8 (https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8)), FFT (https://gist.github.com/VictorTaelin/5776ede998d0039ad1cc9b12fd96811c (https://gist.github.com/VictorTaelin/5776ede998d0039ad1cc9b12fd96811c)), SAT (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615 (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615)).

- Reuse: cite paper for interaction-combinator semantics; port sieve/FFT/SAT to LamBench.

HVM3 (HASKELL IC, MIT, 305★)

- Role: fast, fully-featured Interaction Calculus implementation; spec docs `HVM.md/IC.md/INTERS.md/MODES.md`.
- Advances: optimal atomic linker with polarization (https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b `(https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b)`), unordered superpositions solving O(n²) case (https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a `(https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a)`).
- Companion spec: https://github.com/VictorTaelin/Interaction-Calculus `(https://github.com/VictorTaelin/Interaction-Calculus)` (VAR/ERA/LAM/APP/SUP/DUP, IC32 runtime, collapse/DUP-perm/labeled-lambda extensions).
- Reuse: cleanest spec + Haskell single-file learner with step debugger.

HVM4 (C SINGLE-FILE, 121★)

- Role: minimal portable C runtime; `src/hvm.c + docs/` (per prior survey; re-clone to confirm paths).
- Activity: only lineage repo with open PRs (14) + issues (10) — review queue before reuse.
- Reuse: embeddable baseline for C-target comparisons (`bendlang/bend` C backend vs HVM4).

WHAT IS NOT IN ORG

- HVM5: only `bendlang/bend/demos/pure_hvm5_mini` — treat as demo, not runtime release.
- Kindelia: `kindelia/Kindelia` org (617★) — WHITEBOOK mana/memory/serialization model (§6 of victor-taelin.md); cryptocomputer, PoW, no token.
- Bend2: `bendlang/bend` org (23k★) — live compiler, see `bendlang-bend-repo.md`.

4. BEND1 (SUPERSEDED) VS KIND (HASKELL PROOF CHECKER)

BEND1 — HTTPS://GITHUB.COM/HIGHERORDERCO/BEND1 `(HTTPS://GITHUB.COM/HIGHERORDERCO/BEND1)` (RUST, APACHE-2.0, 5★)

- Status: superseded banner; `examples/` (16 files per prior survey), `GUIDE.md/FEATURES.md/FAQ.md`.
- Do not carry Bend1 programs to Bend2 — `bendlang/bend` Limitations states explicitly: "Bend 2 is a new language. Bend 1 programs and HVM do not carry over."
- Reuse: historical syntax comparison only; archive any Bend1→Bend2 migration notes separately.

KIND — HTTPS://GITHUB.COM/HIGHERORDERCO/KIND `(HTTPS://GITHUB.COM/HIGHERORDERCO/KIND)` (HASKELL, MIT, 3.8K★, 279 COMMITS)

- Layout: `app/`, `src/`, `main.kindc`, `kind-lang.cabal`, `cabal.project`, `CHANGELOG.md`.
- README stub (2026-09-28): "minimal Proof Checker ... full rewrite from old JS to Haskell ... actively developed ... examples on KindBook https://github.com/HigherOrderCO/KindBook `(https://github.com/HigherOrderCO/KindBook)`".
- Lineage: old JS Kind → Haskell Kind → BendTT (`bend2/bend.lean`, `paper/BendTT.pdf`).
- Ancestor: https://github.com/VictorTaelin/Cedille-Core `(https://github.com/VictorTaelin/Cedille-Core)` (220★, minimal JS proof lang, erasure + heterogeneous equality).
- Checker note: https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5 `(https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5)` (conversion checker).
- Install: `cabal run kind -- --help` (verify flag; README defers to source).

5. VERSIONS AND UPDATE CADENCE

- No tags/branches enumerated via webfetch (auth-walled commit graphs); use `git ls-remote` locally: `git ls-remote https://github.com/HigherOrderCO/HVM1 --tags | tail -20`.
- Observed `Updated` dates (§2 table) — treat 2026-09-16 cluster as maintenance sweep, not feature release.
- Kind stale since 2025-01-22 — pin dependency, do not expect upstream fixes.
- HVM4 (May 2026) + HVM3 (Jan 2026) are freshest code-bearing lineage repos.
- For Bend2 versions, see `bendlang/bend/CHANGELOG.md` + `bendlang/bend` releases (3088 commits on `main`).

6. CLI SURFACE (ORG-ERA VS BEND2-ERA)

Org-era CLIs (verify by cloning; flags drift):

- HVM1/HVM2 (Rust): `hvm run/check` style; CUDA backend flags; `HOW.md` examples.
- HVM3 (Haskell): `hvm` with IC modes (`HVM.md/MODES.md`).
- HVM4 (C): `make && ./hvm ...` or `src/hvm.c` single-file build.
- Kind (Haskell): `kind check/run on .kindc` terms.

Bend2-era CLI (canonical, from `bendlang/bend` README):

```
curl -fsSL https://bend-lang.com/install.sh | sh
bend guide        # print GUIDE.md — agent entry point
bend base         # print bend2/base.bend
bend PROOF.bend   # check proofs against LAWS.bend
bend run main.bend
```

Agent contract (`AGENTS.md` , also in `bendlang/bend`):

```
When using Bend:
- run `bend guide` to learn it
- use `LAWS.bend` to keep important rules
- run `bend PROOF.bend` before committing
- parallelize the code whenever possible
```

Note: installed `bend` ignores project `bunfig.toml` / `.env` ; `bun bend2/main.ts` from checkout reads them — audit untrusted code with installed `bend` .

7. BENCHES (WHERE NUMBERS LIVE)

- Org has no standalone bench repo. Benchmark sources:
  - HVM1/HVM2 in-repo benches + `paper/HVM2.pdf` charts.
  - `bendlang/bend/bench/` — `runtime/` (Bend vs C/TS/Lean, 1-core/16-core/GPU) + `checker/` (Bend vs Isabelle/Agda/Lean/Rocq) backing `media/runtime.gif + media/checker.gif` .
  - `bendlang/bend/evals/` , `tests/` , `gates/` — harness + gates (see `bendlang-bend-repo.md`).
- Targets: Bend2 aims "as fast as C on CPU, as fast as CUDA on GPU"; checker aims "OOMs faster than every proof assistant" — status GIFs, not tables; re-run `bench/` for numbers.
- LamBench mapping: HVM sieve/FFT/SAT/search gists → `bench/runtime/` kernels; Kind checker cases → `bench/checker/` ; IC 16-interaction default term → smoke test.

8. LIMITS (ORG-ERA INHERITED + BEND2 EXPLICIT)

Inherited (Kindelia WHITEBOOK, still useful as mental model):

- `mana_limit = 4M*(block+1)` ; `bits_limit = 2048*(block+1)` ; ~55M mana/s M1 baseline.
- Linearity + `dup` discipline; `Save` zero-cost, heap growth charged.

Bend2 explicit (`bendlang/bend` Limitations — abridged, see full list in `bendlang-bend-repo.md`):

- New language: Bend1/HVM do not carry over.
- Verbose (everything annotated, nothing inferred); no type classes/traits; no tactics/proof search.
- Affine values (closures/arrays unshareable); recursion must terminate (`@unsafe / def f?` escapes).
- No computed matches, no if-then-else syntax; Nat/U32/F32 only; F32 axiomatic; strings are char lists.
- Small Base; few effects (print/env/time/sleep/spawn/channels/files/TCP/UDP); no TLS/HTTP/JSON/regex.
- Targets C/Metal/CUDA/JS (Lua/Luau/Python planned); JS single-core, no gfx/audio; balanced-call parallelism; 1 GPU/program, 1 event loop, no multi-machine; 1 C file/program, clang-limited native builds.
- Compiler 99% AI-written, unaudited; checker unproven (`--verdict` uses proven `bend.lean` kernel); hub hash-only (no names/versions/accounts/search); terse errors, no debugger/profiler/REPL.

9. REUSE CHECKLIST

- <input disabled="" type="checkbox"> Link org https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) with tagline + 2.8k followers (date).
- <input disabled="" type="checkbox"> Table all 7 repos with stars/forks/updated/license; flag Bend2 stub → `bendlang/bend` .

- <input disabled="" type="checkbox"> State HVM1→HVM4 roles in one paragraph; cite HOW.md + HVM2 PDF + IC repo + HVM4 PR queue.
- <input disabled="" type="checkbox"> Mark Bend1 superseded with no-carry warning; cite Bend2 Limitations verbatim.
- <input disabled="" type="checkbox"> Describe Kind Haskell rewrite + KindBook + Cedille-Core ancestor + conversion gist.
- <input disabled="" type="checkbox"> Document CLI split (org-era per-runtime vs Bend2 `bend guide/base/PROOF.bend`).
- <input disabled="" type="checkbox"> Point benches to `bendlang/bend/bench|evals|tests|gates`, not org.
- <input disabled="" type="checkbox"> Copy limits block with provenance (WHITEBOOK vs Bend2 README).
- <input disabled="" type="checkbox"> Re-run `git ls-remote --tags` for versions before publishing (webfetch hides tags).

10. OPEN QUESTIONS / TO VERIFY BY CLONING

- `HVM4/src/hvm.c + docs/` paths; `HVM3/HVM.md|IC.md|INTERS.md|MODES.md` contents.
- `Bend1/examples/` count (16?) + `GUIDE.md/FEATURES.md/FAQ.md` migration deltas.
- `Kind/app|src` entry points + `main.kindc` example coverage + KindBook URL liveness.
- HVM2 fork-vs-star anomaly (449 forks / 41★) — history rewrite or Bend1 fork influx?
- `HOW.md` hash `73da3bb` still valid after HVM1/HVM2 split?

F.31 · bend-runtime/knowledge/05-ecosystem/papers-formalization-community.md

Papers, formalization, community

- Theory: https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf) — affine dependent type theory (live/dead wall, `Type:Type`, no positivity).
- Runtime: https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf) — parallel CPU/GPU design + benchmarks.
- HVM2 paper: https://paper.higherorderco.com (https://paper.higherorderco.com) (mirror `HigherOrderCO/HVM1/blob/main/paper/HVM2.pdf`).
- Formalization: https://github.com/bendlang/bend/blob/main/bend2/bend.lean (https://github.com/bendlang/bend/blob/main/bend2/bend.lean) (lags `bend.ts`).
- Base: https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend) (`= bend base`).
- Guides: https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md), `EFFECTS.md`, `SHADERS.md`.
- Site/hub: https://bend-lang.com (https://bend-lang.com) (install, charts, lab https://bend-lang.com/#lab (https://bend-lang.com/#lab)), https://hub.bend-lang.com (https://hub.bend-lang.com) (hash packages), terms https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18).
- Lafont: https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718) (1990 nets), https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432) (1997 combinators).
- Optimal: http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf) (Lévy), https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711) (Lamping).
- Community: https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues).

F.32 · bend-runtime/knowledge/05-ecosystem/victor-taelin.md

Victor Taelin — Profile, Pinned Repos, and Research Lineage

Survey date (UTC): 2026-09-28. Sources: https://github.com/VictorTaelin (https://github.com/VictorTaelin), https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus), https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind), https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md), https://github.com/HigherOrderCO (https://github.com/HigherOrderCO), https://github.com/bendlang/bend (https://github.com/bendlang/bend).

1. PROFILE SNAPSHOT

- Handle: `VictorTaelin` — display name "Victor Taelin".
- Bio (verbatim, 2026-09-28): "Functional-programming enthusiast, author of Kind and HVM."
- Location: Rio de Janeiro, Brazil.
- Affiliation: Higher Order Company (`HigherOrderCO` org avatar listed under Organizations).
- Contacts from profile + `VictorTaelin/VictorTaelin/README.md` :
  - X: https://twitter.com/VictorTaelin (https://twitter.com/VictorTaelin) (`@victortaelin`).
  - Instagram: https://instagram.com/victortaelin (https://instagram.com/victortaelin).
  - Reddit: https://www.reddit.com/user/SrPeixinho/ (https://www.reddit.com/user/SrPeixinho/) (`SrPeixinho`).
  - Email: `taelin@higherorderco.com` .
  - Company site: https://higherorderco.com (https://higherorderco.com).
  - Discord (org-level): https://discord.higherorderco.com (https://discord.higherorderco.com) (listed on prior survey; verify before use).
- Scale (profile header, fetched 2026-09-28):
  - 6.5k followers, 24 following.
  - 261 repositories, 735 starred repos (Stars tab), achievements: Galaxy Brain, Public Sponsor, Starstruck x4, Pull Shark x2, Arctic Code Vault Contributor.
- Founder role: README opens "Hi! I'm a programmer and founder of the HigherOrderCO."
- What to reuse: bio line + contact block + org link are the canonical identity triple for citations.

2. PINNED REPOSITORIES (6, AS RENDERED 2026-09-28)

#	Pinned repo	URL	Language	Stars / forks (fetched)	One-line charter
1	HVM1	https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1)	Cuda	11.3k / 437	HVM1 (2022): massively parallel, optimal functional runtime in Rust
2	Kind	https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind)	Haskell	3.8k / 152	A modern proof language (Haskell rewrite)
3	WebMonkeys	https://github.com/VictorTaelin/WebMonkeys (https://github.com/VictorTaelin/WebMonkeys)	JavaScript	1.4k / 53	Massively parallel GPU programming on JS, via WebGL
4	Interaction-Calculus	https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus)	C	958 / 71	Minimal calculus matching optimal λ-calculus reduction
5	Cedille-Core	https://github.com/VictorTaelin/Cedille-Core (https://github.com/VictorTaelin/Cedille-Core)	JavaScript	220 / 13	Minimal proof language (~1k LOC JS)
6	UrnaCripto	https://github.com/VictorTaelin/UrnaCripto (https://github.com/VictorTaelin/UrnaCripto)	JavaScript	62 / 1	Referendos criptograficamente incorruptíveis (PT)

Notes:

- Pins mix org-owned (HVM1, Kind) and personal (WebMonkeys, IC, Cedille-Core, UrnaCripto).
- Star order is not sorted: HVM1 > Kind > WebMonkeys > IC > Cedille-Core > UrnaCripto.
- HVM1 pin label says "in Rust" but top language badge shows Cuda — runtime has Rust host + CUDA backend.
- Prior survey counted "957★" for IC; live count on fetch is 958★. Use live count with date.

3. INTERACTION-CALCULUS (IC) DEEP DIVE

Repo: https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus) — 135 commits, `main`, layout `examples/`, `haskell/`, `src/`, `CLAUDE.md`, `Makefile`, `README.md`.

3.1 THESIS

README thesis (paraphrased, keep verbatim for quotes):

> Minimal term rewriting inspired by λ-calculus, but affine + global vars + first-class superpositions/duplications; resembles Lamping's optimal evaluator; fully affine GC; simplifies parallelism. HVM3 is the fast, full implementation.

Key properties:

1. Vars affine (≤1 occurrence).
2. Vars global (can occur anywhere — enables continuations, linear HOAS, mutable refs).
3. First-class SUP/DUP (optimal evaluation, some computations exponentially faster).

3.2 GRAMMAR

```
Term ::= VAR: Name | ERA: "*" | LAM: "λ" Name "." Term
       | APP: "(" Term " " Term ")"
       | SUP: "&" Label "{" Term "," Term "}"
       | DUP: "!" "&" Label "{" Name "," Name "}" "=" Term ";" Term
```

Labels are numeric and control DUP-SUP (equal = annihilate/collapse, different = commute/duplicate).

3.3 SIX CORE RULES (+ EXTENSIONS)

Core (from README):

- `APP-ERA`: `(* a) → *`.
- `APP-LAM` (beta): `((λx.f) a) → f[x:=a]` via global substitution map `sub[var]=value`, O(1), no traversal, fresh globally-unique names avoid capture; concurrent via atomic-swap.
- `APP-SUP`: `(&L{a,b} c) → !&L{c0,c1}=c; &L{(a c0),(b c1)}`.
- `DUP-ERA`: `!&L{r,s}=*; K → K[r:=*,s:=*]`.
- `DUP-LAM`: duplicates lambda lazily, pairs bound var: `x <- &L{x0,x1}`, then `!&L{f0,f1}=f`.
- `DUP-SUP`: if `L==R`, `x<-a,y<-b`; else cross-duplicate with fresh `a0,a1,b0,b1`.

Extensions documented in same README:

- Collapsing (ERA-LAM, ERA-APP, SUP-LAM, SUP-APP, DUP-VAR, DUP-APP) — collapse IC to superposed tree of pure λ-terms without SUP/DUP.
- DUP permutations (`APP-DUP`, `DUP-DUP`) for implementations storing DUP body.
- Labeled lambdas (`&Lλ`, `&L(...)`) with extended APP-LAM for mismatched labels.
- Theory section "IC = λC ∪ Interaction Combinators": lams/apps as γ (constructor), pars/lets as δ (duplicator); annihilation = beta/project; commutation = lambda-projection / pair-application, previously stuck λ-terms now compute.

3.4 IC32 32-BIT RUNTIME

Portable C runtime. Term = 32-bit word: `sub` (1 bit) + `tag` (5 bit) + `val` (26 bit). Tags: VAR 0x00, LAM 0x01, APP 0x02, ERA 0x03, NUM 0x04, SUC 0x05, SWI 0x06, TMP 0x07, SP0–SP7 0x08–0x0F, CX0–CX7 0x10–0x17, CY0–CY7 0x18–0x1F. Nodes are consecutive heap blocks; DUP floats (no body in AST, reachable only via CX/CY vars); binder node is overwritten with `sub`-bit set on interaction (see `dup_sup` pseudocode in README). Parsing uses `lcs/uses + vars` two-phase global linking; stringify assigns binder IDs, hoists floating DUPS to prefix `!&{..}=..` lines.

3.5 USAGE

```
make clean && make
./bin/ic run examples/test_0.ic
# Haskell single-file in haskell/ has step debugger — use for learning
```

4. KIND (PROOF LANGUAGE)

- Canonical: https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind) — Haskell rewrite, 279 commits, MIT, 3.8k★.
- Layout: `app/` , `src/` , `main.kindc` , `kind-lang.cabal` , `cabal.project` , `CHANGELOG.md` .
- README (2026-09-28) is stub: "Kind is a minimal Proof Checker ... full rewrite from old JS to Haskell ... See examples on KindBook https://github.com/HigherOrderCO/KindBook (https://github.com/HigherOrderCO/KindBook) ... proper README later".
- Historical JS Kind → Haskell Kind → BendTT trajectory: Cedille-Core (erasure + heterogeneous equality, ~1k LOC) is the minimal ancestor; Kind adds conversion checker (see gist below); Bend2's `bend2/bend.lean`
  - `paper/BendTT.pdf` is the successor proof core.
- Clone/run (Haskell): `git clone https://github.com/HigherOrderCO/Kind && cd Kind && cabal run kind -- --help` .
- Reuse: cite KindBook for examples; do not expect stable guide in repo itself.

5. KINDELIA + WHITEBOOK (FUNCTIONAL CRYPTOCOMPUTER)

- Repo: https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) — 617★, 41 forks (on WHITEBOOK page header).
- Doc: https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md) — 2066 lines, 80 KB, `master` .
- Thesis (verbatim core): HVM-backed P2P functional computer; formally verified DApps cheap; reversible heaps make dynamic apps cheap; no built-in token — "it isn't a cryptocurrency".
- Language: affine lazy HVM terms; 8 term variants; `dup x y = e; body` for cloning; 120-bit native numbers (128-bit ptr − 8-bit tag); 16 numeric ops (ADD/SUB/MUL/DIV/MOD/AND/OR/XOR/SHL/SHR/LE/LT/EQ/GT/GE/NE).
- Statements: `ctr` , `fun ... with {init}` , `run {IO}` , `reg Namespace {owner} sign {...}` ; namespace dots, secp256k1 `sign{}` , subject/caller (`Subj` / `From`).
- Effects as pure `IO s a` : `Done`/`Take`/`Save`/`Call`/`Name`/`From` ; `Take` removes state (linear), `Load` clones (defined in genesis); `Save` itself zero-cost — only heap growth + mana charged.
- Costs (mana table): APP-LAM 2, APP-SUP 4, OP2-NUM 2, OP2-SUP 4, FUN-CTR 2+M, FUN-SUP 2+A, DUP-LAM 4, DUP-NUM 2, DUP-CTR 2+A, DUP-SUP 4 (dup) / 2 (collapse), DUP-ERA 2. Limits: `mana_limit = 4_000_000*(block+1)` , `bits_limit = 2048*(block+1)` (~8 GB/yr). Impl: ~55M mana/s single-thread M1 (~7x limit).
- Memory: `Mem = Vec<u128>` ; `Ptr = TT(8b)+A(72b)+B(48b)` ; 12 tags DP0/DP1/VAR/ARG/ERA/LAM/APP/SUP/CTR/FUN/OP2/NUM.
- Serialization: fixlen/varlen/list/number/name/term/statement; 3-bit term tags, De Bruijn/compressed names, `serialize_number = varlen(bit_size)+fixlen` .
- Consensus: Nakamoto PoW, 1s/block target; no fee mechanism — early self-mining, later external fee market; "Why not L2 rollup / why PoW / why no currency" answered in Whys section.
- Reuse for Bend2: mana model → bench design; reversible-heap zero-SSTORE → affine state discipline; HVM HOW.md → optimal-evaluation explainer.

6. HVM LINEAGE (VICTOR'S ANGLE)

- HVM1 (2022, Rust+CUDA, 11.3k★): first massively parallel optimal runtime; `HOW.md` intuition doc.
- HVM2 (2024, fork, C/CUDA): `paper/HVM2.pdf` "A Parallel Evaluator for Interaction Combinators"; prime-sieve, FFT, SAT-solver gists explore linearity/SUP-DUP.
- HVM3 (Haskell, IC): `HVM.md/IC.md/INTERS.md/MODES.md` ; optimal atomic linker with polarization; unordered superpositions fix $O(n^2)$ case.
- HVM4 (C single-file `src/hvm.c + docs/`): minimal portable successor.
- HVM5-mini (not org repo): `bendlang/bend/demos/pure_hvm5_mini` — study minicore before org history.
- IC repo is the spec anchor: HVM3 README points back to IC; Bend2 (`bendlang/bend`) explicitly breaks compat ("Bend 2 is a new language. Bend 1 programs and HVM do not carry over" — Limitations).
- LamBench hook: HVM optimal-vs-naive gaps (high-order tasks exponential) are the benchmark thesis; reuse prime-sieve/FFT/SAT/search gists as LamBench candidates.

7. GISTS (README-INDEXED, GROUPED)

Company:

- HOC optimal computer: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b)

- HOC lore: https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1 (https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1)
- Tweet version: https://x.com/VictorTaelin/status/1806690584670679387 (https://x.com/VictorTaelin/status/1806690584670679387)

Learning:

- HOW (HVM1): https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md (https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md)
- 10-min optimal: https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6)
- Mystery (exponential): https://gist.github.com/VictorTaelin/85b94b5ba5b8b5440ded64bba8c89ac2 (https://gist.github.com/VictorTaelin/85b94b5ba5b8b5440ded64bba8c89ac2)
- Deforestation: https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474 (https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474)
- Kind conversion: https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5 (https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5)
- Termination via fold: https://gist.github.com/VictorTaelin/676241eb4e290a006e5618f97b8a1c25 (https://gist.github.com/VictorTaelin/676241eb4e290a006e5618f97b8a1c25)

Research code:

- Prime sieve: https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8 (https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8)
- FFT λ-terms: https://gist.github.com/VictorTaelin/5776ede998d0039ad1cc9b12fd96811c (https://gist.github.com/VictorTaelin/5776ede998d0039ad1cc9b12fd96811c)
- SAT SUP/DUP: https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615 (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615)
- Search #1 add-carry: https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34 (https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34)
- Search #2 bits DSL: https://gist.github.com/VictorTaelin/7fe49a99ebca42e5721aa1a3bb32e278 (https://gist.github.com/VictorTaelin/7fe49a99ebca42e5721aa1a3bb32e278)
- Search #3 enum superposed: https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e (https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e)
- Atomic linker polarized: https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b (https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b)
- Unordered SUP optimal: https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a (https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a)
- Collapse monad: https://gist.github.com/VictorTaelin/60d3bc72fb4edefecd42095e44138b41 (https://gist.github.com/VictorTaelin/60d3bc72fb4edefecd42095e44138b41)
- Linear ctx passing: https://gist.github.com/VictorTaelin/fb798a5bd182f8c57dd302380f69777a (https://gist.github.com/VictorTaelin/fb798a5bd182f8c57dd302380f69777a)
- 1D rotate trees: https://gist.github.com/VictorTaelin/c672b5bd9e734bae63ac35c010be38ab (https://gist.github.com/VictorTaelin/c672b5bd9e734bae63ac35c010be38ab)
- Equality recursive λ: https://gist.github.com/VictorTaelin/1af22e2c87f176da0e5ff8cd3430b04f (https://gist.github.com/VictorTaelin/1af22e2c87f176da0e5ff8cd3430b04f)
- GPT-4 compress ctx: https://gist.github.com/VictorTaelin/d293328f75291b23e203e9d9db9bd136 (https://gist.github.com/VictorTaelin/d293328f75291b23e203e9d9db9bd136)

8. TALKS, RELEASE, COVERAGE

- Talk (USP, EN subs): https://www.youtube.com/watch?v=sDPuQ-UjhVQ (https://www.youtube.com/watch?v=sDPuQ-UjhVQ) — "HVM: A Parallel Runtime".
- Release: https://x.com/VictorTaelin/status/1791213162525524076 (https://x.com/VictorTaelin/status/1791213162525524076) — "Bend: Release Day".
- Coverage:
  - Fireship: https://www.youtube.com/watch?v=HC0QmKTFzYY (https://www.youtube.com/watch?v=HC0QmKTFzYY)
  - Primeagen: https://www.youtube.com/watch?v=NaytZ0iX3fs (https://www.youtube.com/watch?v=NaytZ0iX3fs)
  - HN: https://news.ycombinator.com/item?id=40390287 (https://news.ycombinator.com/item?id=40390287) ("Bend: high-level language that runs on GPUs").

- Paper: HVM2 PDF via https://docs.google.com/viewer?url=https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf (https://docs.google.com/viewer?url=https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf) (legacy path; prefer HVM1/HVM2 repo `paper/` if cloning).

9. LAMBENCH HOOKS

- Candidate kernels: sieve, FFT, SAT, `pow2` splitting, Church-expt, Y-combinator, nested-loop SUP tests, IC `examples/test_0.ic` default term (16 interactions).
- Metrics to lift: mana table (§5) as cost model; IC global-sub O(1) vs traversal; collapse cost; HVM M1 55M mana/s baseline.
- Method: port each gist kernel to `bend2/ + bench/runtime/ + evals/` ; record C/CUDA/Metal/JS deltas; keep Bend1/HVM incompat note in harness header.

10. REUSE CHECKLIST (COPY-PASTE)

- <input disabled="" type="checkbox"> Cite profile: bio + Rio + HOC + 6.5k/261 (date 2026-09-28).
- <input disabled="" type="checkbox"> Cite pins with live stars; note HVM1 Rust-vs-Cuda badge nuance.
- <input disabled="" type="checkbox"> For optimal-eval claims, cite IC README rules + HOW.md + 10-min gist + mystery gist.
- <input disabled="" type="checkbox"> For proof lineage, cite Cedille-Core → Kind (Haskell) → BendTT (`bend2/bend.lean` , `paper/BendTT.pdf`).
- <input disabled="" type="checkbox"> For cost model, cite WHITEBOOK mana table + limits + memory model.
- <input disabled="" type="checkbox"> For benchmarks, cite HVM2 PDF + sieve/FFT/SAT gists; do not claim Bend2 inherits HVM numbers.
- <input disabled="" type="checkbox"> For talks, cite USP talk + release tweet + 3 coverage links.
- <input disabled="" type="checkbox"> Re-verify star/fork counts before publishing (they drift; IC 957→958 observed).

11. OPEN QUESTIONS / TO VERIFY

- Discord canonical: `discord.higherorderco.com` (profile era) vs `discord.bend-lang.com` (Bend2 era).
- WebMonkeys maintenance status; UrnaCripto relevance to Bend2 (likely none — list as historical).
- HVM `HOW.md` commit hash `73da3bb` freshness vs HVM1/HVM2 renames.
- Kindelia genesis `Namer` + PoW params if reusing namespace design.

F.33 · `bend-runtime/knowledge/06-examples/01-hello-area-adder.bend`

01-hello-area-adder.bend

Date: 2026-09-28

Sources: `bend guide` (bend 2.0.25); https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md);

https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos)

(io_hello_world); Base via `bend base`

Run: bend 06-examples/01-hello-area-adder.bend

Expected output:

Hello, Bend2!

circle~=27

square=16

rect=28

1+2=3

import Base

Data type with one constructor per shape (cf. 01-bend-language/types-quantities-kinds.md).

type Shape is Data: Circle{r: U32} Square{s: U32} Rect{w: U32, h: U32}

Plain (affine) scrutinee; `+r` in the pattern makes reused fields reusable.

def area(s: Shape) -> U32: match s: case Circle{+r}: (3 * r * r : U32) case Square{+ss}: (ss * ss : U32) case Rect{+w, +h}: (w * h : U32)

def circle_area() -> U32: area(Circle{3})

def square_area(s: U32) -> U32: area(Square{s})

def rect_area(w: U32, h: U32) -> U32: area(Rect{w, h})

Closure: affine, call at most once per binding (cf. functions-closures-templates-recursion.md).

def adder(k: U32) -> U32 -> U32: x => (x + k : U32)

def apply_adder(k: U32, x: U32) -> U32: adder(k)(x)

def main() -> IO(Unit): do IO<Unit>: IO.print("Hello, Bend2!") IO.print("circle~=" ++ U32.show(circle_area())) IO.print("square=" ++ U32.show(square_area(4))) IO.print("rect=" ++ U32.show(rect_area(7, 4))) IO.print("1+2=" ++ U32.show(apply_adder(2, 1)))

F.34 · `bend-runtime/knowledge/06-examples/02-recursion-parallel-arrays.bend`

02-recursion-parallel-arrays.bend

Date: 2026-09-28

Sources: `bend guide` + `bend guide shaders` (bend 2.0.25);

https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) **(pure_par_sum, pure_par_sort);**

https://github.com/bendlang/bend/tree/main/bench/runtime (https://github.com/bendlang/bend/tree/main/bench/runti

me)

Run: bend 06-examples/02-recursion-parallel-arrays.bend

(needs clang 14+, 19+ for `!`; without GPU the `!` call still runs, on CPU)

Expected output:

sum=10

pow2=1024

gpu=1024

tree=60

arr5=42

import Base

Tail recursion -> loop. Shrinking param (xs) first (termination rule).

def sum_list(xs: List<U32>, acc: U32) -> U32: match xs: case Nil{}: acc case Con{h, t}: sum_list(t, (acc + h : U32))

def sum1234() -> U32: sum_list([1, 2, 3, 4], 0)

Balanced fork-join: `a b = ... ...` runs both calls in parallel.

`+n` so the predecessor `p` can feed both branches.

def pow2(+n: Nat) -> U32: match n: case 0n: 1 case 1n+p: a b = pow2(p) pow2(p) (a + b : U32)

Parallel tree reduction (host builds tree, device/CPU reduces).

type Tree is Data: Leaf{v: U32} Node{l: Tree, r: Tree}

def tree_sum(t: Tree) -> U32: match t: case Leaf{+v}: v case Node{l, r}: a b = tree_sum(l) tree_sum(r) (a + b : U32)

def demo_tree() -> U32: tree_sum(Node{Leaf{10}, Node{Leaf{20}, Leaf{30}}})

Arrays: single owner (Type); writes hand the array back; reads hand

back `Array & elem`. Slot count after `*` is a power of two.

def array_demo() -> Array<U32> & U32: a = [0 : U32*8n] a[0] <- 10 a[1] <- 20 a[5] <- 42 a[5]

Pair projection must match on a parameter, never on a call result.

def proj_v(pair: Array<U32> & U32) -> U32: match pair: case (arr, v): v

def array_val() -> U32: proj_v(array_demo())

def main() -> IO(Unit): do IO<Unit>: IO.print("sum=" ++ U32.show(sum1234())) IO.print("pow2=" ++ U32.show(pow2(10n))) IO.print("gpu=" ++ U32.show(pow2!(10n))) IO.print("tree=" ++ U32.show(demo_tree())) IO.print("arr5=" ++ U32.show(array_val()))

F.35 · bend-runtime/knowledge/06-examples/03-io-channels-tcp-files.bend

03-io-channels-tcp-files.bend

Date: 2026-09-28

Sources: `bend guide + bend guide effects` (bend 2.0.25); `bend base` (IO/Chan/File/TCP/Socket/Listener);

https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) (io_hello_world, io_http_server, io_tcp_echos)

Run: USER=you bend 06-examples/03-io-channels-tcp-files.bend

Expected output (USER=tester):

Waiting...

Hello, tester

ping

file=bend-io-ok

tcp ready: serve_once(8080)

Notes:

- One event loop (Node-like): pure code parallel until the next effect.

- Handles are affine and single-owner. `File/Socket/Listener : Type` (linear: thread through one path; the error branch dies and drops the handle, which is free). `Chan : Data` (copyable with `+`, so both match branches may close it).

- `match` only scrutinizes parameters / pattern-bound vars, never a call result: every `File & Result / Socket & Result` pair is destructured via a helper taking the pair as a parameter. Defs are ordered before use.

```
import Base

def greet(name: String) -> IO(String): do IO<String>: IO.sleep(10) return "Hello, " ++ name

--- File write: File.write returns File & Result, threaded linearly. ---

def write_done(f: File, r: Result<&1, &1, U32 & String, Unit>) -> IO(File): match r: case Done{value}: IO.pure(File, f) case Fail{(code, message)}: IO.die(File, code, message)

def write_cont(pair: File & Result<&1, &1, U32 & String, Unit>) -> IO(File): match pair: case (f2, r): write_done(f2, r)

def write_all(f: File, data: String) -> IO(File): do IO<File>: pair : (File & Result<&1, &1, U32 & String, Unit>) <- File.write(f, data) write_cont(pair)
```

--- File read: close first (single use), then branch on the result. ---

```
def read_show(r: Result<&1, &1, U32 & String, String>) -> IO(Unit): match r: case Done{text}: IO.print("file=" ++ text) case Fail{(code, message)}: IO.die(Unit, code, message)

def read_close(f: File, r: Result<&1, &1, U32 & String, String>) -> IO(Unit): do IO<Unit>: u : Unit <- File.close(f) read_show(r)

def read_cont(pair: File & Result<&1, &1, U32 & String, String>) -> IO(Unit): match pair: case (f2, r): read_close(f2, r)

def file_roundtrip(path: String, data: String) -> IO(Unit): do IO<Unit>: w0 : File <- IO.try(File, File.open(path, "w")) w1 : File <- write_all(w0, data) File.close(w1)

def file_readback(path: String) -> IO(Unit): do IO<Unit>: r0 : File <- IO.try(File, File.open(path, "r")) pair : (File & Result<&1, &1, U32 & String, String>) <- File.read(r0, 256) read_cont(pair)
```

--- Explicit channel ping: Chan is Data, so `+ch` serves both branches. ---

```
def ping_show(m: Maybe<&1, String>, +ch: Chan(String)) -> IO(String): match m: case Some{text}: do IO<String>: Chan.close(String, ch) return text case None{}: do IO<String>: Chan.close(String, ch) IO.die(String, 3, "chan empty")

def ping_recv(ok: Bool, +ch: Chan(String)) -> IO(String): match ok: case True{}: do IO<String>: m : Maybe<&1, String> <- Chan.recv(String, ch) ping_show(m, ch) case False{}: do IO<String>: Chan.close(String, ch) IO.die(String, 2, "send failed")

def ping_send(+ch: Chan(String)) -> IO(String): do IO<String>: ok : Bool <- Chan.send(String, ch, "ping") ping_recv(ok, ch)

def chan_ping() -> IO(String): do IO<String>: ch : Chan(String) <- Chan.new(String, 1) ping_send(ch)
```

--- TCP echo server sketch (defined + checkable; call serve_once to run).

Pattern: io_http_server demo (TCP.listen + per-conn IO.spawn + accept loop). ---

```
def tcp_send_done(s: Socket, r: Result<&1, &1, U32 & String, Unit>) -> IO(Socket): match r: case Done{value}: IO.pure(Socket, s) case Fail{(code, message)}: IO.die(Socket, code, message)

def tcp_send_cont(pair: Socket & Result<&1, &1, U32 & String, Unit>) -> IO(Socket): match pair: case (s2, r): tcp_send_done(s2, r)

def tcp_send_all(s: Socket, data: String) -> IO(Socket): do IO<Socket>: pair : (Socket & Result<&1, &1, U32 & String, Unit>) <- TCP.send(s, data) tcp_send_cont(pair)

def tcp_close(s: Socket, r: Result<&1, &1, U32 & String, Unit>) -> IO(Unit): match r: case Done{value}: Socket.close(s) case Fail{(code, message)}: IO.die(Unit, code, message)

def tcp_reply_done(pair: Socket & Result<&1, &1, U32 & String, Unit>) -> IO(Unit): match pair: case (s2, r): tcp_close(s2, r)

def tcp_reply(s: Socket, msg: String) -> IO(Unit): do IO<Unit>: pair : (Socket & Result<&1, &1, U32 & String, Unit>) <- TCP.send(s, msg) tcp_reply_done(pair)

def tcp_echo_send(s: Socket, r: Result<&1, &1, U32 & String, String>) -> IO(Unit): match r: case Done{msg}: tcp_reply(s, msg) case Fail{(code, message)}: IO.die(Unit, code, message)

def tcp_echo_cont(pair: Socket & Result<&1, &1, U32 & String, String>) -> IO(Unit): match pair: case (s2, r): tcp_echo_send(s2, r)

def tcp_echo(s: Socket) -> IO(Unit): do IO<Unit>: pair : (Socket & Result<&1, &1, U32 & String, String>) <- TCP.recv(s, 1024) tcp_echo_cont(pair)

def serve_conn(li: Listener, sock: Socket) -> IO(Unit): do IO<Unit>: u : Unit <- Listener.close(li) tcp_echo(sock)

def accept_serve(li: Listener, r: Result<&1, &1, U32 & String, Socket>) -> IO(Unit): match r: case Done{sock}: serve_conn(li, sock) case Fail{(code, message)}: IO.die(Unit, code, message)

def accept_cont(pair: Listener & Result<&1, &1, U32 & String, Socket>) -> IO(Unit): match pair: case (li, r): accept_serve(li, r)

def serve_once(port: U32) -> IO(Unit): do IO<Unit>: listener : Listener <- IO.try(Listener, TCP.listen(port)) pair : (Listener & Result<&1, &1, U32 & String, Socket>) <- TCP.accept(listener) accept_cont(pair)
```

--- Entry: fork/join future + explicit channel + file roundtrip. ---

def main() -> IO(Unit): do IO<Unit>: name : String <- IO.try(String, IO.get_env("USER")) chan : Chan(String) <- IO.fork(String, greet(name)) IO.print("Waiting...") text : String <- IO.join(String, chan) IO.print(text) pong : String <- chan_ping() IO.print(pong) file_roundtrip("/tmp/bend-io-demo.txt", "bend-io-ok") file_readback("/tmp/bend-io-demo.txt") IO.print("tcp ready: serve_once(8080)")

F.36 · `bend-runtime/knowledge/06-examples/04-laws-proofs-app-loop.bend`

04-laws-proofs-app-loop.bend

Date: 2026-09-28

Sources: `bend guide` (bend 2.0.25); https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos)

(proof_insertion_sort, app_pong_game_2d, app_win_is_bug_2d); `bend base` (App/U32/List)

Run: bend 06-examples/04-laws-proofs-app-loop.bend [-- --gui]

Default (no args) prints the pure demos headlessly.

With `-- --gui` it opens the App window (needs Window libs: libx11-dev on Linux).

Expected output (no args):

42=42

law add_zero holds (bend PROOF gate)

pass --gui to open the app window

Workflow (law-driven development):

LAWS.bend imports code and states open claims (human-owned, AI does not touch).

PROOF.bend imports LAWS.bend and proves each as a same-name def (AI-owned).

`bend PROOF.bend` gates: fails while any law is open/false.

import Base

Proposition = type, proof = def. `{==}` closes a goal whose sides compute equal.

Recursion = induction; `%e : P` rewrites with a previous proof (`_` marks `b`).

law add_zero: for x: Nat {Nat.add(x, 0n) == x : Nat}

def add_zero(x): match x: case 0n: {==} case 1n+p: %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat} {==}

Templates (~) are compile-time generics: inlined per call-site, zero runtime

cost, callable many times (unlike affine closures). Used by List.map.

def twice(~f: U32 -> U32, x: U32) -> U32: f(f(x))

def inc(x: U32) -> U32: (x + 1 : U32)

def forty_two() -> U32: twice(~inc, 40)

App loop: frame = Image quadtree drawn by recursion; view hands state back

beside the image (affine); tick folds events until None quits.

Nested `Con{Close{}, rest}` pattern keeps one self-recursive def (mutual

recursion is forbidden; the shrinking param `events` stays first).

def tick(events: List<Event>, color: U32) -> IO(Maybe<U32>): match events: case Nil{}: IO.pure(Maybe<U32>, Some{color}) case Con{Close{}, rest}: IO.pure(Maybe<U32>, None{}) case Con{e, rest}: tick(rest, (color + 1 : U32))

def print_demos() -> IO(Unit): do IO<Unit>: IO.print("42=" ++ U32.show(forty_two())) IO.print("law add_zero holds (bend PROOF gate)") IO.print("pass --gui to open the app window")

def choose(args: List<String>) -> IO(Unit): match args: case Nil{}: print_demos() case Con{head, tail}: App.run(~U32, ~App{+s => (s, Pix{s}), tick}, "Hello", 256, 256, 0)

def main() -> IO(Unit): do IO<Unit>: args : List<String> <- IO.args() choose(args)

F.37 · `bend-runtime/knowledge/06-examples/05-bench-and-proof-templates.md`

Bench + proof templates

Date: 2026-09-28. Sources: https://github.com/bendlang/bend/tree/main/bench/runtime `(https://github.com/bendlang/bend/tree/main/bench/runtime)`, https://github.com/bendlang/bend/tree/main/bench/checker `(https://github.com/bendlang/bend/tree/main/bench/checker)`, https://github.com/bendlang/bend/tree/main/demos `(https://github.com/bendlang/bend/tree/main/demos)` (`pure_par_sum` , `proof_insertion_sort` , `app_win_is_bug_2d`), `bend guide` (bend 2.0.25), `bend base` .

BENCH SUITES (UPSTREAM)

Runtime (`bench/runtime` , 17 entries, each with C/TS/Lean twins; pins in `bench/*/_pin_/apple_m4*` for `gates/perf.ts`): `_pin_` , `bfs` , `editdist` , `gameoflife` , `hashmap` , `kmeans` , `lexer` , `mandelbrot` , `merkle` , `nbody` , `queens` , `raytrace` , `symreg` , `terrain` , `tree-bitonic` , `tree-matmul` , `tree-radix` . GPU-friendly = uniform numeric work (`mandelbrot` , `nbody` , `tree-matmul`); divergent search (`queens`) stays on CPU.

Checker (`bench/checker` , 6 entries): `_pin_` , `compute_1600` , `defs_12800` , `generics_3200` , `proofs_3200` , `trees_400` .

COPY-PASTE TEMPLATES

Parallel fork-join (aim ~4^7 leaves per bang, one fork tree ending in flat loops; host builds the tree, device reduces):

```
import Base
def pow2(+n: Nat) -> U32:
  match n:
    case 0n: 1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)
def main() -> IO(Unit):
  do IO<Unit>:
    IO.print(U32.show(pow2!(20n)))
```

In-place array (`Array<T> : Type` , single owner; write hands the array back; `a[i]` reads back `Array & elem` ; slot count is a power of two):

```
import Base
def main() -> Array<U32> & U32:
  a = [0 : U32*8n]
  a[5] <- 42
  a[5]
```

Law + proof (`bend PROOF.bend` gates deploys; recursion = induction; `%e : P` rewrites, `_` marks `b`):

```
import Base
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}
def add_zero(x):
  match x:
    case 0n: {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}
```

LAWS/PROOF split (law-driven development: humans state specs, AI implements + proves, compiler checks):

```
# LAWS.bend (human-owned, AI does not touch)
import Base
import ./code.bend as C
law sorted_keeps_count:
  for xs: List<U32>
  for v: U32
  {Count.of(C.sort(xs), v) == Count.of(xs, v) : Nat}
```

```
# PROOF.bend (AI-owned; same-name def fills each open law)
import Base
import ./LAWS.bend as L
def sorted_keeps_count(xs, v):
  match xs:
    case Nil{}: {==}
    case Con{h, t}:
      %sorted_keeps_count(t, v) : {Count.of(L.sort(Con{h, t}), v) == _ : Nat}
      {==}
```

RULES THAT BITE (ALL VERIFIED AGAINST BEND 2.0.25)

- `match` scrutinizes a parameter or pattern-bound var only; computed values go through a helper taking the value as a parameter.
- Destructure `File & Result` / `Socket & Result` pairs via `match pair: case (h2, r):` in a helper; never `pair.0` in term position.
- One use per affine binding across all match branches; `+x` (requires `Data`) for reused values. `File/Socket/Listener : Type` (linear: thread, don't share); `Chan : Data` (shareable with `+`).
- Defs must be ordered before use; no mutual recursion (merge into one def + selector arg); the shrinking argument stays first.
- No pure `x : T = v` lets mid-`do` after effects: hoist pure values into helper defs and call them from the block.
- Open `law` with no filling `def` makes the file report TODOs and blocks `main`; ship only proven laws in runnable files.

F.38 · bend-runtime/knowledge/07-smart-contracts-dynamic/01-kindelia-model.md

Kindelia model (S = sourced, D = derived design note)

Date: 2026-09-28. Sources: https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia), https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md), https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl (https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl), https://kindelia.org (https://kindelia.org), https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md).

NETWORK

Decentralized REPL: blocks are files settled via proof-of-work. No native coin, PoW forever, ~1s blocks, ~10k LOC Rust (vs 600k+ Go-Ethereum). Archived lineage: distinct from HVM2/Bend2; Bend 2 programs do not carry over from Kindelia/HVM.

STATEMENTS (EACH MAY CARRY A SECP256K1 SIGN)

- `ctr {Name fields}` — global constructor, arity <= 16.
- `fun (Name args) {rules} with {init}` — global stateful function.

- `run {IO_expr}` — metered evaluation; the signer is the subject.
- `reg Name {owner}` — namespace registration.

CONTRACTS = STATEFUL FUNCTIONS

State is any HVM structure (number/list/tree, no U256 serialization). `load` / `save` pin pointers (zero-cost SSTOREs: you pay for building state plus heap growth, not the pin). `IO.take` *removes* state (must `save` back, or it is emptied); `Load = Take + dup + Save`.

IO (PURE DESCRIPTION, SETTLED BY THE NODE)

`Done/Take/Save/Call/Subj/From/Tick/GetIdx/StmHash/Time/Meta/Hax` (see genesis.kdl). `Subj/From` = authenticated signer; `Tick/Time` = oracles; `StmHash/GetIdx` = statement addressing.

NAMESPACES

Dotted names (`Foo.Bar.cats`): deploy rights checked against the parent owner, top `Namer` in genesis; undotted names deployable by anyone. Mirrors Bend2 hub `name@ver` pointers over immutable `0x<hash>` packages (https://hub.bend-lang.com `(https://hub.bend-lang.com)`).

COSTS

Per-rule mana (`APP-LAM 2, APP-SUP 4, OP2-NUM 2, FUN-CTR 2+M, DUP-LAM 4`, ...), `mana_limit = 4M x (height+1)`, `bits_limit = 2048 x (height+1)` (~8GB/yr), unused allowance accumulates.

BEND2 ANALOGUE (D)

A contract is a module exposing `State/Msg/transition` compiled to interaction rules; the chain pins the state pointer plus the code hash; `run` = submit -> step -> meter -> pin-or-revert. Sketch:

```
import Base
type State is Data:
  Vault{balance: U32, owner: U32}
type Msg is Data:
  Deposit{amount: U32}
  Withdraw{amount: U32, requester: U32}
# Total, terminating transition: the ONLY place state changes (runtime pins).
# Auth arrives as an explicit envelope param (cf. SUBJ/FROM), never forged
# inside Msg. See 03-domain-types-hvm5-checklist.bend for the proven version.
def transition(s: State, m: Msg, a: Auth) -> State & (Unit | Reject):
  match s:
    case Vault{balance, owner}:
      apply_msg(owner, balance, m, a)
```

Rules (D): keep `transition` total + terminating (Bend2 termination is mandatory except `@unsafe`), so metering bounds cost, not halting. Keep consensus-step code pure: floats, wall-clock, RNG and foreign effects live outside `transition` (see `bend guide effects`). Gate publish/upgrade on `bend PROOF.bend` success.

F.39 · `bend-runtime/knowledge/07-smart-contracts-dynamic/02-hub-fork-upgrade.md`

Hub, forks, upgrades (S = sourced, D = derived)

Date: 2026-09-28. Sources: https://hub.bend-lang.com `(https://hub.bend-lang.com)`, https://github.com/bendlang/bend/blob/main/guide/GUIDE.md `(https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)` (Modules), https://bend-lang.com/bender/terms#s18 `(https://bend-lang.com/bender/terms#s18)` (publish terms), https://github.com/kindelia/Kindelia `(https://github.com/kindelia/Kindelia)` (`reg`, genesis `Namer`), https://ethereum.org/developers/docs/smart-contracts/upgrading `(https://ethereum.org/developers/docs/smart-contracts/upgrading)`, https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts `(https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts)`, https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades `(https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades)`, https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md `(https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md)`, https://en.wikipedia.org/wiki/Interaction_nets `(https://en.wikipedia.org/wiki/Interaction_nets)`.

CONTENT ADDRESSING (S)

"A package is a directory of files, named by a hash of all of its contents. That hash is its whole identity, and there are no names, versions or accounts." (https://hub.bend-lang.com (https://hub.bend-lang.com)). `import 0x<hash>/main.bend as P` fetches and hash-verifies. `bend main.bend --publish` prints that import line. Named pointers: `import name@ver/main.bend` after `bend login` + publish (`~/.bend/lib/names` caches; a version never moves). Publish is public + permanent; `LICENSE` files are hashed; no-LICENSE means MIT-0. Alias is file-local; dots are name chars; an open law in one file is fillable as `def M.name(..)` elsewhere, so proofs ship separately. The exact hash algorithm is not verified here: never cite one.

FORK = PIN + OVERRIDE (D)

```
import Base
import 0x<orig-hash>/main.bend as V1   # immutable (S syntax above)
def transition(s: State, m: Msg, a: Auth) -> State & (Unit | Reject):
  match m:
    case Patched{..}: <new logic>
    case _: V1.transition(s, m, a)     # delegate the remainder
```

Forking = publishing a new package that imports the old hash and re-exports a new `transition`. No proxy is needed for the fork case.

UPGRADE VIA INDIRECTION (D, THREE OPTIONS)

- **(a) Registry/dispatch (recommended).** A stable name (namespace or `name@ver` pointer) holds `current_hash`; `dispatch` tail-calls the target. Upgrade = registry tx (admin/multisig-gated). Audit = diff of two hashes.
- **(b) Versioned namespaces.** `MyApp.V1.transition`, `MyApp.V2.transition`; clients opt in; the registry maps "latest". Mirrors Kindelia `reg` (S).
- **(c) Migration.** V2 starts empty; explicit `migrate : V1.State -> V2.State`; old contract frozen. Most auditable, most coordination.

RULES (D)

- Keep `State a Data` tree; `migrate` total with `law migrate_roundtrip`, so `bend PROOF.bend` gates the upgrade.
- Never emulate `delegatecall` byte-patching: interaction rules are global and confluent (Interaction nets), so upgrades are new rules plus a pointer swap/redeploy, never live-rule edits. This removes storage-collision classes by construction (contrast EVM proxy/diamond, TON delayed `setCode`).
- Old frames finish old rules; new calls use new rules (Dart hot-reload analogy). `transition` stays total + terminating; metering bounds cost.
- Determinism gate for the step function: `(state, msg, env)` only; metered and bounded with revert (not partial commit) on exceed; canonical encodings; rule-set immutability outside upgrades.

F.40 · bend-runtime/knowledge/07-smart-contracts-dynamic/03-domain-types-hvm5-checklist.bend

03-domain-types-hvm5-checklist.bend

Date: 2026-09-28

Sources: https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) (WHITEBOOK, genesis.kdl);

https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md); bend base ;

https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos)

(proof_insertion_sort)

Run: bend 07-smart-contracts-dynamic/03-domain-types-hvm5-checklist.bend

Expected output: 157 (150 = 100 + 50 deposit; 7 = owner preserved on withdraw)

HVM5 API checklist (D, Kindelia-mapped) -- each item is either encoded below

or named so a runtime implementer can tick it off:

- [x] total `transition(State, Msg, Auth)` as the ONLY state-changing def

- [x] `State/Msg/Reject/Auth` as `Data` trees (no serialization layer)

- [x] auth envelope (`Auth` , cf. genesis SUBJ/FROM) passed explicitly, never forged in Msg

- [x] laws `vault_owner_kept` + `deposit_adds_balance` , proven below (`bend PROOF.bend` gate)

- [ ] `deploy_ctr/deploy_fun` immutable, namespace-gated, arity <= 16 (cf. `ctr` / `fun`)

- [ ] `submit(run_expr, sig)` signed runs (cf. signed `run` , Subj/From)

- [ ] `load/save` with take-vs-clone semantics explicit (take moves, load copies)

- [ ] oracles `tick/time/meta/stm_hash/get_idx` read-only (cf. TICK/TIME/META/STH/GIDX)

- [ ] metering per-rule mana table + revert (not partial commit) on exceed

- [ ] `fork(hash, overrides)` = new package importing the old hash (see 02-hub-fork-upgrade.md)

- [ ] `propose/approve/execute_upgrade` timelocked + `migrate` hook + `Upgraded(old,new)` event

- [ ] determinism: confluence-fixed semantics, canonical serialization, no

float/wall-clock/RNG/foreign code inside `transition`

```
import Base

type State is Data: Vault{balance: U32, owner: U32}

type Msg is Data: Deposit{amount: U32} Withdraw{amount: U32, requester: U32}

type Reject is Data: Unauthorized{who: U32} Insufficient{balance: U32, wanted: U32}
```

Capability-style auth evidence, decided by the (metered, effectful) edge and

consumed purely here -- cf. Kindelia SUBJ/FROM, which are also outside Msg.

```
type Auth is Data: MkAuth{is_owner: Bool, covered: Bool}

def withdraw_funds(ok: Bool, +owner: U32, +balance: U32, +amount: U32) -> State & (Unit | Reject): match ok: case True{}: (Vault{(balance - amount : U32), owner}, Inl{Unit{}}) case False{}: (Vault{balance, owner}, Inr{Insufficient{balance, amount}})

def apply_checked(ok: Bool, le: Bool, +owner: U32, +balance: U32, +amount: U32, +requester: U32) -> State & (Unit | Reject): match ok: case True{}: withdraw_funds(le, owner, balance, amount) case False{}: (Vault{balance, owner}, Inr{Unauthorized{requester}})

def apply_withdraw(+owner: U32, +balance: U32, +amount: U32, +requester: U32, a: Auth) -> State & (Unit | Reject): match a: case MkAuth{ok, le}: apply_checked(ok, le, owner, balance, amount, requester)
```

```
def apply_msg(+owner: U32, +balance: U32, m: Msg, a: Auth) -> State & (Unit | Reject): match m: case Deposit{+amount}: (Vault{(balance + amount : U32), owner}, Inl{Unit{}}) case Withdraw{+amount, +requester}: apply_withdraw(owner, balance, amount, requester, a)

def transition(s: State, m: Msg, a: Auth) -> State & (Unit | Reject): match s: case Vault{balance, owner}: apply_msg(owner, balance, m, a)

def st_owner(s: State) -> U32: match s: case Vault{balance, owner}: owner

def st_balance(s: State) -> U32: match s: case Vault{balance, owner}: balance

def next_of(r: State & (Unit | Reject)) -> State: match r: case (ns, res): ns

def next_state(s: State, m: Msg, a: Auth) -> State: next_of(transition(s, m, a))

def balance_of(r: State & (Unit | Reject)) -> U32: match r: case (ns, res): st_balance(ns)
```

Owner is preserved on every path (deposit + both withdraw branches).

Proof is fully structural: match params in binder order down to `{==}` leaves.

```
law vault_owner_kept: for s: State for m: Msg for a: Auth {st_owner(next_state(s, m, a)) == st_owner(s) : U32}

def vault_owner_kept(s, m, a): match s: case Vault{balance, owner}: match m: case Deposit{amount}: {==} case Withdraw{amount, requester}: match a: case MkAuth{ok, le}: match ok: case True{}: match le: case True{}: {==} case False{}: {==} case False{}: {==}

law deposit_adds_balance: for s: State for x: U32 for a: Auth {st_balance(next_state(s, Deposit{x}, a)) == U32.add(st_balance(s), x) : U32}

def deposit_adds_balance(s, x, a): match s: case Vault{balance, owner}: {==}

def demo_owner() -> U32: st_owner(next_of(transition(Vault{100, 7}, Withdraw{30, 7}, MkAuth{True{}, True{}})))

def demo_balance() -> U32: balance_of(transition(Vault{100, 7}, Deposit{50}, MkAuth{False{}, False{}}))

def main() -> U32: (demo_balance() + demo_owner() : U32)
```

F.41 · `bend-runtime/knowledge/08-roadmap-hvm5-spec/references-index.md`

References index (all fetched 2026-09-26, bend 2.0.25)

CLI (LOCAL DUMPS IN `02-GUIDES-CLI/RAW/`)

`bend guide`, `bend guide shaders`, `bend guide effects`, `bend base`, `bend base --types`, `bend --help`

ECOSYSTEM

- https://github.com/VictorTaelin `(https://github.com/VictorTaelin)`
- https://github.com/HigherOrderCO `(https://github.com/HigherOrderCO)` (+ /HVM1 /HVM2 /HVM3 /HVM4 /Bend1 /Kind /Bend2-stub)
- https://github.com/bendlang/bend `(https://github.com/bendlang/bend)` (`+ /tree/main/bend2`, `/tree/main/demos`, `/tree/main/guide`, `/tree/main/paper`, `/tree/main/bench`, `/tree/main/bench/runtime`, `/tree/main/bench/checker`, `/tree/main/evals`)
- Demos: `/tree/main/demos/pure_hvm5_mini`, `/pure_par_sum`, `/pure_par_sort`, `/proof_insertion_sort`, `/app_slash_boss_3d`, `/app_pong_game_2d`, `/app_ray_tracer_3d`, `/app_win_is_bug_2d`, `/io_http_server`
- https://github.com/HigherOrderCO/Bend1/tree/main/examples `(https://github.com/HigherOrderCO/Bend1/tree/main/examples)`
- https://github.com/VictorTaelin/Interaction-Calculus `(https://github.com/VictorTaelin/Interaction-Calculus)`, /WebMonkeys, /Cedille-Core
- https://github.com/kindelia/Kindelia `(https://github.com/kindelia/Kindelia)` (`+ /blob/master/WHITEBOOK.md`, `/blob/master/WHITEPAPER.md`)
- https://github.com/HigherOrderCO-archive/hvm-64 `(https://github.com/HigherOrderCO-archive/hvm-64)`

DOCS / PAPERS / SITES

- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md), `EFFECTS.md`, `SHADERS.md`
- https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend), `bend2/bend.lean`
- https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf), `paper/BendRT.pdf`
- https://bend-lang.com (https://bend-lang.com), https://hub.bend-lang.com (https://hub.bend-lang.com), https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18)
- https://paper.higherorderco.com (https://paper.higherorderco.com), https://higherorderco.com (https://higherorderco.com)
- Gists: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b), https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6)

THEORY

- https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets)
- https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718), https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432)
- http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf), https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711)

CONTRACTS / UPGRADES

- https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl (https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl), https://kindelia.org (https://kindelia.org)
- https://ethereum.org/developers/docs/smart-contracts/upgrading (https://ethereum.org/developers/docs/smart-contracts/upgrading)
- https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts (https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts)
- https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades (https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades)
- https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md (https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md)

COMMUNITY

https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues), https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp)

F.42 · `bend-runtime/knowledge/08-roadmap-hvm5-spec/spec-todo.md`

Spec TODO (toward full HVM5 dev spec)

PHASE 1 — GROUND TRUTH

- <input disabled="" type="checkbox"> Read `demos/pure_hvm5_mini/{main,LAWS,PROOF}.bend` line-by-line; map to API checklist in `07-smart-contracts-dynamic/hvm5-api-checklist.md`.
- <input disabled="" type="checkbox"> Read `paper/BendTT.pdf + paper/BendRT.pdf`; extract normative rules (live/dead, termination, scheduler, memory).
- <input disabled="" type="checkbox"> Read `bend2/comp.ts` emitters (C/Metal/CUDA/JS) for tag layout, task format, `term_peek/term_keep` ownership.
- <input disabled="" type="checkbox"> Resolve `bend.ts` vs `bend.lean` normativity; record version (`bend version`).

PHASE 2 — DOMAIN + API

- <input disabled="" type="checkbox"> Freeze `State/Msg/Reject/Query/Genesis` domain types (`07-smart-contracts-dynamic/domain-types-sketch.md` → real `.bend`).

- <input disabled="" type="checkbox"> Define `transition/init/migrate/view` signatures + totality/termination laws.
- <input disabled="" type="checkbox"> Define node API: deploy/register/submit/step/load/save/oracles/meter/query/fork/upgrade (checklist → OpenAPI/RPC).
- <input disabled="" type="checkbox"> Define canonical serialization + metering table (per-rewrite costs + heap-growth costs).

PHASE 3 — DYNAMIC UPGRADES

- <input disabled="" type="checkbox"> Implement fork template (`import 0x<hash>` + override + re-publish) as runnable demo.
- <input disabled="" type="checkbox"> Implement registry/dispatch + timelocked upgrade + `migrate` + `Upgraded` event.
- <input disabled="" type="checkbox"> Proof-gate CI: `bend PROOF.bend` must pass before publish/upgrade accepted.
- <input disabled="" type="checkbox"> In-flight semantics test (old frames finish old rules).

PHASE 4 — HARDEN

- <input disabled="" type="checkbox"> Determinism suite (no RNG/time/net/foreign/float in `transition` ; revert-on-exceed; canonical vectors).
- <input disabled="" type="checkbox"> Perf pins (`bench/*/_pin_` , `gates/perf.ts`) for target hardware.
- <input disabled="" type="checkbox"> Security review: namespace auth, handle unforgability, meter bypass, hub hash verification.

F.43 · `bend-runtime/knowledge/08-roadmap-hvm5-spec/what-we-know-vs-gaps.md`

Knowns vs gaps (for HVM5 spec work)

KNOWNS (S)

- Bend2 syntax/semantics/CLI/Base fully captured from `bend guide` + `bend base` (bend 2.0.25) in `02-guides-cli/` .
- BendRT model: one C file, 64-bit words, unified heap, no GC, flat state machine, fork-join + `!` GPU (papers BendTT/BendRT, `bend2/`).
- INET theory: nets, combinators, annihilate/commute/erase, oracle problem + affine dodge, HVM1→BendRT lineage.
- Ecosystem map: Taelin profile, HigherOrderCO 7 repos, live `bendlang/bend` layout, 16 demos, benches, evals.
- Contracts ancestor: Kindelia statements/costs/namespaces/IO; Bend hub hash imports + publish flows.

GAPS (MUST CLOSE BEFORE SPEC)

1. No verified standalone HVM5 repo/paper/spec — only `demos/pure_hvm5_mini` . Read its 3 files first.
2. No verified Bend hash-function spec (only `0x<hash>` syntax + hub behavior).
3. BendRT internals (ring-grid scheduler params, GPU lane config, tag bit-fields beyond paper abstract, CUDA/Metal kernels) need `paper/BendRT.pdf` + `bend2/comp.ts` code reading.
4. `bend.ts` vs `bend.lean` mismatch — which is normative for proofs?
5. Metering table for BendRT (Kindelia mana table is HVM-era, not BendRT).
6. Hub APIs (`BEND_HUB` , resolve/fetch/publish wire format) need network inspection.
7. Kindelia Block #5 TODO; no Kindelia→Bend2 migration doc exists.

F.44 · `bend-runtime/knowledge/09-hvm5-cluster-runtime/01-concepts.md`

Concepts

HOMOICONIC INET SUBSTRATE

HVM5 written in Bend2, compiling to INETs. Code and running graph share one representation (like IC `Term` : VAR/ERA/LAM/APP/SUP/DUP). Consequence: an update IS a graph rewrite, not a redeploy. Hole = global affine free VAR (or `REF` / `ALO` indirection node). Fill = atomic `sub[hole]=term` swap, O(1), no heap walk. Sources: `03-hvm-inet-theory/` , IC repo, HVM3/HVM4 `ALO` / `REF` lessons.

DUP = FORK, SUP = MERGE CANDIDATE

- `DUP (!&L{x,y}=v; b)`: lazily clones a subgraph layer-by-layer (commute if labels differ). Use for **fork**: dup live program graph into shadow; in-flight calls keep old side, tests drive new side.
- `SUP (&L{a,b})`: superposed alternatives distributing over consumers (`F{A B} -> {F A, F B}`). Use for **canary**: old+new coexist as superposed variants (different labels -> commute, same label -> annihilate on promote). Collapse/extract to commit, drop loser to rollback.
- Never duplicate a duplicator (dup-dup interference). Never publish linked heap with floating dups — collapse linked->quoted first.

HOLE-FILL VS FORK-TEST-MERGE

- **Hole-fill (instant)**: pure function swap via alias pointer (`Call 'Alias'` -> swap Book entry / binder SUB slot). Old frames drain (Erlang current/old), new calls use new rules. Requires: REF indirection (no inlining into live heap), slow-path fallback, version-pinned callers.
- **Fork-test-merge (safe)**: branch namespace + shadow storage + replay recent ingress (Dark flags + Unison refactor session + Erlang unpack/install/make_permanent). Merge = alias flip + extraction choice; rollback = flip back (old graph retained until drain).

REGISTRY: HASH VS NAME

Immutable blob by content hash (Unison `#hash`, IPFS CID, Agoric bundleID, Bend hub `0x<hash>`); mutable alias by signed record (IPNS sequence+validity, ENS resolver indirection, Dark `_vN` flags). Resolve once at bind time, pin hash, then use live cap. Never use DNS name as authority inside execution.

OCAP WIRES

Wire = unforgeable per-connection export ID (CapTP/Cap'n Proto), not a string. Invocation = eventual-send -> promise + pipelining + 3-party handoff + Drop/GC. Vat = one Bend2 program (turn-transactional, Goblins `$` sync intra-vat only, `<-` async inter-vat). Sturdy refs persistable; live refs connection-scoped. Attenuation via membranes/caretakers; upgrade gated by admin cap.

F.45 · bend-runtime/knowledge/09-hvm5-cluster-runtime/02-architecture-layers.md

Architecture layers

```
L7  API + CLI           create/update/fork/test/merge/invoke/register/resolve (04-api.md)
L6  Registry            name -> signed record -> code-hash (+petnames, TTL) (05-registry-naming.md)
L5  Orchestrator        fork-test-merge lifecycle, replay, promotion, rollback (06-update-fork-merge.md)
L4  OCap mesh           vat network, wire table (live/sturdy), eventual-send, pipelining, GC (07-ocap-wires.md)
L3  HVM5-in-Bend2       Book (static rules) + live heap; REF/ALO indirection; SUB-slot hole-fill; collapse/extract
L2  INET compiler       Bend2 -> interaction net (linearize_vars dup/era, Scott/num encodings); additive rule set
L1  Node runtime        event loop, meter (mana/bits), canonical serialization, snapshot (Take/Save), dual-generation routing
L0  Distribution        hash sync (Unison on-demand DAG fetch), op-log deltas (Automerge), DHT/edge validation
```

Rules:

- L3 keeps Book immutable; live heap points via alias. No fast-path inlining across alias boundary.
- L1 keeps current+old generations alive during cutover (Erlang/workerd dual routing).
- L0 validates at edge (typecheck + mana/space bound + namespace cap) before any node applies a fill.
- L4 is the ONLY inter-program call path. No direct global function names across vats.
- L6 resolve is bind-time only; hot path uses pinned hash + live wire.

F.46 · bend-runtime/knowledge/09-hvm5-cluster-runtime/03-domain-types.md

Domain types (sketch — freeze in Phase 2)

```
import Base
# Identity
type CodeHash is Data: H{bytes: List<U32>}          # content hash of compiled artifact (Unison/CID/bundleID style)
type ProgName is Data: N{labels: List<String>}      # DNS-like, e.g. ["dex","stable"]
type Version  is Data: V{major: U32, minor: U32, hash: CodeHash}
# Registry
type RegRecord is Data:
  R{name: ProgName, hash: CodeHash, seq: Nat, validity: Nat, ttl: Nat, pubkey: U32, sig: List<U32>}
type Petname is Data: P{local: String, target: CodeHash}   # local alias, Zooko triangle edge
# Programs / instances (Agoric bundleID -> installation -> instance)
type Program  is Data: Prog{hash: CodeHash, abi: List<String>}
type Instance is Data: Inst{id: U32, prog: CodeHash, state: StatePtr}
type StatePtr is Data: S{ptr: U32}                  # pinned heap pointer (Kindelia Save/Take)
type Facets is Data: F{public: Wire, creator: Wire, admin: Wire}
# Wires (OCap)
type Wire is Data: W{vat: U32, export_id: U32, rights: U32}  # unforgeable, per-connection
type SturdyRef is Data: SR{record: RegRecord, token: List<U32>}
# Updates (op-log, Automerge-style)
type Op is Data:
  AddRule{hash: CodeHash, rule: String}
  RetireRule{hash: CodeHash}
  Rewire{from: String, to: String}
  SwapAlias{name: ProgName, old: CodeHash, new: CodeHash, seq: Nat}
type UpdateReq is Data:
  U{target: ProgName, candidate: CodeHash, mode: UpdateMode, migr: CodeHash}
type UpdateMode is Data: Instant{} | ForkTestMerge{}
type MergeReq is Data: M{fork_id: U32, checks: List<String>, promote: Bool}
# Migration (Erlang code_change + Dark converters)
type MigrFun is Data: MF{from: CodeHash, to: CodeHash}  # must export: OldState -> Extra -> NewState
law migrate_total:
  for s: StatePtr
  for m: MigrFun
  exs s2: StatePtr
  {apply_migrate(m, s) == s2 : StatePtr}
```

Laws required: `transition_total`, `migrate_total`, `no_forge_wires` (wires unforgable, affine), `alias_flip_atomic`, `old_drains_new_serves`.

F.47 · bend-runtime/knowledge/09-hvm5-cluster-runtime/04-api.md

API (simple — full spec in implementation Phase 2)

PROGRAMS

- `create(name: ProgName, bundle: Bytes, init: State) -> (CodeHash, Instance, Facets)` — hash, store, spawn vat, register `name -> hash (seq 0)`.
- `update(req: UpdateReq) -> fork_id | applied` — `Instant`: validate -> atomic alias swap (hole-fill). `ForkTestMerge`: dup graph + shadow storage -> return fork_id.
- `test(fork_id, replay: List<Msg>) -> Report` — run recorded ingress + unit tests vs fork; compare old/new.
- `merge(req: MergeReq) -> CodeHash` — promote (alias flip, collapse sup, drain old) or abandon (drop fork, GC).
- `invoke(wire: Wire, fn: String, args: List<Value>) -> Promise<Value>` — eventual-send only. Pipelining: `invoke(invoke(w))` batches one turn.
- `share(wire: Wire, atten: Rights) -> Wire` — attenuate via membrane/caretaker; never share admin facet by default.

REGISTRY

- `register(name, hash, sig) -> RegRecord` — namespace ownership check (Kindelia `reg` + signer).
- `resolve(name) -> (CodeHash, RegRecord)` — verify sig + seq > cached + validity window; pin hash.
- `pin(hash) -> Program` — fetch missing DAG on demand (Unison sync); verify hash.
- `petname(local, hash)` / `whoami` — local aliases, no global authority.

NODES / OPS

- `snapshot(instance) -> StatePtr` (`Save`), `restore(ptr)`, `migrate(fork_id, MigrFun)`.
- `meter(program) -> (mana_table, bits_limit)`, `dry_run(hash, msgs) -> receipt`.
- `events: Upgraded(old,new) | Forked(id) | Merged(id) | RolledBack(id)`.

Errors are values (`Reject`), never stuckness. All mutating calls require the relevant facet cap (`admin` for upgrade, `creator` for share).

F.48 · bend-runtime/knowledge/09-hvm5-cluster-runtime/05-registry-naming.md

Registry + naming

Two layers, never conflated:

1. **Content layer (immutable)**: compiled Bend2 artifactaddrs by hash. `CodeHash = hash(bytes)` (Unison SHA3 / CID / Agoric bundleID / Bend hub `0x`). Fetch from anyone, verify locally. Dedupes cluster storage. Invocations pin this when determinism needed.
2. **Mutable layer (signed pointer)**: `RegRecord{name, hash, seq, validity, ttl, pubkey, sig}` (IPNS-style). Higher `seq` supersedes; `validity` favors availability, `ttl` favors freshness. DHT = consistency (slow, quorum); PubSub = speed (mesh fan-out, conn limits). Alternatives: DNSLink (fast, trusts DNS), ENS on-chain (`namehash -> {owner,resolver,ttl}`, resolver indirection rotates code-hash without rename).

Rules:

- Human names (`alice.eth`, `dex/stable`, petnames) are discovery hints resolved once at bind time -> pinned hash + live wire. Never authority inside execution.
- Zooko triangle: global-readable (ENS), secure (hash), memorable (petname) — pick two per layer; provide all three across layers (petname/edge-name/proposed-name over crypto IDs).
- Short TTL for fast rotation (revocation), long validity for availability. Old CIDs stay fetchable — revocation = rotate key / repoint + drop wire (GC propagates).
- Hot path: no registry lookup per call. Resolve -> pin -> invoke via wire table.

F.49 · bend-runtime/knowledge/09-hvm5-cluster-runtime/06-update-fork-merge.md

Update flows: instant hole-fill vs fork-test-merge

A. INSTANT HOLE-FILL (PURE / LOW-RISK)

1. `dry_run(candidate, sample_msgs)` + edge validation (typecheck, mana/space bound, namespace cap).
2. Parse candidate with pending-uses link phase; collapse linked->quoted; sweep floating dups.
3. Atomic step: swap `ALO` /Book pointer (or binder SUB slot) + bump version. Old generation kept (Erlang old/current).
4. Router: new calls -> new net; in-flight -> old net drains; then GC old.
5. Emit `Upgraded(old,new)`. Rollback = flip pointer back (old retained until drain).

B. FORK-TEST-MERGE (STATEFUL / RISKY)

1. `update(mode=ForkTestMerge)` -> dup live graph (DUP commute) + clone storage to shadow (Durable-Object-style) -> `fork_id`. Old version locked (Dark).
2. Apply candidate as superposed variant (different label -> commute; e-graph sibling; CRDT concurrent op). Never overwrite in place.
3. `test(fork_id)`: replay recent ingress log + unit tests + `bend PROOF.bend` gate; compare receipts; cost (mana) check.
4. `merge(promote=true)`: alias flip + collapse/extract winner + `Take->migrate->Save` state transaction (`code_change` mandatory). `promote=false`: drop fork, GC.
5. Three-phase commit (Erlang): `unpack (fork) -> install (reversible) -> make_permanent (merge)`.

State rule: update tuple is `(rules, migrated_state)` — never rules alone. Versioned stores (`Store-vHash` + forward/backward converters, Dark dual-read) during transition; background rewrite after no `v_old` callers remain.

F.50 · bend-runtime/knowledge/09-hvm5-cluster-runtime/07-ocap-wires.md

OCap wires (internal invocation)

- Each Bend2 program = vat (isolated turn loop). Intra-program: direct call. Inter-program: `<-` eventual-send only -> `Promise`. Sync `$` across vats forbidden (would break atomicity).
- Wire = `(vat, export_id, rights)` in per-connection cap table (CapTP L1-L4 / Cap'n Proto). Marshal: pass-by-copy data + pass-by-reference presences (Agoric `marshal`). No ambient DNS/cookies.
- Features: promise pipelining (`invoke(invoke(w))` one round-trip), 3-party handoff (A gives B ref to C => B dials C directly), distributed GC (`Drop/Release` on unreachable), sleepy passivation (`save/restore` sturdy refs), membranes/caretakers for attenuation, `whenBroken` contagion.
- Sharing: holder of `public` facet may `share(attenuated)`; `admin` (upgrade) never leaves governance vat/committee unless explicitly delegated; invitations/seats revocable.
- Implementation: wire table per node `{live: conn-scoped, sturdy: persistable token}`; kernel routes inter-vat messages (SwingSet-style); turn rollback on throw (Goblins `bcom` behavior update = upgrade primitive).
- Rule: resolve name -> hash -> spawn/bind -> handshake yields wire. Thereafter invoke by wire only.

F.51 · bend-runtime/knowledge/09-hvm5-cluster-runtime/08-similar-projects.md

Similar projects — distilled lessons (full research via subagents 2026-09-26)

HOT-SWAP RUNTIMES

- **Erlang/OTP** (erlang.org code_loading/release_handling/release_structure): current+old code server, `m:f()` switch point, `.appup` (`load_module` vs `update+code_change`), `unpack->install->make_permanent`. Copy: two-version rules + mandatory `code_change` + 3-phase commit.
- **Elixir/BEAM**: same mechanism; ecosystem defaults to blue/green — warning: make migration the default path, not opt-in.
- **Unison** (unison-lang.org, github.com/unisonweb/unison): SHA3 content-addressed defs, names = metadata, update = add hash + remap name, on-demand DAG sync, durable typed storage. **Highest relevance — copy identity + sync model.**
- **Darklang** (github.com/darklang/dark, 50ms deploy blog): AST-diff atomic deploy, flags + replay before switch, `f_v0/f_v1` coexistence, versioned datastores + converters. Copy: flag-wrapped edits + traffic replay.
- **Cloudflare workerd** (github.com/cloudflare/workerd): isolates per version, dual-generation routing, capability bindings (`workerd.capnp`), compatibility dates, Durable Objects (named singleton + SQLite). Copy: isolate routing fallback + bindings + compat-date pin.
- **Wasmtime** (github.com/bytecodealliance/wasmtime): no hot purge; swap `Instance` behind atomic pointer, WIT-stable swap. Use WASM as freeze/export boundary, not live-update primitive.

OCAP + NAMING

- **E/CapTP** (ocapn.org, capnproto.org/rpc.html): eventual-send + pipelining + 3-party handoff + sturdy refs. Wire = unguessable ref.
- **Goblins** (spritely.institute, spritely-core.html, petnames.html): vat+turn transactions, `$` sync / `<-` async, `bcom` become, live vs sturdy refs, membranes, petnames over DIDs.
- **Agoric** (github.com/agoric/agoric-sdk): `bundleID -> installation -> instance`, `public/creator/admin` facets, `upgradeContract` + durable baggage, null-upgrade GC.
- **Cap'n Proto RPC**: per-connection cap table, pipelining default, Drop/GC, schema evolution.
- **IPFS/IPNS** (docs.ipfs.tech/concepts/ipns, github.com/ipfs/kubo): CID immutable + IPNS signed `{seq,validity,ttl}` mutable; DHT consistency vs PubSub speed; `validity` vs TTL tradeoff.
- **ENS** (docs.ens.domains): `namehash -> {owner,resolver,ttl}`, registrar hierarchy, resolver indirection rotates code without rename. Human layer only.

GRAPH / MERGE

- **IC/HVM3/HVM4** (VictorTaelin/Interaction-Calculus, HVM3 IC.md/HVM.md, HVM4 memory.md/core.md): hole = free VAR / ALO / REF; fill = SUB swap; floating dups need pre-pass sweep; `DSU/DDU` runtime labels for canary; keep slow-path fallback (inlining kills updates).
- **Kindelia** (WHITEBOOK): `(rules, state)` tuple updates; `Take->migrate->Save`; forbid redefine `fun`, use versioned names + alias; zero-cost save = pointer swap.

- **egg/egglog** (github.com/egraphs-good/egg): additive rules, e-class siblings, extract-to-commit. Trial = superpose, promote = extract.
- **Automerge** (github.com/automerge/automerge): op-log (`add/retire/rewire/swap-alias`), delta sync, deterministic merge. Encode fills as ops, not snapshots.
- **Holochain** (README-level): DHT-validated entries, agent chains; edge validation lesson (typecheck+mana+cap before apply).

F.52 · `bend-runtime/knowledge/09-hvm5-cluster-runtime/09-implementation-plan.md`

Implementation plan + repo folder structure

PHASES

1. **Ground truth**: read `pure_hvm5_mini` 3 files, BendTT/BendRT PDFs, `bend2/comp.ts` emitters; resolve `bend.ts` vs `bend.lean` ; record `bend version` .
2. **Core types + registry**: freeze `03-domain-types.md` into real `.bend` ; implement hash store + `RegRecord` + petnames; `register/resolve/pin` with sig + seq checks.
3. **Vat + wires**: one program = vat; wire table (live/sturdy); eventual-send + pipelining + Drop/GC; `invoke/share` ; turn rollback.
4. **Updates**: REF/ALO indirection (no inline across alias); hole-fill atomic swap + dual-generation drain; fork (dup + shadow storage) + replay + `bend PROOF.bend` gate + merge/abandon; `code_change` mandatory for stateful.
5. **Harden**: determinism suite, metering (mana/bits), canonical vectors, compat-date pins, perf pins, security review (namespace auth, wire unforgability, meter bypass, hash verification).

SUGGESTED IMPLEMENTATION REPO LAYOUT (FOR BUILDER LLM)

```
hvm5-cluster/
  README.md  AGENTS.md  LAWS.bend  PROOF.bend
  bend/          # .bend sources: types.bend transition.bend registry.bend wires.bend migrate.bend
  runtime/       # HVM5-in-Bend2: book/ heap/ scheduler/ meter/ snapshot/
  cluster/       # orchestrator: fork/ test/ merge/ replay/ drain/ events/
  registry/      # hash store, RegRecord, resolvers (ipns/ens/petname adapters)
  api/           # RPC + CLI: create/update/test/merge/invoke/register/resolve
  tests/         # .bend tests ending `#|` + replay fixtures + canary scenarios
  docs/          # architecture, api, domain types, update flows, ocap, registry
```

Conventions: `LAWS.bend` human-owned claims, `PROOF.bend` AI-owned + `bend PROOF.bend` gate before publish/merge; parallelize divide-and-conquer; `bend guide` canonical.

F.53 · `bend-runtime/knowledge/09-hvm5-cluster-runtime/10-meta-prompt-for-builder-llm.md`

META PROMPT for builder LLM (copy-paste to a less intelligent LLM)

> Paste everything below the line into the builder model. It is self-contained. Replace `[REPO_ROOT]` with your checkout.

You are implementing the **HVM5 Bend2 cluster runtime**: live Bend2 programs as updatable smart-contract-like services on an HVM5 runtime written in Bend2, compiling to homoiconic INETs.

1. GOAL (ACCEPTANCE)

- Simple API: `create / update / test / merge / invoke / register / resolve / share / snapshot / migrate` .
- Two update modes, zero downtime: (a) **instant hole-fill** = atomic pointer swap filling a typed hole in the running graph (graph transformation, instantaneous); (b) **fork-test-merge** = dup live graph into shadow fork (dup/sup-like), test (replay + proofs), then merge (collapse + alias flip) or abandon.
- Registry: public programs register under DNS-like name OR hash. `name -> signed record -> code-hash` ; content blobs immutable by hash.

- Internal invocation between running programs ONLY via **OCap wires** (shared unforgeable wire to invoke; eventual-send + pipelining; no ambient authority).

2. CONCEPTS YOU MUST IMPLEMENT (NO SUBSTITUTES)

- Homoiconic INETs: code == graph. Hole = global affine free VAR / REF / ALO indirection node. Fill = single atomic SUB swap (O(1), no heap walk). Parse new subgraph with pending-uses, collapse linked->quoted, sweep floating DUPs before publish.
- dup=fork (lazy layer-by-layer clone via commute), sup=canary (old+new superposed, different labels commute; promote = same-label annihilate/collapse/extract). Never duplicate a duplicator. Never inline across alias boundary (keep slow-path fallback).
- State migration explicit: update `= (rules, migrated_state)` tuple via `Take->migrate->Save` ; every stateful program exports `code_change: OldState -> Extra -> NewState` + forward/backward converters. Laws `transition_total` , `migrate_total` , `no_forge_wires` , `alias_flip_atomic` must hold (`bend PROOF.bend` gates merge).
- Registry two layers: immutable `CodeHash` blobs (verify locally, dedupe, on-demand DAG fetch) + mutable signed `RegRecord{name,hash,seq,validity,ttl,pubkey,sig}` (higher seq wins). Petnames local. Resolve once at bind -> pin hash -> invoke by wire.
- Wires: `(vat, export_id, rights)` per-connection table; live (ephemeral) vs sturdy (persistable token) refs; `invoke` eventual-send -> Promise; support pipelining, 3-party handoff, Drop/GC, membranes/caretakers, admin-facet-gated upgrade. One program = one vat (turn-transactional; intra-vat direct call, inter-vat async only, rollback on throw).

3. LAYERS (RESPECT BOUNDARIES)

L7 API/CLI > L6 Registry > L5 Orchestrator (fork/test/merge/replay/drain) > L4 OCap mesh > L3 HVM5-in-Bend2 (Book immutable + live heap via alias) > L2 INET compiler (linearize dup/era, additive rules) > L1 Node (event loop, mana/bits meter, canonical serialization, dual-generation current+old routing) > L0 Distribution (hash sync deltas, op-log, edge validation: typecheck + mana/space + namespace cap before apply).

4. DOMAIN TYPES (START FROM THESE, FREEZE BEFORE CODING)

CodeHash, ProgName, Version, RegRecord, Petname, Program, Instance, StatePtr, Facets{public,creator,admin}, Wire, SturdyRef, Op{AddRule,RetireRule,Rewire,SwapAlias}, UpdateReq{target,candidate,mode,migr}, UpdateMode{Instant,ForkTestMerge}, MergeReq, MigrFun. See `09-hvm5-cluster-runtime/03-domain-types.md` for the Bend2 sketch.

5. API SEMANTICS (IMPLEMENT EXACTLY)

create(name,bundle,init)->(hash,instance,facets); update(req)->fork_id|applied; test(fork_id,replay)->Report (recorded ingress + unit tests + proofs + mana check); merge(promote)->hash (alias flip + collapse + drain old, or drop fork); invoke(wire,fn,args)->Promise (pipelined); share(wire,atten)->Wire; register/resolve/pin/petname; snapshot/restore/migrate; meter/dry_run; events Forked/Merged/RolledBack/Upgraded. Errors as values. Mutations require facet caps.

6. UPDATE FLOWS (IMPLEMENT BOTH)

A) Instant: dry_run + validate -> link-phase parse -> atomic Book/alias swap + version bump -> new calls to new net, in-flight drains old -> GC old -> Upgraded event (rollback = flip back). B) Fork-test-merge: dup graph + clone storage -> fork_id (lock old) -> superpose candidate -> test (replay + PROOF + receipts + cost) -> merge (alias flip + extract + Take/migrate/Save) or abandon (GC). Three-phase: unpack->install(reversible)->make_permanent.

7. REPO LAYOUT TO CREATE

hvm5-cluster/{README.md,AGENTS.md,LAWS.bend,PROOF.bend,bend/,runtime/{book,heap,scheduler,meter,snapshot},cluster/{fork,test,merge,replay,drain,events},registry/{store,resolvers},api/{rpc,cli},tests/,docs/}. LAWS human-owned, PROOF AI-owned; `bend PROOF.bend` must pass before merge. Tests are `.bend` files ending `#|` .

8. PRIOR ART TO MIRROR (LESSONS, NOT DEPS)

Unison (content hashes + on-demand sync), Erlang OTP (current/old + code_change + unpack/install/make_permanent), Darklang (flags + replay + versioned stores), workerd (isolates + bindings + compat dates + Durable Objects), CapTP/Goblins/Agoric/Cap'n Proto (wires, vats, facets, pipelining, membranes), IPNS/ENS (signed pointers + resolver indirection), IC/HVM3/HVM4/Kindelia (SUB-slot fill, ALO/REF, Take/Save, no-redefine + alias), egg (additive + extract), Automerge (op-log deltas).

9. DEFINITION OF DONE

- Demo: deploy v1 counter, fork to v2, replay 1000 ingress, merge with zero failed calls, rollback path tested, registry resolve + wire invoke across two programs, meter receipts, PROOF gate enforced. Document deviations from this prompt.

Work in small commits. When stuck, re-read `bend guide` (canonical) and the `09-hvm5-cluster-runtime/` docs. Do not invent hash algorithms or standalone HVM5 specs — record gaps in `docs/GAPS.md`.

F.54 · `bend-runtime/knowledge/09-hvm5-cluster-runtime/README.md`

HVM5 Bend2 Cluster Runtime — goal overview

End goal: a cluster where Bend2 programs run as live smart-contract-like services on an HVM5 runtime (HVM5 itself written in Bend2, compiling to homoiconic INETs), with:

1. **Simple API** to create / update a Bend2 program with zero downtime.
2. **Two update modes**:
   - (a) **Fork-test-merge** (dup/sup-like): fork live graph into shadow (dup), test, then merge (sup collapse / alias flip).
   - (b) **Instant hole-fill**: fill a typed hole in the running program graph — a graph transformation (single atomic pointer swap), instantaneously done.
3. **Registry**: public programs register under DNS-like name or hash (`name -> hash`, mutable pointer -> immutable blob).
4. **OCap invocation**: programs invoke each other only via shared wires (unforgeable capabilities), eventual-send + pipelining, no ambient authority.

Non-goals: consensus/coin (see Kindelia for that layer); GPU scheduling (see BendRT); full proof automation (see LAWS/PROOF workflow).

Read order: `01-concepts.md -> 02-architecture-layers.md -> 03-domain-types.md -> 04-api.md -> 05-registry-naming.md -> 06-update-fork-merge.md -> 07-ocap-wires.md -> 08-similar-projects.md -> 09-implementation-plan.md -> 10-meta-prompt-for-builder-llm.md`

F.55 · `bend-runtime/knowledge/ALL-COMBINED.md`

Bend2 / HVM5 / INET — ALL COMBINED

Generated 2026-09-26 from bend2-knowledge/ — 49 files combined. Source file path precedes each section.

CONTENTS

1. `README.md`
2. `00-start-here/glossary.md`
3. `00-start-here/how-to-use-this-knowledge-base.md`
4. `00-start-here/overview.md`
5. `01-bend-language/apps-modules-packages.md`
6. `01-bend-language/arrays.md`
7. `01-bend-language/functions-closures-templates-recursion.md`
8. `01-bend-language/io-concurrency-monads.md`
9. `01-bend-language/laws-proofs.md`
10. `01-bend-language/parallelism-fork-join-gpu.md`
11. `01-bend-language/syntax-reference.md`
12. `01-bend-language/types-quantities-kinds.md`
13. `02-guides-cli/base-library.md`
14. `02-guides-cli/cli-tooling.md`
15. `02-guides-cli/effects-summary.md`
16. `02-guides-cli/guides-index.md`
17. `02-guides-cli/raw/BASE-FULL-DUMP.md`
18. `02-guides-cli/raw/BASE-TYPES-DUMP.md`
19. `02-guides-cli/raw/EFFECTS-CLI-DUMP.md`

20. `02-guides-cli/raw/GUIDE-CLI-DUMP.md`
21. `02-guides-cli/raw/SHADERS-CLI-DUMP.md`
22. `02-guides-cli/shaders-summary.md`
23. `03-hvm-inet-theory/bend-to-hvm-compilation.md`
24. `03-hvm-inet-theory/hvm-lineage.md`
25. `03-hvm-inet-theory/interaction-nets-primer.md`
26. `03-hvm-inet-theory/optimal-reduction-oracle.md`
27. `03-hvm-inet-theory/symmetric-combinators-dup-sup-era-con.md`
28. `04-runtime-bendrt-hvm5/bendrt-architecture.md`
29. `04-runtime-bendrt-hvm5/ffi-determinism.md`
30. `04-runtime-bendrt-hvm5/hvm5-mini-reference.md`
31. `04-runtime-bendrt-hvm5/memory-64bit-heap-no-gc.md`
32. `04-runtime-bendrt-hvm5/schedulers-cpu-gpu.md`
33. `05-ecosystem/bendlang-bend-repo.md`
34. `05-ecosystem/higherorderco-org.md`
35. `05-ecosystem/papers-formalization-community.md`
36. `05-ecosystem/victor-taelin.md`
37. `06-examples/README-examples-index.md`
38. `06-examples/app-io-examples.md`
39. `06-examples/bench-suite-templates.md`
40. `06-examples/parallel-examples.md`
41. `06-examples/proof-examples.md`
42. `07-smart-contracts-dynamic/content-addressing-hub.md`
43. `07-smart-contracts-dynamic/domain-types-sketch.md`
44. `07-smart-contracts-dynamic/fork-upgrade-patterns.md`
45. `07-smart-contracts-dynamic/hvm5-api-checklist.md`
46. `07-smart-contracts-dynamic/kindelia-model.md`
47. `08-roadmap-hvm5-spec/references-index.md`
48. `08-roadmap-hvm5-spec/spec-todo.md`
49. `08-roadmap-hvm5-spec/what-we-know-vs-gaps.md`

FILE: README.md

Source: `bend2-knowledge/README.md`

Bend2 / HVM5 / INET Knowledge Base

Comprehensive overview of Bend2, HVM5/BendRT, interaction nets, runtime, interfaces, and dynamic smart-contract patterns. Built 2026-09-26 from CLI (`bend 2.0.25`) + GitHub sources.

LAYOUT

- `00-start-here/` — overview, how to use, glossary
- `01-bend-language/` — syntax, types, parallelism, proofs, IO, apps, modules
- `02-guides-cli/` — full `bend guide` / `shaders` / `effects` dumps in `raw/` + summaries, CLI tooling, Base library
- `03-hvm-inet-theory/` — interaction nets, combinators (dup/sup/era/con), optimal reduction, HVM lineage, compilation
- `04-runtime-bendrt-hvm5/` — BendRT architecture, HVM5-mini, memory, schedulers, FFI, determinism
- `05-ecosystem/` — VictorTaelin, HigherOrderCO, bendlang/bend, papers, community
- `06-examples/` — curated demos, benches, templates
- `07-smart-contracts-dynamic/` — Kindelia model, hub content-addressing, fork/upgrade, domain types, HVM5 API checklist
- `08-roadmap-hvm5-spec/` — knowns vs gaps, spec TODO, reference index

SOURCES (ALL FETCHED, NO INVENTED URLS)

- CLI: `bend guide`, `bend guide shaders`, `bend guide effects`, `bend base`, `bend base --types`, `bend --help` (bend 2.0.25) — raw dumps in `02-guides-cli/raw/`
- https://github.com/VictorTaelin (https://github.com/VictorTaelin)
- https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) (org) + HVM1/HVM2/HVM3/HVM4/Bend1/Kind
- https://github.com/bendlang/bend (https://github.com/bendlang/bend) (live Bend2 repo: `bend2/`, `demos/`, `guide/`, `paper/`, `bench/`, `evals/`)
- https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) (esp. `pure_hvm5_mini`, `pure_par_*`, `proof_*`, `app_*`, `io_*`)
- https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) + WHITEPAPER/WHITEBOOK
- https://bend-lang.com (https://bend-lang.com), https://hub.bend-lang.com (https://hub.bend-lang.com)
- Theory: https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets), Lafont POPL 1990 / 1997, HVM2 paper, BendTT/BendRT PDFs

> Bend 2 is a new language. Bend 1 programs and HVM do not carry over. (bendlang/bend README Limitations)

QUICK START

1. Read `00-start-here/overview.md`
2. Run `bend guide` yourself; compare with `02-guides-cli/raw/GUIDE-CLI-DUMP.md`
3. Pick a track: language (`01-`), runtime (`03-` + `04-`), contracts (`07-`)
4. Spec work starts at `08-roadmap-hvm5-spec/spec-todo.md`

FILE: 00-start-here/glossary.md

Source: *`bend2-knowledge/00-start-here/glossary.md`*

Glossary

BEND2 LANGUAGE

- **Affine**: use at most once (default). Dropping is free.
- **Quantity**: `-x` erased (types/proofs only), `x` affine, `+x` reusable (needs `Data`).
- **Kind**: `Type = Kind(&1)` linear, `Data = Kind(&2)` copyable, `Kind(a)` generic, `a <&> b` minimum.
- **Template (`~f`)**: compile-time inlined arg, must be closed, per-site copy. How `List.map` works.
- **Law**: `law name: for x: T … claim` — human-owned open claim in `LAWS.bend`.
- **Proof**: `def name(…)` of law's type — AI-owned in `PROOF.bend`. `{==}` reflexivity, `%e : P` rewrite (`_` = new side), `?name` / `?TODO` holes.
- **Parallel let**: `a b = f(x) g(y)` — fork/join, promises independence + balance.
- **Bang**: `f!(x)` — run subtree on GPU (else CPU threads).
- **Array**: `Array<T> : Type`, single owner, `[v : T*n]` / `[v : T^d]`, `a[i]` / `a[i] <- v`, read returns `Array & elem`.
- **do-block**: `do M<xs..,R>:` over `M.bind` / `M.pure`; `x : A <- m`, `x : A = v`, `return`.
- **Handle**: affine opaque (`File/Socket/Window/Chan`), handed back beside result, unforgable.

INET / HVM

- **Agent/cell/node**: 1 principal + n aux ports. Only principal–principal pairs rewrite.
- **Active pair / redex**: principal–principal connection; unit of parallel work.
- **CON (γ)**: constructor (Bend1: `( )`; IC: LAM/APP side).
- **DUP (δ)**: duplicator; `let {a b} = x` (consume) vs `SUP {a b}` (build). Same node, polarity decides.
- **SUP**: superposition `{a b}` — parallel alternatives distributing over consumers.
- **ERA (ε, `*`)**: eraser; propagates + frees.
- **REF (`@foo`)**: top-level def pointer; CALL expands, VOID deletes, COPY fast-path.
- **NUM/OPR/SWI**: native numbers, prim ops, zero-vs-succ branch.
- **Annihilate**: same family meets → cancel, auxes wire straight.

- **Commute/copy**: different families meet → duplicate through each other.
- **Oracle/brackets**: Lamping bookkeeping to match dups; avoided by affine restriction.
- **Beta-optimal**: Lévy family sharing; HVM hits it on affine/higher-order fragment.
- **Fork-join**: BendRT scheduler unit; join waits for both children.
- **Unified heap**: one address space for CPU+GPU; no separate copies.
- **BendRT**: Bend2 flat evaluator (not interaction-net reducer). One C file → clang/Metal/CUDA.
- **HVM5-mini**: `demos/pure_hvm5_mini` — minimal pure runtime model in Bend.

CONTRACTS

- **ctr/fun/run/reg**: Kindelia statements (define ctor, stateful fn, metered eval, namespace).
- **mana/bits**: compute cost vs heap growth limits.
- **Registry/dispatch**: stable name → current code hash pointer for upgrades.
- **Fork**: new package importing old hash + overriding defs.
- **Migration**: explicit `V1.State -> V2.State` total function + law.

FILE: 00-start-here/how-to-use-this-knowledge-base.md

Source: `bend2-knowledge/00-start-here/how-to-use-this-knowledge-base.md`

How to use this knowledge base

FOR LEARNING BEND2

1. Install: `curl -fsSL https://bend-lang.com/install.sh | sh`
2. `bend guide` (compare `02-guides-cli/raw/GUIDE-CLI-DUMP.md`), `bend guide shaders`, `bend guide effects`
3. `bend base --types` then `bend base Map` for the `Type.verb` scheme
4. Run `demos/io_hello_world`, `demos/pure_par_sum`, `demos/proof_insertion_sort` (see `06-examples/`)

FOR RUNTIME / HVM5 SPEC WORK

1. `03-hvm-inet-theory/` for nets + combinators + optimal reduction
2. `04-runtime-bendrt-hvm5/` for BendRT, memory, schedulers, FFI
3. `07-smart-contracts-dynamic/` for Kindelia + fork/upgrade patterns
4. `08-roadmap-hvm5-spec/spec-todo.md` for gaps before writing the spec

CONVENTIONS

- `(S)` = sourced fact with URL; `(D)` = design suggestion, not spec.
- No invented URLs: every link was fetched 2026-09-26 or comes from CLI dumps.
- `bend guide` text is canonical for syntax; GitHub `guide/GUIDE.md` mirrors it: https://github.com/bendlang/bend/blob/main/guide/GUIDE.md `(https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)`
- Bend1/HVM CLIs (`bend run-c`, `hvm run`) do NOT apply to Bend2.

REFRESHING

```
bend guide > bend2-knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md
bend guide shaders > bend2-knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md
bend guide effects > bend2-knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md
bend base > bend2-knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md
bend version
```

FILE: 00-start-here/overview.md

Source: `bend2-knowledge/00-start-here/overview.md`

Overview — what Bend2 / HVM5 / INETs are

ONE-PARAGRAPH VERSION

Bend2 is a Python-shaped, Haskell/Lean-semantics, Rust-like affine language with dependent types, mandatory termination, and explicit parallelism (`a b = f(x) g(y)` fork/join, `f!(x)` GPU). **BendRT** is its parallel CPU/GPU runtime (one C file = CPU program + GPU kernel, single shared heap, no GC, 64-bit words). **HVM5** exists in-repo only as `demos/pure_hvm5_mini` — a minimal pure model, not a released standalone runtime. **Interaction nets/combinators** (Girard/Lafont: CON/DUP/SUP/ERA, annihilate vs commute) are the historical HVM1–HVM4 execution substrate; BendRT keeps the affine/no-GC/parallel ideas but is a flat evaluator, not an interaction-net reducer. **Kindelia** is the prior functional blockchain (stateful pure functions + metered `run` , mana/bits, namespaces) and the template for dynamic Bend2 contracts via content-hashed hub imports (`import 0x<hash>/main.bend`).

GENERATIONS (DO NOT MIX)

Gen	Repo	Runtime	Status
HVM1 (2022)	https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1)	Rust, lazy, parallel, beta-optimal fragment	Archived predecessor
HVM2 (2024)	https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)	Rust/C/CUDA IR, 7 node kinds	Bend1 target, LTS paper
HVM3	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3)	Haskell, Interaction Calculus	WIP
HVM4	https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)	C single-file IC runtime	Experimental
Bend1	https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1)	On HVM2	Superseded
Bend2	https://github.com/bendlang/bend (https://github.com/bendlang/bend)	**BendRT** (TS compiler, C/Metal/CUDA/JS)	**Live (bend 2.0.25)**
Kindelia	https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia)	HVM opcodes, PoW, 1s blocks	WIP reference for contracts

CORE CLAIMS

- Pure + affine by default (`is Data` copyable, `is Type` linear). `match` frees consumed node; only `+` values refcounted. No GC.
- Termination mandatory (structural recursion, decreasing arg first); `@unsafe / def f?` escapes proofs.
- Laws (`LAWS.bend` , human) + proofs (`PROOF.bend` , AI) gated by `bend PROOF.bend` .
- Parallelism is explicit + must be balanced (binary fork-join, contention-free; GPU cube ~16384 lanes, aim 4^7 leaves).
- Effects only via `IO` + foreign `.c` / `.js` twins; host event loop; JS can import `.bend` .
- Packages are content hashes on https://hub.bend-lang.com (https://hub.bend-lang.com); `bend --publish` prints the `import 0x…` line.

WHERE TO GO NEXT

- Language: `01-bend-language/`
- CLI + Base: `02-guides-cli/cli-tooling.md` , `base-library.md`
- Deep INET: `03-hvm-inet-theory/`
- Runtime + HVM5: `04-runtime-bendrt-hvm5/`
- Contracts/forking: `07-smart-contracts-dynamic/`
- Spec work: `08-roadmap-hvm5-spec/`

FILE: 01-bend-language/apps-modules-packages.md

Source: `bend2-knowledge/01-bend-language/apps-modules-packages.md`

Apps (pure graphics) + modules/packages

APPS

```
import Base
def tick(events: List<Event>, color: U32) -> IO(Maybe<U32>):
  match events:
    case Nil{}: IO.pure(Maybe<U32>, Some{color})
    case Con{Close{}, rest}: IO.pure(Maybe<U32>, None{})
    case Con{e, rest}: tick(rest, (color + 1 : U32))
def main() -> IO(Unit):
  App.run(~U32, ~App{+s => (s, Pix{s}), tick}, "Hello", 256, 256, 0)
```

- Frame = `Image` quadtree (`Pix{color}` / `Qua{tl,tr,bl,br}`), drawn by recursion, parallel if wanted.
- `App.run` calls `view` then `tick` per frame until `None` . `view` hands state back beside image (affine).
- Events `Key/Mouse/Move/Close` . Underneath `Window.open/frame/close` , `Audio.open/write/close` .
- See `demos/app_pong_game_2d` (complete), `demos/app_slash_boss_3d` (flagship).

MODULES

```
# math.bend
import Base
def square(+x: U32) -> U32: (x * x : U32)
# main.bend
import Base
import ./math.bend as M
def main() -> U32: M.square(7)
```

- File = module, alias file-local, dots in names are chars (`U32.show` needs no module).
- Open law in one file fillable as `def M.name(..)` elsewhere → proofs ship separately.

PACKAGES (HUB)

- `import 0x<hash>/main.bend as P` — content hash, fetched + verified from hub.
- `bend main.bend --publish` uploads + prints that line. Named: `import name@ver/main.bend` , `bend main.bend --publish name@ver` after `bend login` .
- Publish public + permanent (https://bend-lang.com/bender/terms#s18 `(https://bend-lang.com/bender/terms#s18)`); `LICENSE` files hashed; no-LICENSE = MIT-0.
- Hub has no names/versions/accounts/search yet — hash is identity; `~/.bend/lib/names` caches names.

FILE: 01-bend-language/arrays.md

Source: `bend2-knowledge/01-bend-language/arrays.md`

Arrays (pure in-place mutation)

```
import Base
def main() -> Array<U32> & U32:
  a = [0 : U32*8n]   # 8 copies of 0
  a[5] <- 42         # in-place rewrite
  a[5]               # read index 5
```

- `Array<T> : Type` → single owner; write hands back same array (no copy); read hands back `Array & elem` (else array would be gone).
- Indexes wrap. Write-then-statement re-binds: `a[5] <- 42` alone = `a = a[5] <- 42` ; as last statement = written array.
- Slot count after `*` is power of two; `[0 : U32^3n]` names depth instead.
- `a[i]` sugar assumes `Array<U32>` ; else `Array.get` (`Data` elems) / `Array.swap` + `Array.set` + `Array.clone` .
- See `bend base` for `Array.*` .

FILE: 01-bend-language/functions-closures-templates-recursion.md

Source: `bend2-knowledge/01-bend-language/functions-closures-templates-recursion.md`

Functions, closures, templates, recursion

CLOSURES

```
import Base
def adder(k: U32) -> U32 -> U32:
  x => (x + k : U32)
def main() -> U32:
  add2 = adder(2)
  add5 = adder(5)
  add5(add2(1))
```

- Affine: call at most once even if captures are `Data`. Only top-level defs callable freely.
- Partial applications (`U32.add(2)`) are closures too.

TEMPLATES (COMPILE-TIME GENERICS)

```
def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))
def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

- `~f` inlined per call-site, zero runtime cost, callable many times (unlike closures).
- Template params first; `~` arg must be closed (only top-level defs, no caller locals).
- Template may call only templates above it. This is how `List.map` is written.

RECURSION + TERMINATION

```
def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}: acc
    case Con{h, t}: sum(t, (acc + h : U32))
```

- Tail calls → loops. Termination mandatory; mutual recursion forbidden (else proofs unsound).
- Rule: recursive call args read left-to-right, unchanged until one is a smaller part from pattern match; rest free. Put shrinking param first.
- Server loops count down `Nat` fuel. Two mutually recursive fns → one def + selector arg.
- `@unsafe / def f?` recurses freely but outside proof guarantees. Types aren't code so datatype order is free.

FILE: 01-bend-language/io-concurrency-monads.md

Source: `bend2-knowledge/01-bend-language/io-concurrency-monads.md`

IO, concurrency, monads

EFFECTS

```
import Base
def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1000)
    return "Hello, " ++ name
def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER"))
    chan : Chan(String) <- IO.fork(String, greet(name))
    IO.print("Waiting...")
    text : String <- IO.join(String, chan)
    IO.print(text)
```

- Every bind annotated; `x : T = v` pure let mid-block.
- Fallible = `Result<.., U32 & String, A>`; `IO.try` unwraps or exits, `IO.die` exits custom.
- `IO.args` = CLI args less runtime options (`--` ends them).
- Handles affine opaque → every effect hands back beside result; unforgable/unreusable.

CONCURRENCY MODEL

- One event loop (Node-like): pure code parallel on all cores up to next effect; socket/sleep/channel waiters step aside.
- `IO.fork` → channel future; `IO.join` waits. Underneath `IO.spawn`, `Chan.new/send/recv/close`.
- Ends when all done; deadlock reported when rest all wait.

CUSTOM EFFECTS

- Each Base effect = def with `import "./x.js" + .c` twin, host fn = lowercase dots→underscores.
- Add own the same way. Only event loop runs them (proofs/termination/GPU never touch host).
- Reverse: JS `import Game from "./game.bend"` (with `bend2/main.ts` preloaded); ctors `{$: "Name", field: value}`, `Nat as BigInt`; `Array` crosses without copy (copy first if kept).

MONADS (GENERIC `DO`)

```
def add_strs(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a)
    y : U32 <- U32.read(b)
    return (x + y : U32)
```

`do M<xs..,R>:` → `M.bind` / `M.pure`; leading args (here `&2`) passed along. Works for `IO`/`Maybe`/`Result`/custom.

FILE: 01-bend-language/laws-proofs.md

Source: `bend2-knowledge/01-bend-language/laws-proofs.md`

Laws and proofs (law-driven development)

```
import Base
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}
def add_zero(x):
  match x:
    case 0n: {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}
```

MODEL

- Proposition = type, proof = def. `{a == b : T}` equality; `{==}` when both sides compute equal.
- `match` refines goal per case; recursion = induction; `%e : P` rewrite (`_` marks `b`); `%e@E : P` named.
- `exs y: T` witness → return `(y, proof)`. `?name` prints goal, `?TODO` open hole, law with no def = open claim.
- `match e:` no cases closes `e : Empty` branch. `{a != b : T} = {a==b:T} -> Empty`. `Equal.sym/trans/cong` in Base.
- No tactics. Failed step prints expected vs observed.

WORKFLOW

- `LAWS.bend` imports code, states open claims (human-owned, AI doesn't touch).
- `PROOF.bend` imports `LAWS.bend`, proves each as same-name def (AI-owned + code).
- `bend PROOF.bend` gates: fails while any law open/false, prints success when all hold.
- Bend refuses `PROOF.bend` beside non-importing `LAWS.bend`.
- Dependent types are just defs returning `Type` (`def IsEven(n: Nat) -> Type:`).

> Vision: law-driven development — humans state precise specs, AI implements + proves, compiler checks to mathematical correctness.

FILE: 01-bend-language/parallelism-fork-join-gpu.md

Source: `bend2-knowledge/01-bend-language/parallelism-fork-join-gpu.md`

Parallelism (fork-join + GPU)

```
import Base
def pow2(+n: Nat) -> U32:
  match n:
    case 0n: 1
    case 1n+p:
      a b = pow2(p) pow2(p)   # parallel call
      (a + b : U32)
def main() -> IO(Unit):
  IO.print(U32.show(pow2!(20n)))  # ! runs on GPU
```

CONTRACT

1. Calls independent (always true: pure + affine).
2. Roughly same time (yours to keep; unbalanced → sub-ideal speedup).

Scheduler is contention-free binary fork-join: each task handed to a core exactly once, never moved. Fast + GPU-friendly but requires balanced work.

! PLACEMENT

- `pow2(20n)` parallel on CPU; `pow2!(20n)` on GPU (that call + inner parallel calls).
- Unified heap: zero-cost CPU→GPU on Apple M-series unified memory; paged/managed on CUDA.
- No GPU → `!` runs on CPU (still parallel). JS target ignores it (sequential).
- GPU shines on uniform numeric work (mandelbrot, nbody); divergent (n-queens) stays on CPU.
- `+` value read by every lane costs an atomic per read.

RULES OF THUMB (SEE `02-GUIDES-CLI/SHADERS-SUMMARY.MD`)

- Aim ~4^7 leaves per bang ≈ 16384 lanes; one fork tree ending in flat loops.
- Host builds scene into lists/quadtrees; device draws; never fork per pixel.
- Read `bend guide shaders` before parallel apps.

FILE: 01-bend-language/syntax-reference.md

Source: `bend2-knowledge/01-bend-language/syntax-reference.md`

Bend2 syntax reference (from `bend guide`)

Source: `02-guides-cli/raw/GUIDE-CLI-DUMP.md` (= `bend guide`, bend 2.0.25). Also https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)

TOP LEVEL

```
import Base
import ./file.bend as M
type D<a, -A: Kind(a)> is Kind(a):
  K{x: A, xs: List<a, A>}
def f(x: A, -y: B, +z: C) -> T:
def f(x, y):                        # fills law named f
def t(~g: A -> B, x: A) -> B:       # template
law f:
  for x: A
  for y: B where P(y)
  exs z: C
  T
@unsafe def f(x: A) -> T:
def e(x: A) -> IO(B):
  import "./e.c"
  import "./e.js"
```

TYPES

```
Type  Data  Kind(q)                 # Type=Kind(&1), Data=Kind(&2)
Quant  &0  &1  &2  a <&> b
A -> B  @x:A -> B  @-x:A -> B       # plain, dependent, erased-dependent
A & B  &x:A -> B  A | B             # pair, dep pair, sum
D<A>  +D<A>  D<&2, A>
{a == b : T}  {a != b : T}
```

TERMS

```
42  1.5  3n  'c'  "s"               # U32 F32 Nat Char String
[a, b]  h <> t  (a, b)
K{a, b}  x => e  +x => e
f(a, b)  f!(a)  t(~g, a)
(a + b * c : T)  {x : T}
[v : T*n]  [v : T^d]  a[i]  a[i] <- v
{==}  %e : P; e2  %e@E : P; e2
?name  ?TODO
```

STATEMENTS

```
x = v  +x = v  -x = v
(a, b) = v  K{x, y} = v
a b = f(x) g(y)                     # parallel let
match a b:
  case K{x, _} 1n+p:
do M<xs.., R>:
  x : A <- m
  x : A = v
  m                                 # Unit step
  return v
```

OPERATOR RULES

- Inside `(.. : T)`: `+ - * / %` → `T.add..mod`, `.&. .|. .^.` bits, `<< >>` shifts (by `Nat`), `< <= > >=` → `T.is_lt` family. Without `: T` they belong to `Nat`.
- `&& ||` on `Bool`, `++` on `String` anywhere. Spaces required both sides.
- Value equality is `T.is_eq(a,b)`; `==` only in types.

- `Nat` literal `> 256n` is `U32.to_nat(n)` underneath, up to `4294967295n`.

MATCH RESTRICTIONS

- Inspect a parameter or pattern-bound var, never a computed value (`match f(x):` rejected → helper).
- Scrutinees follow binder order; `let` may not precede `match` on a parameter.
- No `if`; match on `True{}` / `False{}`.

FILE: 01-bend-language/types-quantities-kinds.md

Source: `bend2-knowledge/01-bend-language/types-quantities-kinds.md`

Types, quantities, kinds

EXAMPLE

```
import Base
type Shape is Data:
  Circle{r: U32}
  Square{s: U32}
def area(x: Shape) -> U32:
  match x:
    case Circle{+r}:
      (3 * r * r : U32)
    case Square{+s}:
      (s * s : U32)
```

QUANTITIES

```
# -A erased (types/proofs only, deleted at runtime)
#  n affine (default, at most once; dropping free)
# +x reusable (requires Data; pays refcount)
def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n: Nil{}
    case 1n+p:
      x <> replicate(A, p, x)
```

- Matching `+` gives `+` fields; on plain values write `+r` in pattern to make reusable.
- Functions, arrays, IO handles are `Type` → never copyable.

KINDS

```
def length(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}: 0n
    case Con{h, t}: 1n+length(a, A, t)
```

- `Type = Kind(&1)`, `Data = Kind(&2)`.
- `type List<a, -A: Kind(a)> is Kind(a)` → list as reusable as elements. `List<U32> = List<&1,U32>`; `+List<U32> = List<&2,U32>`.
- Two element types combine with `a <&> b` (minimum).
- Bare `a` param `= -a: Quant`.

PRACTICAL RULES

- Bend does almost no inference — annotate when checker stuck (`{3 : U32}`, `(a+b : U32)`).
- `is Data` vs `is Type` on declaration caps reuse.
- `Nat` unary (`Zero/Succ`, `3n`); `U32 = Word(32n)`; `F32` axiomatic (nothing provable); `String = SNil/SCon` char list (slow).

FILE: 02-guides-cli/base-library.md

Source: bend2-knowledge/02-guides-cli/base-library.md

Base library summary (bend base , 3006 lines)

Full: raw/BASE-FULL-DUMP.md ; types: raw/BASE-TYPES-DUMP.md ; upstream: https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend)

NAMING SCHEME

Every def Type.verb ; same verbs across Nat/U32/F32 : add sub mul div mod , bits and or xor not shl shr (U32), cmp→Cmp , is_eq/is_ne/is_lt/is_le/is_gt/is_ge→Bool , show→String , read: String→Maybe . Conversions T.to_x/from_x . Operators sugar need (.. : T) .

TYPES

Empty/Unit/Bool/Cmp/Either/Sigma/Nat/Maybe/Result/List/Word/U32/F32/Char/String/Array/Image/Event/Map (+ IO.OP/IO/App laws, File/Socket/Listener/Window/Audio/Chan handles).

COLLECTIONS

- List : map/filter/foldl/foldr/any/all/find , sort(~A,~le,+xs) mergesort with fuel.
- Map (String keys): new/set/get/has/del/keys ; get takes default, get/has return map beside result. Set on top.
- Array : get/swap/set/clone ; see 01-bend-language/arrays.md .
- String : SNil/SCon char list — correct but slow.

EFFECTS SURFACE

IO.print/write/print_err/get_env/args/random_u32/spawn/sleep/now/thread_count , Chan.new/send/recv/close , File.open/read/read_bytes/read_at/size/write/write_bytes/close , TCP.listen/accept/connect/send/recv/poll , UDP.bind/send_to/recv_from/poll , Socket/Listener.close , Window.open/frame/set_title/close , Audio.open/write/close , Process.run(program,args,input,max_output,timeout_ms) (no shell, needs glibc 2.34+).

TRY

```
bend base --types | head -n 60
bend base Map | head -n 100
bend base IO | head -n 100
bend base List | head -n 120
```

FILE: 02-guides-cli/cli-tooling.md

Source: bend2-knowledge/02-guides-cli/cli-tooling.md

CLI + tooling (bend --help , bend 2.0.25)

```
bend <file.bend> [args]       check, then run main with args
bend <file.bend> -o <out>     build binary; <out>.c emits C, <out>.js JS
bend <file.bend> --check-only check + imports; run nothing
bend <file.bend> --publish    publish + imports to hub
bend <page.html> -o <dir>     bundle page importing .bend
bend base [--types|<name>]    print Base / types / name subtree
bend guide                    print guide
bend update                   install latest (curl | sh shown first)
bend version                  print version
```

RUN SEMANTICS

- main: IO runs compiled; main: value normalized by checker (slow for big work) + printed; no main just checks.
- Native needs clang 14+ (19+ with !); ! builds file.gpu beside binary (Metal macOS, CUDA 12 Linux); Linux Window libx11-dev , Audio libasound2-dev .

- `./file --threads 8`, `./file --gpu off`, `./file --gpu 4GB`.

EXTRA COMMANDS (REPO CHANGELOG 2.0.0–2.0.29)

`bend main.bend --publish [name@ver]`, `bend login | bend link`, `bend guide shaders|effects`, `bend PROOF.bend` (law gate), gates `test.ts/perf.ts/repo.ts/ping.ts`, formatter `tools/bend-fmt-lsp`, community LSP https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp).

ENV / PACKAGING

`BEND_HUB/BEND_ORIGIN/BEND_LIB`, `~/.bend/lib/names`, `User-Agent: bend/<ver>`, daily version check (`BEND_NO_TELEMETRY=1` off). Installed `bend` ignores `bunfig.toml` / `.env`; `bun bend2/main.ts` does not — check untrusted code with `bend`.

LIMITS (WONTFIX HIGHLIGHTS)

Verbose (no inference), no typeclasses, templates = macros only, affine closures/arrays unshareable, no computed scrutinee, only `Nat/U32/F32` (F32 axiomatic), slow char-list strings, small Base, one GPU/event-loop, one C file (no incremental, slow native compile — use JS for dev), terse errors, no debugger/profiler/REPL/test-framework, balanced-parallelism only, atomics sharing experimental `@unsafe`.

FILE: 02-guides-cli/effects-summary.md

Source: `bend2-knowledge/02-guides-cli/effects-summary.md`

Effects guide summary (`bend guide effects`)

Full: `raw/EFFECTS-CLI-DUMP.md` (129 lines).

MODEL

- Every Base effect = Bend def with `import "./x.c" + import "./x.js"` twins, host fn named lowercase dots→underscores (e.g. `IO.print` → `io_print`).
- Only the event loop runs them; proofs/termination/GPU never touch host code.
- Handles affine opaque; JS↔Bend values cross without copy for `Array` (copy first if kept).

WRITING CUSTOM EFFECTS

1. Declare `def My.eff(x: A) -> IO(B):` with both imports.
2. Implement C twin (native binary) + JS twin (JS target / dev-server).
3. Keep consensus-step code pure — foreign effects allowed only outside deterministic `transition` (see `07-smart-contracts-dynamic/`).
4. Minimal example pattern is `clock.c/.js` style (`io_work/io_wait_on/io_park_on`, `io_eff(CID(..))` — see upstream https://github.com/bendlang/bend/blob/main/guide/EFFECTS.md (https://github.com/bendlang/bend/blob/main/guide/EFFECTS.md)).

Read the full dump + upstream EFFECTS.md before writing one.

FILE: 02-guides-cli/guides-index.md

Source: `bend2-knowledge/02-guides-cli/guides-index.md`

Guides index (what `bend guides` can do — bend 2.0.25)

> Note: CLI is `bend guide` (singular). `bend guides` errors (`no such file: guides`). Raw dumps: `raw/`.

Command	File	What it covers
`bend guide`	`raw/GUIDE-CLI-DUMP.md` (621 lines)	Full language: hello, types, closures, recursion, parallelism, arrays, quantities, kinds, templates, laws/proofs, IO/concurrency, monads, apps, Base, modules, tooling, syntax ref, under-the-hood, further reading
`bend guide shaders`	`raw/SHADERS-CLI-DUMP.md` (268 lines)	Perf tutorial from `demos/app_slash_boss_3d` (120 FPS pure Bend): fork tree + flat loops, 4^7 leaves, host-build/device-draw
`bend guide effects`	`raw/EFFECTS-CLI-DUMP.md` (129 lines)	C/JS side of custom effects
`bend base`	`raw/BASE-FULL-DUMP.md` (3006 lines)	Entire Base library
`bend base --types`	`raw/BASE-TYPES-DUMP.md`	Types only
`bend base <Name>`	—	One name + subtree (e.g. `bend base Map`, `bend base IO`)
`bend --help`	`raw/CLI-HELP-DUMP.txt`	CLI usage

Repo mirrors: https://github.com/bendlang/bend/tree/main/guide (https://github.com/bendlang/bend/tree/main/guide) (`GUIDE.md`, `SHADERS.md`, `EFFECTS.md`), https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend)

WHAT BEND CURRENTLY CAN DO (GUIDE CAPABILITIES)

- Python-shaped pure programs with Lean-like proofs + C/CUDA speed; `bend file.bend` check+run, `-o` native/C/JS, HTML bundling.
- Numbers `Nat/U32/F32`, strings, lists, maps/sets, arrays with in-place ops, images/quadtrees, events, audio.
- Effects: files, TCP/UDP, sockets, windows, audio, processes, time, random, env, channels, spawn/sleep. No TLS/HTTP-lib/JSON/regex (use foreigns).
- GPU via `!` (Metal macOS, CUDA 12 Linux `/usr/local/cuda`), `--threads`, `--gpu off/4GB`.
- Proofs gate releases; hub publish by hash or `name@ver`.

FILE: 02-guides-cli/raw/BASE-FULL-DUMP.md

Source: `bend2-knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md`

Types

=====

Data

----

type Empty is Data:

type Unit is Data: Unit{}

type Bool is Data: False{} True{}

type Cmp is Data: LT{} EQ{} GT{}

type Either<a, b, -A: Kind(a), -B: Kind(b)> is Kind(a <&> b): Inl{value: A} Inr{value: B}

```
type Sigma<a, b, -A: Kind(a), -B: @-x: A -> Kind(b)> is Kind(a <&> b): Tuple{fst: A, snd: B(fst)}

type Nat is Data: Zero{} Succ{pred: Nat}

type Maybe<a, -A: Kind(a)> is Kind(a): None{} Some{value: A}

type Result<a, b, -E: Kind(a), -A: Kind(b)> is Kind(a <&> b): Fail{error: E} Done{value: A}

type List<a, -A: Kind(a)> is Kind(a): Nil{} Con{head: A, tail: List<a, A>}

law Word: for n: Nat Data

type Word.Nil is Data: WNil{}

type Word.Con<-p: Nat> is Data: WCon{head: Bool, tail: Word(p)}

type U32 is Data: U32{data: Word(32n)}

type F32 is Data: F32{data: Word(32n)}

type Char is Data: Chr{code: U32}

type String is Data: SNil{} SCon{head: Char, tail: String}

type Array<-T: Type> is Type: ALeaf{value: T} ANode{xs: Array<T>, ys: Array<T>}

type Image is Data: Pix{color: U32} Qua{tl: Image, tr: Image, bl: Image, br: Image}

type Event is Data: Key{code: U32, down: Bool} Mouse{x: U32, y: U32, button: U32, down: Bool} Move{x: U32, y: U32}
Close{}

type Map<a, -V: Kind(a)> is Kind(a): MTip{} MLeaf{key: String, val: V} MNode{pos: Nat, lo: Map<a, V>, hi: Map<a, V>}
```

Effects

A handle is opaque, a law with no constructor and no field, so no

program forges one, and linear (Type), so none copies or reuses one.

```
law File: Type

law Socket: Type

law Listener: Type

law Window: Type

law Audio: Type
```

A channel is Data: the computations it links each hold a copy.

```
law Chan: for -A: Type Data

type IO.OP<-R: Type> is Type: Emit{value: R} Halt{code: U32, message: String}

law Pair: for -A: Type for -B: Type Type

law IO: for -A: Type Type

type App<-S: Type> is Type: App{view: S -> Pair(S, Image), tick: List<Event> -> S -> IO(Maybe<S>)}
```

Defs

====

Word

```
def Word(n): match n: case 0n: Word.Nil case 1n+p: Word.Con
```

Pair

```
def Pair(A, B): Sigma<&1, &1, A, _ => B>

def Exists(-A: Type, -B: @-x: A -> Type) -> Type: Sigma<&1, &1, A, B>

def Or(-A: Type, -B: Type) -> Type: Either<&1, &1, A, B>

def Pair.fst(-A: Type, -B: Type, p: A & B) -> A: (a, b) = p a

def Pair.snd(-A: Type, -B: Type, p: A & B) -> B: (a, b) = p b
```

Effects

```
def IO(A): @-R: Type -> @k: (A -> IO.OP<R>) -> IO.OP<R>

def IO.pure(-A: Type, x: A) -> IO(A): R => k => k(x)

def IO.bind(-A: Type, -B: Type, m: IO(A), f: A -> IO(B)) -> IO(B): R => k => m(R, x => f(x, R, k))

def IO.print(text: String) -> IO(Unit): import "./effs/print.c" import "./effs/print.js"

def IO.write(text: String) -> IO(Unit): import "./effs/write.c" import "./effs/write.js"

def IO.print_err(text: String) -> IO(Unit): import "./effs/print_err.c" import "./effs/print_err.js"

def IO.get_env(name: String) -> IO(Result<&1, &1, U32 & String, String>): import "./effs/get_env.c" import "./effs/get_env.js"

def IO.args() -> IO(List<String>): import "./effs/args.c" import "./effs/args.js"

def IO.die(-A: Type, code: U32, msg: String) -> IO(A): R => k => Halt{code, msg}

def IO.pass(-A: Type, r: Result<&1, &1, U32 & String, A>) -> IO(A): match r: case Done{value}: IO.pure(A, value) case Fail{(code, message)}: IO.die(A, code, message)

def IO.try(-A: Type, act: IO(Result<&1, &1, U32 & String, A>)) -> IO(A): IO.bind(Result<&1, &1, U32 & String, A>, A, act, IO.pass(A))

def IO.random_u32() -> IO(Result<&1, &1, U32 & String, U32>): import "./effs/random_u32.c" import "./effs/random_u32.js"

def IO.spawn(-A: Type, act: IO(A)) -> IO(Unit): import "./effs/spawn.c" import "./effs/spawn.js"

def IO.sleep(ms: U32) -> IO(Unit): import "./effs/sleep.c" import "./effs/sleep.js"

def IO.now() -> IO(Nat): import "./effs/now.c" import "./effs/now.js"

def Chan.new(-A: Type, room: U32) -> IO(Chan(A)): import "./effs/chan_new.c" import "./effs/chan_new.js"

def Chan.send(-A: Type, chan: Chan(A), value: A) -> IO(Bool): import "./effs/chan_send.c" import "./effs/chan_send.js"

def Chan.recv(-A: Type, chan: Chan(A)) -> IO(Maybe<&1, A>): import "./effs/chan_recv.c" import "./effs/chan_recv.js"

def Chan.close(-A: Type, chan: Chan(A)) -> IO(Unit): import "./effs/chan_close.c" import "./effs/chan_close.js"

def IO.fork.go(-A: Type, act: IO(A), chan: Chan(A)) -> IO(Chan(A)): +c = chan sent = IO.bind(A, Bool, act, x => Chan.send(A, c, x)) IO.bind(Unit, Chan(A), IO.spawn(Bool, sent), u => IO.pure(Chan(A), c))
```

```
def IO.fork(-A: Type, act: IO(A)) -> IO(Chan(A)): IO.bind(Chan(A), Chan(A), Chan.new(A, 1), IO.fork.go(A, act))

def IO.join.go(-A: Type, chan: Chan(A), got: Maybe<&1, A>) -> IO(A): match got: case None{}: IO.die(A, 1, "IO.join: the channel was closed") case Some{value}: IO.bind(Unit, A, Chan.close(A, chan), u => IO.pure(A, value))

def IO.join(-A: Type, +chan: Chan(A)) -> IO(A): IO.bind(Maybe<&1, A>, A, Chan.recv(A, chan), IO.join.go(A, chan))

def File.open(path: String, mode: String) -> IO(Result<&1, &1, U32 & String, File>): import "./effs/file_open.c" import "./effs/file_open.js"

def File.read(file: File, max: U32) -> IO(File & Result<&1, &1, U32 & String, String>): import "./effs/file_read.c" import "./effs/file_read.js"

def File.read_bytes(file: File, max: U32) -> IO(File & Result<&1, &1, U32 & String, List<&2, U32>>): import "./effs/file_read.c" import "./effs/file_read.js"

def File.read_at(file: File, offset: U32, max: U32) -> IO(File & Result<&1, &1, U32 & String, List<&2, U32>>): import "./effs/file_read.c" import "./effs/file_read.js"

def File.size(file: File) -> IO(File & Result<&1, &1, U32 & String, U32>): import "./effs/file_size.c" import "./effs/file_size.js"

def File.write(file: File, data: String) -> IO(File & Result<&1, &1, U32 & String, Unit>): import "./effs/file_write.c" import "./effs/file_write.js"

def File.write_bytes(file: File, data: List<&2, U32>) -> IO(File & Result<&1, &1, U32 & String, Unit>): import "./effs/file_write.c" import "./effs/file_write.js"

def File.close(file: File) -> IO(Unit): import "./effs/file_close.c" import "./effs/file_close.js"

def TCP.listen(port: U32) -> IO(Result<&1, &1, U32 & String, Listener>): import "./effs/tcp_listen.c" import "./effs/tcp_listen.js"

def TCP.accept(listener: Listener) -> IO(Listener & Result<&1, &1, U32 & String, Socket>): import "./effs/tcp_accept.c" import "./effs/tcp_accept.js"

def TCP.connect(host: String, port: U32) -> IO(Result<&1, &1, U32 & String, Socket>): import "./effs/tcp_connect.c" import "./effs/tcp_connect.js"

def TCP.send(sock: Socket, data: String) -> IO(Socket & Result<&1, &1, U32 & String, Unit>): import "./effs/tcp_send.c" import "./effs/tcp_send.js"

def TCP.recv(sock: Socket, max: U32) -> IO(Socket & Result<&1, &1, U32 & String, String>): import "./effs/tcp_recv.c" import "./effs/tcp_recv.js"

def TCP.poll(sock: Socket, max: U32, ms: U32) -> IO(Socket & Result<&1, &1, U32 & String, Maybe<&1, String>>): import "./effs/tcp_poll.c" import "./effs/tcp_poll.js"

def UDP.bind(port: U32) -> IO(Result<&1, &1, U32 & String, Socket>): import "./effs/udp_bind.c" import "./effs/udp_bind.js"

def UDP.send_to(sock: Socket, host: String, port: U32, data: String) -> IO(Socket & Result<&1, &1, U32 & String, Unit>): import "./effs/udp_send_to.c" import "./effs/udp_send_to.js"

def UDP.recv_from(sock: Socket, max: U32) -> IO(Socket & Result<&1, &1, U32 & String, String & U32 & String>): import "./effs/udp_recv_from.c" import "./effs/udp_recv_from.js"

def UDP.poll(sock: Socket, max: U32) -> IO(Socket & Result<&1, &1, U32 & String, Maybe<&1, String & U32 & String>>): import "./effs/udp_poll.c" import "./effs/udp_poll.js"

def Socket.close(socket: Socket) -> IO(Unit): import "./effs/socket_close.c" import "./effs/socket_close.js"

def Listener.close(listener: Listener) -> IO(Unit): import "./effs/listener_close.c" import "./effs/listener_close.js"

def Window.open(title: String, width: U32, height: U32) -> IO(Result<&1, &1, U32 & String, Window>): import "./effs/window_open.c" import "./effs/window_open.js"

def Window.frame(window: Window, image: Image) -> IO(Window & Image & List<Event>): import "./effs/window_frame.c" import "./effs/window_frame.js"

def Window.set_title(window: Window, title: String) -> IO(Window): import "./effs/window_set_title.c" import "./effs/window_set_title.js"

def Window.close(window: Window) -> IO(Unit): import "./effs/window_close.c" import "./effs/window_close.js"
```

```
def Audio.open(rate: U32) -> IO(Result<&1, &1, U32 & String, Audio>): import "./effs/audio.c" import "./effs/audio_open.js"

def Audio.write(audio: Audio, samples: List<&2, F32>) -> IO(Audio & U32): import "./effs/audio.c" import "./effs/audio_write.js"

def Audio.close(audio: Audio) -> IO(Unit): import "./effs/audio.c" import "./effs/audio_close.js"
```

Equal

```
law Equal.cong: for -A: Type for -B: Type for -f: A -> B for -a: A for -b: A for e: {a == b : A} {f(a) == f(b) : B}

def Equal.cong(A, B, f, a, b, e): %e : {f(a) == f(_) : B} {==}

law Equal.sym: for -A: Type for -a: A for -b: A for e: {a == b : A} {b == a : A}

def Equal.sym(A, a, b, e): %e : {_ == a : A} {==}

law Equal.trans: for -A : Type for -a : A for -b : A for -c : A for ab : {a == b : A} for bc : {b == c : A} {a == c : A}

def Equal.trans(A, a, b, c, ab, bc): %bc : {a == _ : A} ab
```

Empty

```
def Empty.absurd(-A: Type, e: Empty) -> A: match e:
```

Bool

```
def Bool.not(b: Bool) -> Bool: match b: case False{}: True{} case True{}: False{}

def Bool.and(a: Bool, b: Bool) -> Bool: match a: case False{}: False{} case True{}: b

def Bool.or(a: Bool, b: Bool) -> Bool: match a: case False{}: b case True{}: True{}

def Bool.xor(a: Bool, b: Bool) -> Bool: match a: case False{}: b case True{}: Bool.not(b)

def Bool.cmp(a: Bool, b: Bool) -> Cmp: match a b: case False{} False{}: EQ{} case False{} True{}: LT{} case True{} False{}: GT{} case True{} True{}: EQ{}

def Bool.full_add(a: Bool, b: Bool, c: Bool) -> Bool & Bool: match a b c: case False{} False{} False{}: (False{}, False{}) case False{} False{} True{}: (True{}, False{}) case False{} True{} False{}: (True{}, False{}) case False{} True{} True{}: (False{}, True{}) case True{} False{} False{}: (True{}, False{}) case True{} False{} True{}: (False{}, True{}) case True{} True{} False{}: (False{}, True{}) case True{} True{} True{}: (True{}, True{})

def Bool.pick(-A: Type, c: Bool, a: A, b: A) -> A: match c: case False{}: b case True{}: a

def Bool.to_u32(b: Bool) -> U32: match b: case False{}: 0 case True{}: 1
```

Cmp

```
def Cmp.is_lt(c: Cmp) -> Bool: match c: case LT{}: True{} case EQ{}: False{} case GT{}: False{}

def Cmp.is_eq(c: Cmp) -> Bool: match c: case LT{}: False{} case EQ{}: True{} case GT{}: False{}

def Cmp.is_gt(c: Cmp) -> Bool: match c: case LT{}: False{} case EQ{}: False{} case GT{}: True{}

def Cmp.is_le(c: Cmp) -> Bool: match c: case LT{}: True{} case EQ{}: True{} case GT{}: False{}

def Cmp.is_ge(c: Cmp) -> Bool: match c: case LT{}: False{} case EQ{}: True{} case GT{}: True{}
```

Nat

```
def Nat.double(n: Nat) -> Nat: match n: case 0n: 0n case 1n+p: 2n+Nat.double(p)

def Nat.add(a: Nat, b: Nat) -> Nat: match a: case 0n: b case 1n+p: 1n+Nat.add(p, b)

def Nat.sub(a: Nat, b: Nat) -> Nat: match a b: case 0n 0n: 0n case 0n 1n+bp: 0n case 1n+ap 0n: 1n+ap case 1n+ap 1n+bp: Nat.sub(ap, bp)

def Nat.mul(a: Nat, +b: Nat) -> Nat: match a: case 0n: 0n case 1n+p: Nat.add(b, Nat.mul(p, b))

def Nat.divmod.go(n: Nat, m: Nat, d: Nat, r: Nat) -> Nat & Nat: match n: case 0n: (d, r) case 1n+np: match m: case 0n: Nat.divmod.go(np, r, 1n+d, 0n) case 1n+mp: Nat.divmod.go(np, mp, d, 1n+r)

def Nat.divmod(a: Nat, b: Nat) -> Nat & Nat: match b: case 0n: (0n, a) case 1n+bp: Nat.divmod.go(a, bp, 0n, 0n)

def Nat.cmp(a: Nat, b: Nat) -> Cmp: match a b: case 0n 0n: EQ{} case 0n 1n+bp: LT{} case 1n+ap 0n: GT{} case 1n+ap 1n+bp: Nat.cmp(ap, bp)

def Nat.is_lt(a: Nat, b: Nat) -> Bool: Cmp.is_lt(Nat.cmp(a, b))

def Nat.is_eq(a: Nat, b: Nat) -> Bool: Cmp.is_eq(Nat.cmp(a, b))

def Nat.is_ne(a: Nat, b: Nat) -> Bool: Bool.not(Cmp.is_eq(Nat.cmp(a, b)))

def Nat.is_le(a: Nat, b: Nat) -> Bool: Cmp.is_le(Nat.cmp(a, b))

def Nat.is_gt(a: Nat, b: Nat) -> Bool: Cmp.is_gt(Nat.cmp(a, b))

def Nat.is_ge(a: Nat, b: Nat) -> Bool: Cmp.is_ge(Nat.cmp(a, b))

def Nat.min(+a: Nat, +b: Nat) -> Nat: match a b: case 0n _: 0n case 1n+ap 0n: 0n case 1n+ap 1n+bp: 1n+Nat.min(ap, bp)

def Nat.max(+a: Nat, +b: Nat) -> Nat: match a b: case 0n _: b case 1n+ap 0n: 1n+ap case 1n+ap 1n+bp: 1n+Nat.max(ap, bp)

law Nat.ge_refl: for a: Nat {Nat.is_ge(a, a) == True{} : Bool}

def Nat.ge_refl(a): match a: case 0n: {==} case 1n+p: Nat.ge_refl(p)

law Nat.max_ge_l: for +a: Nat for +b: Nat {Nat.is_ge(Nat.max(a, b), a) == True{} : Bool}

def Nat.max_ge_l(a, b): match a b: case 0n 0n: {==} case 0n 1n+bp: {==} case 1n+ap 0n: Nat.ge_refl(a) case 1n+ap 1n+bp: Nat.max_ge_l(ap, bp)

law Nat.max_ge_r: for +a: Nat for +b: Nat {Nat.is_ge(Nat.max(a, b), b) == True{} : Bool}

def Nat.max_ge_r(a, b): match a b: case 0n _: Nat.ge_refl(b) case 1n+ap 0n: {==} case 1n+ap 1n+bp: Nat.max_ge_r(ap, bp)

def Nat.div.fin(qr: Nat & Nat) -> Nat: (q, r) = qr q

def Nat.div(a: Nat, b: Nat) -> Nat: Nat.div.fin(Nat.divmod(a, b))

def Nat.mod.fin(qr: Nat & Nat) -> Nat: (q, r) = qr r

def Nat.mod(a: Nat, b: Nat) -> Nat: Nat.mod.fin(Nat.divmod(a, b))

def Nat.pow(+a: Nat, b: Nat) -> Nat: match b: case 0n: 1n case 1n+p: Nat.mul(a, Nat.pow(a, p))
```

Maybe

```
def Maybe.pure(a, -A: Kind(a), x: A) -> Maybe<a, A>: Some{x}

def Maybe.bind( a, -A: Kind(a), -B: Kind(a), m: Maybe<a, A>, f: A -> Maybe<a, B> ) -> Maybe<a, B>: match m: case None{}: None{} case Some{x}: f(x)

def Maybe.default(a, -A: Kind(a), m: Maybe<a, A>, d: A) -> A: match m: case None{}: d case Some{x}: x

def Maybe.is_some(a, -A: Kind(a), m: Maybe<a, A>) -> Bool: match m: case None{}: False{} case Some{x}: True{}
```

```
def Maybe.is_none(a, -A: Kind(a), m: Maybe<a, A>) -> Bool: Bool.not(Maybe.is_some(a, A, m))

def Maybe.map( a, -A: Kind(a), -B: Kind(a), f: A -> B, m: Maybe<a, A> ) -> Maybe<a, B>: match m: case None{}: None{} case Some{x}: Some{f(x)}

def Maybe.or(a, -A: Kind(a), m: Maybe<a, A>, n: Maybe<a, A>) -> Maybe<a, A>: match m: case None{}: n case Some{x}: Some{x}
```

Result

```
------

def Result.pure(a, b, -E: Kind(a), -A: Kind(b), x: A) -> Result<a, b, E, A>: Done{x}

def Result.bind( a, b, -E: Kind(a), -A: Kind(b), -B: Kind(b), r: Result<a, b, E, A>, f: A -> Result<a, b, E, B> ) -> Result<a, b, E, B>: match r: case Fail{e}: Fail{e} case Done{x}: f(x)

def Result.default( a, b, -E: Kind(a), -A: Kind(b), r: Result<a, b, E, A>, d: A ) -> A: match r: case Fail{e}: d case Done{x}: x

def Result.is_done( a, b, -E: Kind(a), -A: Kind(b), r: Result<a, b, E, A> ) -> Bool: match r: case Fail{e}: False{} case Done{x}: True{}

def Result.is_fail( a, b, -E: Kind(a), -A: Kind(b), r: Result<a, b, E, A> ) -> Bool: Bool.not(Result.is_done(a, b, E, A, r))

def Result.map( a, b, -E: Kind(a), -A: Kind(b), -B: Kind(b), f: A -> B, r: Result<a, b, E, A> ) -> Result<a, b, E, B>: match r: case Fail{e}: Fail{e} case Done{x}: Done{f(x)}
```

List

```
----

def List.map(~A: Type, ~B: Type, ~f: A -> B, xs: List) -> List<B>: match xs: case Nil{}: Nil{} case h <> t: f(h) <> List.map(~A, ~B, ~f, t)

def List.length(a, -A: Kind(a), xs: List<a, A>) -> Nat: match xs: case Nil{}: 0n case h <> t: 1n+List.length(a, A, t)

def List.append(a, -A: Kind(a), xs: List<a, A>, ys: List<a, A>) -> List<a, A>: match xs: case Nil{}: ys case h <> t: h <> List.append(a, A, t, ys)

def List.concat(a, -A: Kind(a), xss: List<a, List<a, A>>) -> List<a, A>: match xss: case Nil{}: Nil{} case h <> t: List.append(a, A, h, List.concat(a, A, t))

def List.reverse.go(a, -A: Kind(a), xs: List<a, A>, acc: List<a, A>) -> List<a, A>: match xs: case Nil{}: acc case h <> t: List.reverse.go(a, A, t, h <> acc)

def List.reverse(a, -A: Kind(a), xs: List<a, A>) -> List<a, A>: List.reverse.go(a, A, xs, Nil{})

def List.is_empty(a, -A: Kind(a), xs: List<a, A>) -> Bool: match xs: case Nil{}: True{} case h <> t: False{}

def List.head(a, -A: Kind(a), xs: List<a, A>) -> Maybe<a, A>: match xs: case Nil{}: None{} case h <> t: Some{h}

def List.tail(a, -A: Kind(a), xs: List<a, A>) -> List<a, A>: match xs: case Nil{}: Nil{} case h <> t: t

def List.last.go(a, -A: Kind(a), xs: List<a, A>, last: A) -> A: match xs: case Nil{}: last case h <> t: List.last.go(a, A, t, h)

def List.last(a, -A: Kind(a), xs: List<a, A>) -> Maybe<a, A>: match xs: case Nil{}: None{} case h <> t: Some{List.last.go(a, A, t, h)}

def List.get(a, -A: Kind(a), xs: List<a, A>, n: Nat) -> Maybe<a, A>: match xs n: case Nil{} _: None{} case h <> t 0n: Some{h} case h <> t 1n+p: List.get(a, A, t, p)

def List.set(a, -A: Kind(a), xs: List<a, A>, n: Nat, x: A) -> List<a, A>: match xs n: case Nil{} _: Nil{} case h <> t 0n: x <> t case h <> t 1n+p: h <> List.set(a, A, t, p, x)

def List.take(a, -A: Kind(a), xs: List<a, A>, n: Nat) -> List<a, A>: match xs n: case Nil{} _: Nil{} case h <> t 0n: Nil{} case h <> t 1n+p: h <> List.take(a, A, t, p)

def List.drop(a, -A: Kind(a), xs: List<a, A>, n: Nat) -> List<a, A>: match xs n: case Nil{} _: Nil{} case h <> t 0n: h <> t case h <> t 1n+p: List.drop(a, A, t, p)
```

```
def List.zip( a, -A: Kind(a), b, -B: Kind(b), xs: List<a, A>, ys: List<b, B> ) -> List<&1, A & B>: match xs ys: case Nil{} _: Nil{} case h <> t Nil{}: Nil{} case x <> xt y <> yt: (x, y) <> List.zip(a, A, b, B, xt, yt)

def List.range.go(+n: Nat, acc: List<&2, Nat>) -> List<&2, Nat>: match n: case 0n: acc case 1n+p: List.range.go(p, p <> acc)

def List.range(n: Nat) -> List<&2, Nat>: List.range.go(n, Nil{})

def List.replicate(-A: Data, n: Nat, +x: A) -> List<&2, A>: match n: case 0n: Nil{} case 1n+p: x <> List.replicate(A, p, x)

def List.filter.put(-A: Data, h: A, r: List<&2, A>, keep: Bool) -> List<&2, A>: match keep: case False{}: r case True{}: h <> r

def List.filter(~A: Data, ~f: A -> Bool, xs: List<&2, A>) -> List<&2, A>: match xs: case Nil{}: Nil{} case +h <> t: List.filter.put(A, h, List.filter(~A, ~f, t), f(h))

def List.foldl( ~a: Quant, ~A: Kind(a), ~B: Type, ~f: B -> A -> B, xs: List<a, A>, acc: B ) -> B: match xs: case Nil{}: acc case h <> t: List.foldl(~a, ~A, ~B, ~f, t, f(acc, h))

def List.foldr( ~a: Quant, ~A: Kind(a), ~B: Type, ~f: A -> B -> B, xs: List<a, A>, z: B ) -> B: match xs: case Nil{}: z case h <> t: f(h, List.foldr(~a, ~A, ~B, ~f, t, z))

def List.any(~a: Quant, ~A: Kind(a), ~f: A -> Bool, xs: List<a, A>) -> Bool: match xs: case Nil{}: False{} case h <> t: Bool.or(f(h), List.any(~a, ~A, ~f, t))

def List.all(~a: Quant, ~A: Kind(a), ~f: A -> Bool, xs: List<a, A>) -> Bool: match xs: case Nil{}: True{} case h <> t: Bool.and(f(h), List.all(~a, ~A, ~f, t))

def List.find.put(-A: Data, h: A, r: Maybe<&2, A>, hit: Bool) -> Maybe<&2, A>: match hit: case False{}: r case True{}: Some{h}

def List.find(~A: Data, ~f: A -> Bool, xs: List<&2, A>) -> Maybe<&2, A>: match xs: case Nil{}: None{} case +h <> t: List.find.put(A, h, List.find(~A, ~f, t), f(h))

def List.contains( ~A: Data, ~eq: A -> A -> Bool, xs: List<&2, A>, +x: A ) -> Bool: match xs: case Nil{}: False{} case h <> t: Bool.or(eq(h, x), List.contains(~A, ~eq, t, x))

def List.merge.step( -A: Data, acc: List<&2, A>, x: A, xt: List<&2, A>, y: A, yt: List<&2, A>, le: Bool ) -> List<&2, A> & List<&2, A> & List<&2, A>: match le: case False{}: (y <> acc, x <> xt, yt) case True{}: (x <> acc, xt, y <> yt)
```

The fuel bounds the steps: a sort passes its length, enough for any merge.

```
def List.merge.go( ~A: Data, ~le: A -> A -> Bool, fuel: Nat, st: List<&2, A> & List<&2, A> & List<&2, A> ) -> List<&2, A>: match fuel: case 0n: (acc, xs, ys) = st List.reverse.go(&2, A, acc, List.append(&2, A, xs, ys)) case 1n+f: (acc, xs, ys) = st match xs ys: case Nil{} Nil{}: List.reverse.go(&2, A, acc, Nil{}) case Nil{} y <> yt: List.reverse.go(&2, A, acc, y <> yt) case x <> xt Nil{}: List.reverse.go(&2, A, acc, x <> xt) case +x <> xt +y <> yt: List.merge.go(~A, ~le, f, List.merge.step(A, acc, x, xt, y, yt, le(x, y)))

def List.sort.runs(-A: Data, xs: List<&2, A>) -> List<&2, List<&2, A>>: match xs: case Nil{}: Nil{} case h <> t: [h] <> List.sort.runs(A, t)

def List.sort.pass( ~A: Data, ~le: A -> A -> Bool, +n: Nat, runs: List<&2, List<&2, A>> ) -> List<&2, List<&2, A>>: match runs: case Nil{}: Nil{} case r <> Nil{}: [r] case r1 <> r2 <> rest: List.merge.go(~A, ~le, n, (Nil{}, r1, r2)) <> List.sort.pass(~A, ~le, n, rest)

def List.sort.go( ~A: Data, ~le: A -> A -> Bool, fuel: Nat, +n: Nat, runs: List<&2, List<&2, A>> ) -> List<&2, A>: match fuel: case 0n: List.concat(&2, A, runs) case 1n+f: match runs: case Nil{}: Nil{} case r <> Nil{}: r case r1 <> r2 <> rest: List.sort.go(~A, ~le, f, n, List.sort.pass(~A, ~le, n, r1 <> r2 <> rest))

def List.sort(~A: Data, ~le: A -> A -> Bool, +xs: List<&2, A>) -> List<&2, A>: +n = List.length(&2, A, xs) List.sort.go(~A, ~le, n, n, List.sort.runs(A, xs))

def List.for_each( ~a: Quant, ~A: Kind(a), ~f: A -> IO(Unit), xs: List<a, A> ) -> IO(Unit): match xs: case Nil{}: IO.pure(Unit, Unit{}) case h <> t: IO.bind(Unit, Unit, f(h), u => List.for_each(~a, ~A, ~f, t))
```

Word

```
def Word.zero(n: Nat) -> Word(n): match n: case 0n: WNil{} case 1n+p: WCon{False{}, Word.zero(p)}
```

```
def Word.not(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: WCon{Bool.not(b), Word.not(p, t)}

def Word.and(n: Nat, a: Word(n), b: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: WCon{Bool.and(ab, bb), Word.and(p, at, bt)}

def Word.or(n: Nat, a: Word(n), b: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: WCon{Bool.or(ab, bb), Word.or(p, at, bt)}

def Word.xor(n: Nat, a: Word(n), b: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: WCon{Bool.xor(ab, bb), Word.xor(p, at, bt)}

def Word.shl.put(n: Nat, c: Bool, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: WCon{c, Word.shl.put(p, b, t)}

def Word.shl(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: WCon{False{}, Word.shl.put(p, b, t)}

def Word.shl.out.con(-p: Nat, c: Bool, r: Bool & Word(p)) -> Bool & Word(1n+p): (hi, t2) = r (hi, WCon{c, t2})

def Word.shl.out(n: Nat, c: Bool, w: Word(n)) -> Bool & Word(n): match n: case 0n: (c, WNil{}) case 1n+p: match w: case WCon{b, t}: Word.shl.out.con(p, c, Word.shl.out(p, b, t))

def Word.shr.pad(n: Nat, w: Word(n)) -> Word(1n+n): match n: case 0n: WCon{False{}, w} case 1n+p: match w: case WCon{b, t}: WCon{b, Word.shr.pad(p, t)}

def Word.shr(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{b, t}: Word.shr.pad(p, t)

def Word.cmp.fin(ab: Bool, bb: Bool, t: Cmp) -> Cmp: match t: case LT{}: LT{} case EQ{}: Bool.cmp(ab, bb) case GT{}: GT{}

def Word.cmp(n: Nat, a: Word(n), b: Word(n)) -> Cmp: match n: case 0n: EQ{} case 1n+p: match a b: case WCon{ab, at} WCon{bb, bt}: Word.cmp.fin(ab, bb, Word.cmp(p, at, bt))

def Word.inc(n: Nat, w: Word(n)) -> Word(n): match n: case 0n: WNil{} case 1n+p: match w: case WCon{False{}, t}: WCon{True{}, t} case WCon{True{}, t}: WCon{False{}, Word.inc(p, t)}

law Word.adc: for n: Nat for a: Word(n) for b: Word(n) for f: Bool for c: Bool Word(n)

def Word.adc.con(p: Nat, at: Word(p), bt: Word(p), f: Bool, sk: Bool & Bool) -> Word(1n+p): (s, k) = sk WCon{s, Word.adc(p, at, bt, f, k)}

def Word.adc(n, a, b, f, c): match n: case 0n: WNil{} case 1n+p: match a b f: case WCon{ab, at} WCon{bb, bt} False{}: Word.adc.con(p, at, bt, False{}, Bool.full_add(ab, bb, c)) case WCon{ab, at} WCon{bb, bt} True{}: Word.adc.con(p, at, bt, True{}, Bool.full_add(ab, Bool.not(bb), c))

def Word.add(n: Nat, a: Word(n), b: Word(n)) -> Word(n): Word.adc(n, a, b, False{}, False{})

def Word.sub(n: Nat, a: Word(n), b: Word(n)) -> Word(n): Word.adc(n, a, b, True{}, True{})

law Word.add_comm.arm: for p : Nat for -h : Bool for at : Word(p) for bt : Word(p) for k : Bool {WCon{h, Word.adc(p, at, bt, False{}, k)} == WCon{h, Word.adc(p, bt, at, False{}, k)} : Word.Con

}

law Word.add_comm.go: for n: Nat for a: Word(n) for b: Word(n) for c: Bool {Word.adc(n, a, b, False{}, c) == Word.adc(n, b, a, False{}, c) : Word(n)}

def Word.add_comm.arm(p, h, at, bt, k): Equal.cong(Word(p), Word(1n+p), w => WCon{h, w}, Word.adc(p, at, bt, False{}, k), Word.adc(p, bt, at, False{}, k), Word.add_comm.go(p, at, bt, k))

def Word.add_comm.go(n, a, b, c): match n: case 0n: match a b: case WNil{} WNil{}: {==} case 1n+p: match a b c: case WCon{False{}, at} WCon{False{}, bt} False{}: Word.add_comm.arm(p, False{}, at, bt, False{}) case WCon{False{}, at} WCon{False{}, bt} True{}: Word.add_comm.arm(p, True{}, at, bt, False{}) case WCon{False{}, at} WCon{True{}, bt} False{}: Word.add_comm.arm(p, True{}, at, bt, False{}) case WCon{False{}, at} WCon{True{}, bt} True{}: Word.add_comm.arm(p, False{}, at, bt, True{}) case WCon{True{}, at} WCon{False{}, bt} False{}: Word.add_comm.arm(p, True{}, at, bt, False{}) case WCon{True{}, at} WCon{False{}, bt} True{}: Word.add_comm.arm(p, False{}, at, bt, True{}) case WCon{True{}, at} WCon{True{}, bt} False{}: Word.add_comm.arm(p, False{}, at, bt, True{}) case WCon{True{}, at} WCon{True{}, bt} True{}: Word.add_comm.arm(p, True{}, at, bt, True{})

law Word.add_comm: for n: Nat for a: Word(n) for b: Word(n) {Word.add(n, a, b) == Word.add(n, b, a) : Word(n)}
```

```
def Word.add_comm(n, a, b): Word.add_comm.go(n, a, b, False{})

def Word.to_nat(n: Nat, w: Word(n)) -> Nat: match n: case 0n: 0n case 1n+p: match w: case WCon{False{}, t}: Nat.double(Word.to_nat(p, t)) case WCon{True{}, t}: 1n+Nat.double(Word.to_nat(p, t))

def Word.mul.go(+n: Nat, m: Nat, a: Word(m), +b: Word(n), acc: Word(n)) -> Word(n): match m: case 0n: acc case 1n+mp: match a: case WCon{False{}, at}: Word.mul.go(n, mp, at, Word.shl(n, b), acc) case WCon{True{}, at}: Word.mul.go(n, mp, at, Word.shl(n, b), Word.add(n, acc, b))

def Word.mul(+n: Nat, a: Word(n), b: Word(n)) -> Word(n): Word.mul.go(n, n, a, b, Word.zero(n))
```

U32

```
def U32.inc(a: U32) -> U32: match a: case U32{x}: U32{Word.inc(32n, x)}

def U32.add(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.add(32n, x, y)}

law U32.add_comm: for a: U32 for b: U32 {U32.add(a, b) == U32.add(b, a) : U32}

def U32.add_comm(a, b): match a b: case U32{x} U32{y}: Equal.cong(Word(32n), U32, w => U32{w}, Word.add(32n, x, y), Word.add(32n, y, x), Word.add_comm(32n, x, y))

def U32.sub(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.sub(32n, x, y)}

def U32.mul(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.mul(32n, x, y)}

def U32.not(a: U32) -> U32: match a: case U32{x}: U32{Word.not(32n, x)}

def U32.and(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.and(32n, x, y)}

def U32.or(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.or(32n, x, y)}

def U32.xor(a: U32, b: U32) -> U32: match a b: case U32{x} U32{y}: U32{Word.xor(32n, x, y)}

def U32.shl(a: U32) -> U32: match a: case U32{x}: U32{Word.shl(32n, x)}

def U32.shr(a: U32) -> U32: match a: case U32{x}: U32{Word.shr(32n, x)}

def U32.shln(a: U32, n: Nat) -> U32: match n: case 0n: a case 1n+p: U32.shl(U32.shln(a, p))

def U32.shrn(a: U32, n: Nat) -> U32: match n: case 0n: a case 1n+p: U32.shr(U32.shrn(a, p))

def U32.cmp(a: U32, b: U32) -> Cmp: match a b: case U32{x} U32{y}: Word.cmp(32n, x, y)

def U32.is_eq(a: U32, b: U32) -> Bool: Cmp.is_eq(U32.cmp(a, b))

def U32.is_ne(a: U32, b: U32) -> Bool: Bool.not(Cmp.is_eq(U32.cmp(a, b)))

def U32.is_lt(a: U32, b: U32) -> Bool: Cmp.is_lt(U32.cmp(a, b))

def U32.is_le(a: U32, b: U32) -> Bool: Cmp.is_le(U32.cmp(a, b))

def U32.is_gt(a: U32, b: U32) -> Bool: Cmp.is_gt(U32.cmp(a, b))

def U32.is_ge(a: U32, b: U32) -> Bool: Cmp.is_ge(U32.cmp(a, b))

def U32.is_zero(a: U32) -> Bool: U32.is_eq(a, 0)

def U32.to_nat(a: U32) -> Nat: match a: case U32{w}: Word.to_nat(32n, w)

def U32.from_nat(n: Nat) -> U32: match n: case 0n: 0 case 1n+p: U32.inc(U32.from_nat(p))

def U32.log2.go(+k: Nat, +n: U32, +d: Nat) -> Nat: match k: case 0n: d case 1n+p: U32.log2.go(p, U32.shr(n), Nat.add(d, U32.to_nat(Bool.to_u32(U32.is_lt(1, n)))))

def U32.log2(+n: U32) -> Nat: U32.log2.go(32n, n, 0n)

def U32.divmod.go.fin(-p: Nat, q: Word(p), s: U32, b: U32, g: Bool) -> Word(1n+p) & U32: match g: case True{}: (WCon{True{}, q}, U32.sub(s, b)) case False{}: (WCon{False{}, q}, s)

def U32.divmod.go.shl(-p: Nat, q: Word(p), +b: U32, ts: Bool & Word(32n)) -> Word(1n+p) & U32: (t, s) = ts +s2 = {U32{s} : U32} U32.divmod.go.fin(p, q, s2, b, Bool.or(t, U32.is_ge(s2, b)))
```

```
def U32.divmod.go.rec(-p: Nat, a0: Bool, +b: U32, qr: Word(p) & U32) -> Word(1n+p) & U32: (q, r) = qr U32{rw} = r U32.divmod.go.shl(p, q, b, Word.shl.out(32n, a0, rw))

def U32.divmod.go(m: Nat, a: Word(m), +b: U32) -> Word(m) & U32: match m a: case 0n WNil{}: (WNil{}, 0) case 1n+p WCon{a0, hi}: U32.divmod.go.rec(p, a0, b, U32.divmod.go(p, hi, b))

def U32.div.fin(qr: Word(32n) & U32) -> U32: (q, r) = qr U32{q}

def U32.div.if(aw: Word(32n), b: U32, z: Bool) -> U32: match z: case True{}: 0 case False{}: U32.div.fin(U32.divmod.go(32n, aw, b))

def U32.div(a: U32, +b: U32) -> U32: U32{aw} = a U32.div.if(aw, b, U32.is_zero(b))

def U32.mod.fin(qr: Word(32n) & U32) -> U32: (q, r) = qr r

def U32.mod.if(aw: Word(32n), b: U32, z: Bool) -> U32: match z: case True{}: U32{aw} case False{}: U32.mod.fin(U32.divmod.go(32n, aw, b))

def U32.mod(a: U32, +b: U32) -> U32: U32{aw} = a U32.mod.if(aw, b, U32.is_zero(b))

def U32.min(+a: U32, +b: U32) -> U32: Bool.pick(U32, U32.is_lt(a, b), a, b)

def U32.max(+a: U32, +b: U32) -> U32: Bool.pick(U32, U32.is_lt(a, b), b, a)

def U32.clamp(x: U32, lo: U32, hi: U32) -> U32: U32.min(U32.max(x, lo), hi)

def U32.pow(+a: U32, n: Nat) -> U32: match n: case 0n: 1 case 1n+p: U32.mul(a, U32.pow(a, p))

def U32.is_even(a: U32) -> Bool: U32.is_zero(U32.and(a, 1))
```

F32

```
law U32.to_f32: for a: U32 F32

law F32.to_u32: for a: F32 U32

law F32.add: for a: F32 for b: F32 F32

law F32.sub: for a: F32 for b: F32 F32

law F32.mul: for a: F32 for b: F32 F32

law F32.div: for a: F32 for b: F32 F32

law F32.mod: for a: F32 for b: F32 F32

law F32.pow: for a: F32 for b: F32 F32

law F32.atan2: for a: F32 for b: F32 F32

law F32.neg: for a: F32 F32

law F32.abs: for a: F32 F32

law F32.sqrt: for a: F32 F32

law F32.exp: for a: F32 F32

law F32.log: for a: F32 F32

law F32.log2: for a: F32 F32

law F32.log10: for a: F32 F32

law F32.sin: for a: F32 F32

law F32.cos: for a: F32 F32

law F32.tan: for a: F32 F32

law F32.asin: for a: F32 F32

law F32.acos: for a: F32 F32
```

```
law F32.atan: for a: F32 F32
law F32.sinh: for a: F32 F32
law F32.cosh: for a: F32 F32
law F32.tanh: for a: F32 F32
law F32.floor: for a: F32 F32
law F32.ceil: for a: F32 F32
law F32.trunc: for a: F32 F32
law F32.is_eq: for a: F32 for b: F32 Bool
law F32.is_ne: for a: F32 for b: F32 Bool
law F32.is_lt: for a: F32 for b: F32 Bool
law F32.is_le: for a: F32 for b: F32 Bool
law F32.is_gt: for a: F32 for b: F32 Bool
law F32.is_ge: for a: F32 for b: F32 Bool
law F32.show: for +a: F32 String
law F32.bits: for a: F32 U32
law F32.read: for s: String Maybe<&2, F32>
def F32.min(+a: F32, +b: F32) -> F32: Bool.pick(F32, F32.is_lt(a, b), a, b)
def F32.max(+a: F32, +b: F32) -> F32: Bool.pick(F32, F32.is_lt(a, b), b, a)
def F32.clamp(x: F32, lo: F32, hi: F32) -> F32: F32.min(F32.max(x, lo), hi)
def F32.lerp(+a: F32, b: F32, t: F32) -> F32: F32.add(a, F32.mul(F32.sub(b, a), t))
def F32.square(+a: F32) -> F32: F32.mul(a, a)
def F32.hypot(+x: F32, +y: F32) -> F32: F32.sqrt(F32.add(F32.mul(x, x), F32.mul(y, y)))
def F32.round(a: F32) -> F32: F32.floor(F32.add(a, 0.5))
def F32.pi() -> F32: 3.14159265
def F32.from_nat(n: Nat) -> F32: U32.to_f32(U32.from_nat(n))
def F32.to_nat(a: F32) -> Nat: U32.to_nat(F32.to_u32(a))
```

Char

```
----
def Char.cmp(a: Char, b: Char) -> (Char & Char) & Cmp: match a b: case Chr{+x} Chr{+y}: ((Chr{x}, Chr{y}), U32.cmp(x, y))
def Char.to_u32(c: Char) -> U32: match c: case Chr{x}: x
def Char.from_u32(x: U32) -> Char: Chr{x}
def Char.is_eq(a: Char, b: Char) -> Bool: match a b: case Chr{x} Chr{y}: U32.is_eq(x, y)
def Char.is_digit(c: Char) -> Bool: match c: case Chr{+x}: Bool.and(U32.is_ge(x, 48), U32.is_le(x, 57))
def Char.is_upper(c: Char) -> Bool: match c: case Chr{+x}: Bool.and(U32.is_ge(x, 65), U32.is_le(x, 90))
def Char.is_lower(c: Char) -> Bool: match c: case Chr{+x}: Bool.and(U32.is_ge(x, 97), U32.is_le(x, 122))
def Char.is_alpha(+c: Char) -> Bool: Bool.or(Char.is_upper(c), Char.is_lower(c))
def Char.is_space(c: Char) -> Bool: match c: case Chr{+x}: Bool.or(U32.is_eq(x, 32), Bool.and(U32.is_ge(x, 9), U32.is_le(x, 13)))
def Char.to_upper(+c: Char) -> Char: Chr{U32.sub(Char.to_u32(c), U32.mul(Bool.to_u32(Char.is_lower(c)), 32))}
```

```
def Char.to_lower(+c: Char) -> Char: Chr{U32.add(Char.to_u32(c), U32.mul(Bool.to_u32(Char.is_upper(c)), 32))}
```

String

```
def String.append(a: String, b: String) -> String: match a: case SNil{}: b case SCon{h, t}: SCon{h, String.append(t, b)}

def String.cmp.rec(h1b: Char, h2b: Char, rr: (String & String) & Cmp) -> (String & String) & Cmp: ((t1b, t2b), r) = rr ((SCon{h1b, t1b}, SCon{h2b, t2b}), r)

law String.cmp: for a: String for b: String (String & String) & Cmp

def String.cmp.fin(t1: String, t2: String, hc: (Char & Char) & Cmp) -> (String & String) & Cmp: ((h1b, h2b), c) = hc match c: case LT{}: ((SCon{h1b, t1}, SCon{h2b, t2}), LT{}) case EQ{}: String.cmp.rec(h1b, h2b, String.cmp(t1, t2)) case GT{}: ((SCon{h1b, t1}, SCon{h2b, t2}), GT{})

def String.cmp(a, b): match a b: case SNil{} SNil{}: ((SNil{}, SNil{}), EQ{}) case SNil{} SCon{h, t}: ((SNil{}, SCon{h, t}), LT{}) case SCon{h, t} SNil{}: ((SCon{h, t}, SNil{}), GT{}) case SCon{h1, t1} SCon{h2, t2}: String.cmp.fin(t1, t2, Char.cmp(h1, h2))

def String.eq.fin(r: (String & String) & Cmp) -> Bool: ((a2, b2), c) = r Cmp.is_eq(c)

def String.eq(a: String, b: String) -> Bool: String.eq.fin(String.cmp(a, b))

def String.length(s: String) -> Nat: match s: case SNil{}: 0n case SCon{h, t}: 1n+String.length(t)

def String.is_empty(s: String) -> Bool: match s: case SNil{}: True{} case SCon{h, t}: False{}

def String.reverse.go(s: String, acc: String) -> String: match s: case SNil{}: acc case SCon{h, t}: String.reverse.go(t, SCon{h, acc})

def String.reverse(s: String) -> String: String.reverse.go(s, SNil{})

def String.order.fin(r: (String & String) & Cmp) -> Cmp: (ab, c) = r c

def String.order(a: String, b: String) -> Cmp: String.order.fin(String.cmp(a, b))

def String.is_lt(a: String, b: String) -> Bool: Cmp.is_lt(String.order(a, b))

def String.is_le(a: String, b: String) -> Bool: Cmp.is_le(String.order(a, b))

def String.is_gt(a: String, b: String) -> Bool: Cmp.is_gt(String.order(a, b))

def String.is_ge(a: String, b: String) -> Bool: Cmp.is_ge(String.order(a, b))

law String.starts_with: for s: String for p: String Bool

def String.starts_with.if(t: String, pt: String, same: Bool) -> Bool: match same: case False{}: False{} case True{}: String.starts_with(t, pt)

def String.starts_with(s, p): match s p: case SNil{} SNil{}: True{} case SNil{} SCon{h, t}: False{} case SCon{h, t} SNil{}: True{} case SCon{x, xt} SCon{y, yt}: String.starts_with.if(xt, yt, Char.is_eq(x, y))

def String.ends_with(s: String, p: String) -> Bool: String.starts_with(String.reverse(s), String.reverse(p))

law String.contains: for +s: String for +p: String Bool

def String.contains.if(t: String, p: String, here: Bool) -> Bool: match here: case False{}: String.contains(t, p) case True{}: True{}

def String.contains(s, p): match s: case SNil{}: String.is_empty(p) case SCon{h, t}: String.contains.if(t, p, String.starts_with(SCon{h, t}, p))

def String.take(s: String, n: Nat) -> String: match s n: case SNil{} _: SNil{} case SCon{h, t} 0n: SNil{} case SCon{h, t} 1n+p: SCon{h, String.take(t, p)}

def String.drop(s: String, n: Nat) -> String: match s n: case SNil{} _: SNil{} case SCon{h, t} 0n: SCon{h, t} case SCon{h, t} 1n+p: String.drop(t, p)

def String.get(s: String, n: Nat) -> Maybe<&2, Char>: match s n: case SNil{} _: None{} case SCon{h, t} 0n: Some{h} case SCon{h, t} 1n+p: String.get(t, p)

def String.to_list(s: String) -> List<&2, Char>: match s: case SNil{}: Nil{} case SCon{h, t}: h <> String.to_list(t)
```

```
def String.from_list(cs: List<&2, Char>) -> String: match cs: case Nil{}: SNil{} case h <> t: SCon{h, String.from_list(t)}

def String.concat(xs: List<&2, String>) -> String: match xs: case Nil{}: SNil{} case h <> t: String.append(h, String.concat(t))

def String.join.go(xs: List<&2, String>, h: String, +sep: String) -> String: match xs: case Nil{}: h case h2 <> t: String.append(h, String.append(sep, String.join.go(t, h2, sep)))

def String.join(xs: List<&2, String>, +sep: String) -> String: match xs: case Nil{}: SNil{} case h <> t: String.join.go(t, h, sep)

def String.split.push(c: Char, ps: List<&2, String>) -> List<&2, String>: match ps: case Nil{}: [SCon{c, SNil{}}] case h <> t: SCon{c, h} <> t

def String.split.fin(c: Char, r: List<&2, String>, cut: Bool) -> List<&2, String>: match cut: case False{}: String.split.push(c, r) case True{}: SNil{} <> r

def String.split(s: String, +sep: Char) -> List<&2, String>: match s: case SNil{}: [SNil{}] case SCon{+h, t}: String.split.fin(h, String.split(t, sep), Char.is_eq(h, sep))

def String.lines(s: String) -> List<&2, String>: String.split(s, '\n')

def String.repeat(+s: String, n: Nat) -> String: match n: case 0n: SNil{} case 1n+p: String.append(s, String.repeat(s, p))

def String.to_upper(s: String) -> String: match s: case SNil{}: SNil{} case SCon{h, t}: SCon{Char.to_upper(h), String.to_upper(t)}

def String.to_lower(s: String) -> String: match s: case SNil{}: SNil{} case SCon{h, t}: SCon{Char.to_lower(h), String.to_lower(t)}

law String.trim_start: for s: String String

def String.trim_start.if(h: Char, t: String, space: Bool) -> String: match space: case False{}: SCon{h, t} case True{}: String.trim_start(t)

def String.trim_start(s): match s: case SNil{}: SNil{} case SCon{+h, t}: String.trim_start.if(h, t, Char.is_space(h))

def String.trim_end(s: String) -> String: String.reverse(String.trim_start(String.reverse(s)))

def String.trim(s: String) -> String: String.trim_end(String.trim_start(s))
```

Text

```
def Nat.show.put(qr: Nat & Nat) -> Char & Nat: (q, r) = qr (Chr{U32.from_nat(Nat.add(48n, r))}, q)

law Nat.show.go: for f : Nat for n : Nat for acc : String String

def Nat.show.fin(g: Nat, acc: String, dq: Char & Nat) -> String: (d, q) = dq match q: case 0n: SCon{d, acc} case 1n+p: Nat.show.go(g, 1n+p, SCon{d, acc})

def Nat.show.go(f, n, acc): match f: case 0n: acc case 1n+g: Nat.show.fin(g, acc, Nat.show.put(Nat.divmod(n, 10n)))

def Nat.show(n: Nat) -> String: +m = n Nat.show.fin(m, SNil{}, Nat.show.put(Nat.divmod(m, 10n)))

law Nat.read.go: for s : String for +acc : Nat Maybe<&2, Nat>

def Nat.read.max() -> Nat: Nat.mul(U32.to_nat(16777215), U32.to_nat(16777217))

def Nat.read.fit(acc: Nat, qr: Nat & Nat) -> Bool: (q, r) = qr Bool.not(Nat.is_lt(q, acc))

def Nat.read.if(t: String, acc: Nat, d: U32, ok: Bool) -> Maybe<&2, Nat>: match ok: case True{}: Nat.read.go(t, Nat.add(Nat.mul(acc, 10n), U32.to_nat(d))) case False{}: None{}

def Nat.read.go(s, acc): match s: case SNil{}: Some{acc} case SCon{Chr{x}, t}: +d = U32.sub(x, 48) Nat.read.if(t, acc, d, Bool.and(U32.is_lt(d, 10), Nat.read.fit(acc, Nat.divmod(Nat.sub(Nat.read.max(), U32.to_nat(d)), 10n))))

def Nat.read(s: String) -> Maybe<&2, Nat>: match s: case SNil{}: None{} case SCon{h, t}: Nat.read.go(SCon{h, t}, 0n)

law U32.show.go: for f : Nat for +n : U32 for acc : String String

def U32.show.fin(g: Nat, acc: String, +n: U32, z: Bool) -> String: match z: case True{}: acc case False{}: U32.show.go(g, U32.div(n, 10), SCon{Chr{U32.add(48, U32.mod(n, 10))}, acc})
```

```
def U32.show.go(f, n, acc): match f: case 0n: acc case 1n+g: U32.show.fin(g, acc, n, U32.is_zero(n))

def U32.show.if(a: U32, z: Bool) -> String: match z: case True{}: SCon{Chr{48}, SNil{}} case False{}: U32.show.go(10n, a, SNil{})

def U32.show(a: U32) -> String: +b = a U32.show.if(b, U32.is_zero(b))

law U32.read.go: for s : String for +acc : U32 Maybe<&2, U32>

def U32.read.if(t: String, n: U32, ok: Bool) -> Maybe<&2, U32>: match ok: case True{}: U32.read.go(t, n) case False{}: None{}

def U32.read.go(s, acc): match s: case SNil{}: Some{acc} case SCon{Chr{x}, t}: +n = U32.add(U32.mul(acc, 10), U32.sub(x, 48)) U32.read.if(t, n, U32.is_eq(U32.div(n, 10), acc))

def U32.read(s: String) -> Maybe<&2, U32>: match s: case SNil{}: None{} case SCon{h, t}: U32.read.go(SCon{h, t}, 0)

def Bool.show(b: Bool) -> String: match b: case False{}: "False" case True{}: "True"

def Char.show(c: Char) -> String: SCon{c, SNil{}}

def Maybe.show(~a: Quant, ~A: Kind(a), ~f: A -> String, m: Maybe<a, A>) -> String: match m: case None{}: "None" case Some{x}: "Some(" ++ f(x) ++ ")"

def List.show.go(~a: Quant, ~A: Kind(a), ~f: A -> String, xs: List<a, A>) -> String: match xs: case Nil{}: "]" case h <> t: ", " ++ f(h) ++ List.show.go(~a, ~A, ~f, t)

def List.show(~a: Quant, ~A: Kind(a), ~f: A -> String, xs: List<a, A>) -> String: match xs: case Nil{}: "[]" case h <> t: "[" ++ f(h) ++ List.show.go(~a, ~A, ~f, t)
```

Array

```
def Array.size.node(-T: Type, ys: Array<T>, r: Array<T> & U32) -> Array<T> & U32: (xs2, n) = r (ANode{xs2, ys}, U32.shl(n))

def Array.size(-T: Type, a: Array<T>) -> Array<T> & U32: match a: case ALeaf{x}: (ALeaf{x}, 1) case ANode{xs, ys}: Array.size.node(T, ys, Array.size(T, xs))

def Array.swap.lo(-T: Type, ys: Array<T>, r: Array<T> & T) -> Array<T> & T: (nxs, old) = r (ANode{nxs, ys}, old)

def Array.swap.hi(-T: Type, xs: Array<T>, r: Array<T> & T) -> Array<T> & T: (nys, old) = r (ANode{xs, nys}, old)

law Array.swap.go: for -T: Type for a: Array<T> for n: U32 for +i: U32 for v: T Array<T> & T

def Array.swap.if( -T: Type, xs: Array<T>, ys: Array<T>, +h: U32, i: U32, v: T, z: Bool ) -> Array<T> & T: match z: case True{}: Array.swap.lo(T, ys, Array.swap.go(T, xs, h, i, v)) case False{}: Array.swap.hi(T, xs, Array.swap.go(T, ys, h, U32.sub(i, h), v))

def Array.swap.go(T, a, n, i, v): match a: case ALeaf{x}: (ALeaf{v}, x) case ANode{xs, ys}: +h = U32.shr(n) Array.swap.if(T, xs, ys, h, i, v, U32.is_lt(i, h))

def Array.swap.at(-T: Type, i: U32, v: T, an: Array<T> & U32) -> Array<T> & T: (a, +n) = an Array.swap.go(T, a, n, U32.and(i, U32.sub(n, 1)), v)

def Array.swap(-T: Type, a: Array<T>, +i: U32, v: T) -> Array<T> & T: Array.swap.at(T, i, v, Array.size(T, a))

def Array.clone.node( -T: Data, cx: Array<T> & Array<T>, cy: Array<T> & Array<T> ) -> Array<T> & Array<T>: (xa, xb) = cx (ya, yb) = cy (ANode{xa, ya}, ANode{xb, yb})

def Array.clone(-T: Data, a: Array<T>) -> Array<T> & Array<T>: match a: case ALeaf{+x}: (ALeaf{x}, ALeaf{x}) case ANode{xs, ys}: Array.clone.node(T, Array.clone(T, xs), Array.clone(T, ys))
```

A fork is two handles to one array in O(1). Both read and write the same cells, so racing writes go through the atomics; a match on a handle copies its part, as a clone does. An atomic takes an Array<U32> (fadd an Array<F32>) and returns the array with the old value.

```
@unsafe def Array.fork(-T: Type, +a: Array<T>) -> Array<T> & Array<T>: (a, a)

@unsafe def Array.join(-T: Type, a: Array<T>, b: Array<T>) -> Array<T>: a

law Array.atomic.add: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.min: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.max: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.and: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.or: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.xor: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.exch: for a: Array<U32> for +i: U32 for +v: U32 Array<U32> & U32

law Array.atomic.cas: for a: Array<U32> for +i: U32 for +x: U32 for +v: U32 Array<U32> & U32

law Array.atomic.fadd: for a: Array<F32> for +i: U32 for +v: F32 Array<F32> & F32

def Array.new(-T: Data, +d: Nat, +v: T) -> Array<T>: match d: case 0n: ALeaf{v} case 1n+p: ANode{[v : T^p], [v : T^p]}

def Array.set.fin(-T: Type, r: Array<T> & T) -> Array<T>: (na, old) = r na

def Array.set(-T: Type, a: Array<T>, i: U32, v: T) -> Array<T>: Array.set.fin(T, Array.swap(T, a, i, v))

law Array.get.go: for -T: Data for a: Array<T> for n: U32 for +i: U32 Array<T> & T

def Array.get.if( -T: Data, xs: Array<T>, ys: Array<T>, +h: U32, i: U32, z: Bool ) -> Array<T> & T: match z: case True{}: Array.swap.lo(T, ys, Array.get.go(T, xs, h, i)) case False{}: Array.swap.hi(T, xs, Array.get.go(T, ys, h, U32.sub(i, h)))

def Array.get.go(T, a, n, i): match a: case ALeaf{+x}: (ALeaf{x}, x) case ANode{xs, ys}: +h = U32.shr(n) Array.get.if(T, xs, ys, h, i, U32.is_lt(i, h))

def Array.get.at(-T: Data, i: U32, an: Array<T> & U32) -> Array<T> & T: (a, +n) = an Array.get.go(T, a, n, U32.and(i, U32.sub(n, 1)))

def Array.get(-T: Data, a: Array<T>, +i: U32) -> Array<T> & T: Array.get.at(T, i, Array.size(T, a))

def Array.to_list.go(~T: Type, a: Array<T>, acc: List<T>) -> List<T>: match a: case ALeaf{x}: x <> acc case ANode{xs, ys}: Array.to_list.go(~T, xs, Array.to_list.go(~T, ys, acc))

def Array.to_list(~T: Type, a: Array<T>) -> List<T>: Array.to_list.go(~T, a, Nil{})

def Array.map.go( ~T: Data, ~U: Data, ~f: T -> U, +n: Nat, +i: U32, b: Array<U>, r: Array<T> & T ) -> Array<U>: match n: case 0n: b case 1n+p: (a, x) = r +j = U32.inc(i) Array.map.go(~T, ~U, ~f, p, j, Array.set(U, b, i, f(x)), Array.get(T, a, j))

def Array.map.at( ~T: Data, ~U: Data, ~f: T -> U, v: U, an: Array<T> & U32 ) -> Array<U>: (a, +n) = an Array.map.go(~T, ~U, ~f, U32.to_nat(U32.sub(n, 1)), 1, Array.new(U, U32.log2(n), v), Array.get(T, a, 1))

def Array.map.seed(~T: Data, ~U: Data, ~f: T -> U, r: Array<T> & T) -> Array<U>: (a, x) = r Array.map.at(~T, ~U, ~f, f(x), Array.size(T, a))

def Array.map(~T: Data, ~U: Data, ~f: T -> U, a: Array<T>) -> Array<U>: Array.map.seed(~T, ~U, ~f, Array.get(T, a, 0))
```

Map

```
def Map.bit.u(x: U32, k: Nat) -> Bool: U32.is_ne(U32.and(U32.shrn(x, k), 1), 0)
```

```
def Map.bit.chr(c: Char, off: Nat) -> Char & Bool: match c off: case Chr{x} 0n: (Chr{x}, True{}) case Chr{+x} 1n+b: (Chr{x}, Map.bit.u(x, Nat.sub(31n, b)))

def Map.bit.go.chr(t: String, r: Char & Bool) -> String & Bool: (c2, b) = r (SCon{c2, t}, b)

def Map.bit.go.rec(c: Char, r: String & Bool) -> String & Bool: (t2, b) = r (SCon{c, t2}, b)

def Map.bit.go(key: String, ci: Nat, off: Nat) -> String & Bool: match key: case SNil{}: (SNil{}, False{}) case SCon{c, t}: match ci: case 0n: Map.bit.go.chr(t, Map.bit.chr(c, off)) case 1n+j: Map.bit.go.rec(c, Map.bit.go(t, j, off))

def Map.bit.at(key: String, co: Nat & Nat) -> String & Bool: (ci, off) = co Map.bit.go(key, ci, off)

def Map.bit(key: String, pos: Nat) -> String & Bool: Map.bit.at(key, Nat.divmod(pos, 33n))

law Map.msb.u: for n: Nat for +x: U32 Nat

def Map.msb.u.if(p: Nat, x2: U32, z: Bool) -> Nat: match z: case True{}: 0n case False{}: Nat.add(1n, Map.msb.u(p, U32.shr(x2)))

def Map.msb.u(n, x): match n: case 0n: 0n case 1n+p: Map.msb.u.if(p, x, U32.is_zero(x))

def Map.diff.chr(x: U32) -> Nat: Nat.sub(33n, Map.msb.u(32n, x))

def Map.diff.step(x: Char, y: Char) -> Nat & Bool: match x y: case Chr{+cx} Chr{+cy}: (Map.diff.chr(U32.xor(cx, cy)), U32.is_eq(cx, cy))

law Map.diff: for a: String for b: String Nat

def Map.diff.fin(xt: String, yt: String, rc: Nat & Bool) -> Nat: (r, c) = rc match c: case True{}: Nat.add(33n, Map.diff(xt, yt)) case False{}: r

def Map.diff(a, b): match a b: case SNil{} SNil{}: 0n case SNil{} SCon{h, t}: 0n case SCon{h, t} SNil{}: 0n case SCon{x, xt} SCon{y, yt}: Map.diff.fin(xt, yt, Map.diff.step(x, y))

def Map.new(a, -V: Kind(a)) -> Map<a, V>: MTip{}

law Map.put: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String for x : V Map<a, V>

def Map.put.bit( a, -V: Kind(a), x: V, p2: Nat, lo: Map<a, V>, hi: Map<a, V>, kb: String & Bool ) -> Map<a, V>: (key2, b) = kb match b: case False{}: MNode{p2, Map.put(a, V, lo, key2, x), hi} case True{}: MNode{p2, lo, Map.put(a, V, hi, key2, x)}

def Map.put(a, V, m, key, x): match m: case MTip{}: MLeaf{key, x} case MLeaf{k, v}: MLeaf{k, x} case MNode{+pos, lo, hi}: Map.put.bit(a, V, x, pos, lo, hi, Map.bit(key, pos))

def Map.ins.splice.bit( a, -V: Kind(a), x: V, rest: Map<a, V>, pb: Nat, kb: String & Bool ) -> Map<a, V>: (key2, b) = kb match b: case False{}: MNode{pb, MLeaf{key2, x}, rest} case True{}: MNode{pb, rest, MLeaf{key2, x}}

def Map.ins.splice( a, -V: Kind(a), +p: Nat, key: String, x: V, rest: Map<a, V> ) -> Map<a, V>: Map.ins.splice.bit(a, V, x, rest, p, Map.bit(key, p))

law Map.ins: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String for x : V for +p : Nat Map<a, V>

def Map.ins.deep( a, -V: Kind(a), x: V, lo: Map<a, V>, hi: Map<a, V>, pb: Nat, qb: Nat, kb: String & Bool ) -> Map<a, V>: (key2, b) = kb match b: case False{}: MNode{qb, Map.ins(a, V, lo, key2, x, pb), hi} case True{}: MNode{qb, lo, Map.ins(a, V, hi, key2, x, pb)}

def Map.ins.if( a, -V: Kind(a), key: String, x: V, lo: Map<a, V>, hi: Map<a, V>, +p2: Nat, pb: Nat, t: Bool ) -> Map<a, V>: match t: case False{}: Map.ins.splice(a, V, pb, key, x, MNode{p2, lo, hi}) case True{}: Map.ins.deep(a, V, x, lo, hi, pb, p2, Map.bit(key, p2))

def Map.ins(a, V, m, key, x, p): match m: case MTip{}: MLeaf{key, x} case MLeaf{k, v}: Map.ins.splice(a, V, p, key, x, MLeaf{k, v}) case MNode{+pos, lo, hi}: Map.ins.if(a, V, key, x, lo, hi, pos, p, Nat.is_lt(pos, p))

def Map.lo( a, -V: Kind(a), -R: Type, p2: Nat, hi: Map<a, V>, r0: Map<a, V> & R ) -> Map<a, V> & R: (lo2, r) = r0 (MNode{p2, lo2, hi}, r)

def Map.hi( a, -V: Kind(a), -R: Type, p2: Nat, lo: Map<a, V>, r0: Map<a, V> & R ) -> Map<a, V> & R: (hi2, r) = r0 (MNode{p2, lo, hi2}, r)

law Map.seek: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String Map<a, V> & String & Maybe<&2, String>
```

```
def Map.seek.bit( a, -V: Kind(a), lo: Map<a, V>, hi: Map<a, V>, p2: Nat, kb: String & Bool ) -> Map<a, V> & String & Maybe<&2, String>: (key2, b) = kb match b: case False{}: Map.lo(a, V, String & Maybe<&2, String>, p2, hi, Map.seek(a, V, lo, key2)) case True{}: Map.hi(a, V, String & Maybe<&2, String>, p2, lo, Map.seek(a, V, hi, key2))

def Map.seek(a, V, m, key): match m: case MTip{}: (MTip{}, key, None{}) case MLeaf{+k, v}: (MLeaf{k, v}, key, Some{k}) case MNode{+pos, lo, hi}: Map.seek.bit(a, V, lo, hi, pos, Map.bit(key, pos))

def Map.set.fin.go( a, -V: Kind(a), m: Map<a, V>, key: String, x: V, r: (String & String) & Cmp ) -> Map<a, V>: ((keyb2, k2), c) = r match c: case LT{}: Map.ins(a, V, m, key, x, Map.diff(keyb2, k2)) case EQ{}: Map.put(a, V, m, key, x) case GT{}: Map.ins(a, V, m, key, x, Map.diff(keyb2, k2))

def Map.set.fin( a, -V: Kind(a), m: Map<a, V>, key: String, x: V, keyb: String, k: String ) -> Map<a, V>: Map.set.fin.go(a, V, m, key, x, String.cmp(keyb, k))

def Map.set.go( a, -V: Kind(a), x: V, r: Map<a, V> & String & Maybe<&2, String> ) -> Map<a, V>: (m2, key2, found) = r match found: case None{}: MLeaf{key2, x} case Some{k}: +ka = {key2 : String} Map.set.fin(a, V, m2, ka, x, ka, k)

def Map.set(a, -V: Kind(a), m: Map<a, V>, key: String, x: V) -> Map<a, V>: Map.set.go(a, V, x, Map.seek(a, V, m, key))

def Map.has.leaf(a, -V: Kind(a), v: V, r: (String & String) & Cmp) -> Map<a, V> & Bool: ((key2, k2), c) = r (MLeaf{k2, v}, Cmp.is_eq(c))

law Map.has: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String Map<a, V> & Bool

def Map.has.bit( a, -V: Kind(a), lo: Map<a, V>, hi: Map<a, V>, p2: Nat, kb: String & Bool ) -> Map<a, V> & Bool: (key2, b) = kb match b: case False{}: Map.lo(a, V, Bool, p2, hi, Map.has(a, V, lo, key2)) case True{}: Map.hi(a, V, Bool, p2, lo, Map.has(a, V, hi, key2))

def Map.has(a, V, m, key): match m: case MTip{}: (MTip{}, False{}) case MLeaf{k, v}: Map.has.leaf(a, V, v, String.cmp(key, k)) case MNode{+pos, lo, hi}: Map.has.bit(a, V, lo, hi, pos, Map.bit(key, pos))

def Map.get.leaf(-V: Data, d: V, +v: V, r: (String & String) & Cmp) -> Map<&2, V> & V: ((key2, k2), c) = r match c: case LT{}: (MLeaf{k2, v}, d) case EQ{}: (MLeaf{k2, v}, v) case GT{}: (MLeaf{k2, v}, d)

law Map.get: for -V : Data for d : V for m : Map<&2, V> for key : String Map<&2, V> & V

def Map.get.bit( -V: Data, d: V, lo: Map<&2, V>, hi: Map<&2, V>, p2: Nat, kb: String & Bool ) -> Map<&2, V> & V: (key2, b) = kb match b: case False{}: Map.lo(&2, V, V, p2, hi, Map.get(V, d, lo, key2)) case True{}: Map.hi(&2, V, V, p2, lo, Map.get(V, d, hi, key2))

def Map.get(V, d, m, key): match m: case MTip{}: (MTip{}, d) case MLeaf{k, v}: Map.get.leaf(V, d, v, String.cmp(key, k)) case MNode{+pos, lo, hi}: Map.get.bit(V, d, lo, hi, pos, Map.bit(key, pos))

def Map.pop.lo( a, -V: Kind(a), pos: Nat, hi: Map<a, V>, r0: Map<a, V> & Maybe<a, V> ) -> Map<a, V> & Maybe<a, V>: (lo, r) = r0 match lo: case MTip{}: (hi, r) case lo2: (MNode{pos, lo2, hi}, r)

def Map.pop.hi( a, -V: Kind(a), pos: Nat, lo: Map<a, V>, r0: Map<a, V> & Maybe<a, V> ) -> Map<a, V> & Maybe<a, V>: (hi, r) = r0 match hi: case MTip{}: (lo, r) case hi2: (MNode{pos, lo, hi2}, r)

def Map.pop.leaf(a, -V: Kind(a), v: V, r: (String & String) & Cmp) -> Map<a, V> & Maybe<a, V>: ((key2, k2), c) = r match c: case LT{}: (MLeaf{k2, v}, None{}) case EQ{}: (MTip{}, Some{v}) case GT{}: (MLeaf{k2, v}, None{})

law Map.pop: for -a : Quant for -V : Kind(a) for m : Map<a, V> for key : String Map<a, V> & Maybe<a, V>

def Map.pop.bit( a, -V: Kind(a), lo: Map<a, V>, hi: Map<a, V>, p2: Nat, kb: String & Bool ) -> Map<a, V> & Maybe<a, V>: (key2, b) = kb match b: case False{}: Map.pop.lo(a, V, p2, hi, Map.pop(a, V, lo, key2)) case True{}: Map.pop.hi(a, V, p2, lo, Map.pop(a, V, hi, key2))

def Map.pop(a, V, m, key): match m: case MTip{}: (MTip{}, None{}) case MLeaf{k, v}: Map.pop.leaf(a, V, v, String.cmp(key, k)) case MNode{+pos, lo, hi}: Map.pop.bit(a, V, lo, hi, pos, Map.bit(key, pos))

def Map.del.fin(a, -V: Kind(a), r: Map<a, V> & Maybe<a, V>) -> Map<a, V>: (m2, x) = r m2

def Map.del(a, -V: Kind(a), m: Map<a, V>, key: String) -> Map<a, V>: Map.del.fin(a, V, Map.pop(a, V, m, key))

def Map.to_list.go( a, -V: Kind(a), m: Map<a, V>, acc: List<a, Sigma<&2, a, String, _ => V>> ) -> List<a, Sigma<&2, a, String, _ => V>>: match m: case MTip{}: acc case MLeaf{k, v}: (k, v) <> acc case MNode{pos, lo, hi}: Map.to_list.go(a, V, lo, Map.to_list.go(a, V, hi, acc))

def Map.to_list(a, -V: Kind(a), m: Map<a, V>) -> List<a, Sigma<&2, a, String, _ => V>>: Map.to_list.go(a, V, m, Nil{})
```

```
def Map.keys.go(a, -V: Kind(a), m: Map<a, V>, acc: List<&2, String>) -> List<&2, String>: match m: case MTip{}: acc case MLeaf{k, v}: k <> acc case MNode{pos, lo, hi}: Map.keys.go(a, V, lo, Map.keys.go(a, V, hi, acc))

def Map.keys(a, -V: Kind(a), m: Map<a, V>) -> List<&2, String>: Map.keys.go(a, V, m, Nil{})

def Map.from_list.go( a, -V: Kind(a), kvs: List<a, Sigma<&2, a, String, _ => V>>, m: Map<a, V> ) -> Map<a, V>: match kvs: case Nil{}: m case (k, v) <> t: Map.from_list.go(a, V, t, Map.set(a, V, m, k, v))

def Map.from_list(a, -V: Kind(a), kvs: List<a, Sigma<&2, a, String, _ => V>>) -> Map<a, V>: Map.from_list.go(a, V, kvs, MTip{})

def Map.union(a, -V: Kind(a), m: Map<a, V>, n: Map<a, V>) -> Map<a, V>: Map.from_list.go(a, V, Map.to_list(a, V, n), m)

def Map.size(a, -V: Kind(a), m: Map<a, V>) -> Nat: match m: case MTip{}: 0n case MLeaf{k, v}: 1n case MNode{pos, lo, hi}: Nat.add(Map.size(a, V, lo), Map.size(a, V, hi))

def Map.values.go(a, -V: Kind(a), m: Map<a, V>, acc: List<a, V>) -> List<a, V>: match m: case MTip{}: acc case MLeaf{k, v}: v <> acc case MNode{pos, lo, hi}: Map.values.go(a, V, lo, Map.values.go(a, V, hi, acc))

def Map.values(a, -V: Kind(a), m: Map<a, V>) -> List<a, V>: Map.values.go(a, V, m, Nil{})
```

Set

```
def Set() -> Data: Map<&2, Unit>

def Set.new() -> Set(): Map.new(&2, Unit)

def Set.add(s: Set(), key: String) -> Set(): Map.set(&2, Unit, s, key, Unit{})

def Set.has(s: Set(), key: String) -> Set() & Bool: Map.has(&2, Unit, s, key)

def Set.del(s: Set(), key: String) -> Set(): Map.del(&2, Unit, s, key)

def Set.size(s: Set()) -> Nat: Map.size(&2, Unit, s)

def Set.to_list(s: Set()) -> List<&2, String>: Map.keys(&2, Unit, s)

def Set.from_list.go(keys: List<&2, String>, s: Set()) -> Set(): match keys: case Nil{}: s case h <> t: Set.from_list.go(t, Set.add(s, h))

def Set.from_list(keys: List<&2, String>) -> Set(): Set.from_list.go(keys, Set.new())
```

Image

```
def Image.sink(img: Image) -> Unit: Unit{}

def Image.drop.join(a: Unit, b: Unit, c: Unit, d: Unit) -> Unit: match a b c d: case Unit{} Unit{} Unit{} Unit{}: Unit{}
```

Forks k levels, then a dead parameter sinks the rest in one loop.

```
def Image.free(+k: Nat, img: Image) -> Unit: match k: case 0n: Image.sink(img) case 1n+e: match img: case Pix{c}: Unit{} case Qua{tl, tr, bl, br}: a b c d = Image.free(e, tl) Image.free(e, tr) Image.free(e, bl) Image.free(e, br) Image.drop.join(a, b, c, d)

def Image.drop(img: Image) -> Unit: Image.free(7n, img)
```

App

```
def App.next( -S: Type, window: Window, rest: Window -> S -> IO(Unit), next: Maybe<S> ) -> IO(Unit): match next: case None{}: Window.close(window) case Some{state}: rest(window, state)

def App.turn( -S: Type, shown: Window & Image & List<Event>, tick: List<Event> -> S -> IO(Maybe<S>), state: S, rest: Window -> S -> IO(Unit) ) -> IO(Unit): (window, image, events) = shown do IO<Unit>: Unit <- IO.pure(Unit, Image.drop!(image)) IO.bind(Maybe<S>, Unit, tick(events, state), App.next(S, window, rest))
```

def App.draw(-S: Type, window: Window, drawn: S & Image, tick: List<Event> -> S -> IO(Maybe<S>), rest: Window -> S -> IO(Unit)) -> IO(Unit): (state, image) = drawn do IO<Unit>: shown : Window & Image & List<Event> <- Window.frame(window, image) App.turn(S, shown, tick, state, rest)

def App.step(-S: Type, app: App<S>, window: Window, state: S, rest: Window -> S -> IO(Unit)) -> IO(Unit): App{view, tick} = app App.draw(S, window, view(state), tick, rest)

def App.loop(~S: Type, ~app: App<S>, fuel: Nat, window: Window, state: S) -> IO(Unit): match fuel: case 0n: Window.close(window) case 1n+f: App.step(S, app, window, state, w => s => App.loop(~S, ~app, f, w, s))

def App.run(~S: Type, ~app: App<S>, title: String, width: U32, height: U32, state: S) -> IO(Unit): do IO<Unit>: window : Window <- IO.try(Window, Window.open(title, width, height)) App.loop(~S, ~app, U32.to_nat(4294967295), window, state)

def App.more(-S: Type, rest: S -> IO(Maybe<S>), next: Maybe<S>) -> IO(Maybe<S>): match next: case None{}: IO.pure(Maybe<S>, None{}) case Some{state}: rest(state)

def App.fold(-S: Type, app: App<S>, events: List<Event>, state: S, rest: S -> IO(Maybe<S>)) -> IO(Maybe<S>): App{view, tick} = app IO.bind(Maybe<S>, Maybe<S>, tick(events, state), App.more(S, rest))

def App.play(~S: Type, ~app: App<S>, frames: List<List<Event>>, state: S) -> IO(Maybe<S>): match frames: case Nil{}: IO.pure(Maybe<S>, Some{state}) case events <> rest: App.fold(S, app, events, state, s => App.play(~S, ~app, rest, s))

FILE: 02-guides-cli/raw/BASE-TYPES-DUMP.md

Source: `bend2-knowledge/02-guides-cli/raw/BASE-TYPES-DUMP.md`

type Empty is Data:

type Unit is Data: Unit{}

type Bool is Data: False{} True{}

type Cmp is Data: LT{} EQ{} GT{}

type Either<a, b, -A: Kind(a), -B: Kind(b)> is Kind(a <&> b): Inl{value: A} Inr{value: B}

type Sigma<a, b, -A: Kind(a), -B: @-x: A -> Kind(b)> is Kind(a <&> b): Tuple{fst: A, snd: B(fst)}

type Nat is Data: Zero{} Succ{pred: Nat}

type Maybe<a, -A: Kind(a)> is Kind(a): None{} Some{value: A}

type Result<a, b, -E: Kind(a), -A: Kind(b)> is Kind(a <&> b): Fail{error: E} Done{value: A}

type List<a, -A: Kind(a)> is Kind(a): Nil{} Con{head: A, tail: List<a, A>}

law Word: for n: Nat Data

type Word.Nil is Data: WNil{}

type Word.Con<-p: Nat> is Data: WCon{head: Bool, tail: Word(p)}

type U32 is Data: U32{data: Word(32n)}

type F32 is Data: F32{data: Word(32n)}

type Char is Data: Chr{code: U32}

type String is Data: SNil{} SCon{head: Char, tail: String}

type Array<-T: Type> is Type: ALeaf{value: T} ANode{xs: Array<T>, ys: Array<T>}

type Image is Data: Pix{color: U32} Qua{tl: Image, tr: Image, bl: Image, br: Image}

type Event is Data: Key{code: U32, down: Bool} Mouse{x: U32, y: U32, button: U32, down: Bool} Move{x: U32, y: U32} Close{}

type Map<a, -V: Kind(a)> is Kind(a): MTip{} MLeaf{key: String, val: V} MNode{pos: Nat, lo: Map<a, V>, hi: Map<a, V>}

law File: Type

law Socket: Type

law Listener: Type

law Window: Type

law Audio: Type

law Chan: for -A: Type Data

type IO.OP<-R: Type> is Type: Emit{value: R} Halt{code: U32, message: String}

law Pair: for -A: Type for -B: Type Type

law IO: for -A: Type Type

type App<-S: Type> is Type: App{view: S -> Pair(S, Image), tick: List<Event> -> S -> IO(Maybe<S>)}

FILE: 02-guides-cli/raw/EFFECTS-CLI-DUMP.md

Source: `bend2-knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md`

Effects in Bend

An AI wrote this text from the runtime's source (`bend2/comp.ts` and `bend2/effs/`). A human will revise it later. Report anything wrong on GitHub.

AN EFFECT DEF

An effect is a def of type `IO(R)` whose body is two imports:

```
def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"
```

The host function is named after the def: lowercased, dots to underscores (`clock_now`). The `.c` file serves `bend x.bend -o x` ; the `.js` file serves `bend x.bend -o x.js` and `bend x.bend` . Every effect in Base is built this way: `bend2/effs/*.c` and `*.js` are the reference.

THE C SIDE

The compiler splices your `.c` into the program's C source after the runtime, so every runtime symbol is in scope. Define the effect and register it in a constructor:

```
Term clock_now_run(Env e, Term* f, IoWork* w);

static void __attribute__((constructor)) clock_now_use(void) {
  io_eff(CID_CLOCK_NOW, clock_now_run, 0);
}
```

`CID_X` is the def's name uppercased, dots to underscores. `f` holds the def's arguments in order: a `U32` is the word (`(u32)f[0]`), a `String` is taken with `io_cstr(e, f[0], &len)` (a `malloc` ed copy you free), a handle with `io_hand_v(f[0])` . The last argument of `io_eff` is the need: `0` runs the effect at once; `IO_READ` parks it until the handle in `f[0]` is readable; `IO_TIME` parks it for `f[0]` milliseconds. Then the loop calls the effect.

The effect returns a Term: a `U32` is `(Term)n` , `Unit` is `term_pak(CID_UNIT, 0)` , a `String` is `io_str(e, p, n)` , a two-field constructor is `io_node(e, CID_K, a, b, 0)` . A `Result` is `io_done(e, v)` or `io_fail(e, code, text)` (`text` NULL prints `strerror(code)`). `io_sys_end(w, n)` stores `errno` in `w->code` when a call fails.

A handle is a descriptor or pointer packed in one word by `io_hand(v)` . Its type must be one of Base's handle laws (`File` , `Socket` , ...): a user handle type is a WONTFIX entry for now, see `WONTFIX.txt` . An effect on a handle hands it back beside its result: `io_tup(e, io_hand(h), r)` , also on failure.

Blocking work leaves the loop in two ways. `io_work(w, call, pack)` runs `call(w)` on a helper thread, then `pack(e, w)` on the loop; `pack` 's value is the answer. `call` has no `Env` : it may only touch `w` . `io_wait_on(w, fd, POLLIN, more)` parks until `fd` is ready (`POLLIN` or `POLLOUT`), then runs `more(e, w)` on the loop; `more` may call `io_wait_on` again. Both return `IO_PARK` , which the effect returns.

`w` is the effect's scratch space: `hand`, `made`, `word`, `size`, `data`, `text`, `code`. The runtime owns `w` and `f`. The effect owns `w->data`: allocate it in the run function and free it in `pack`. In `bend2/effs/`, `file_read.c` is the pattern for `io_work` and `tcp_recv.c` for `io_wait_on`.

THE JS SIDE

The `.js` file is a plain script, wrapped in a closure; the compiler finds the function by name. It takes the def's arguments as JS values: a `U32` is a number, a `Nat` a `BigInt`, a `String` a string, a constructor `{$: "Name", field: value}`, a handle its host value (a descriptor). It returns the answer the same way: `io_done(v)`, `io_fail(code)`, `io_tup(handle, result)`, `{$: "Unit"}`.

A need is a second function, `clock_now_need`, that returns `{read: true}` or `{time: true}`. A blocking effect takes one more argument, `k`, and parks with `io_park_on(fd, out, k, more)`: it returns `undefined`, and the loop calls `more()` when `fd` is ready; `more` answers the value, or `undefined` to park again. `io_sys()` is `libc` through `bun:ffi` (`read`, `recv`, `poll`, `errno`); `tcp_recv.js` shows the full shape.

A COMPLETE EXAMPLE

`main.bend`:

```
import Base

def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"

def main() -> IO(Unit):
  do IO<Unit>:
    t : U32 <- Clock.now()
    IO.print("ms since boot: " ++ U32.show(t))
```

`clock.c`:

```
Term clock_now_run(Env e, Term* f, IoWork* w) {
  return (Term)(uint32_t)(io_tick() / 1000000ull);
}

static void __attribute__((constructor)) clock_now_use(void) {
  io_eff(CID_CLOCK_NOW, clock_now_run, 0);
}
```

`clock.js`:

```
function clock_now() {
  return Math.floor(performance.now()) >>> 0;
}
```

`bend main.bend -o main && ./main` prints a line; so do `bend main.bend -o main.js && bun main.js` and `bend main.bend`. The output name must not be one of the effect's files.

COMPATIBILITY

The C side tracks the exact compiler version: the names above are the runtime's internals, and a release may rename any of them. There is no ABI promise. Rebuild your effects with every update.

FILE: 02-guides-cli/raw/GUIDE-CLI-DUMP.md

Source: `bend2-knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md`

Bend

Bend is a new programming language that combines Lean-like formal proofs with C-like speeds and CUDA-like parallelism. It gives humans an ambiguity-free language to communicate their intents to AIs, a compiler capable of mechanically checking that the AI implemented these intents to unquestionable mathematical correctness, and a

compiler that runs that code fast on CPUs and GPUs.

HELLO, WORLD!

Bend's syntax is Python-shaped, but its semantics are closer to Haskell / Lean, while being resource-aware like Rust (if less annoyingly). A hello world is just a typed definition returning an IO block:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

To run it, install Bend (`curl -fsSL https://bend-lang.com/install.sh | sh`) and type `bend hello.bend` .

Bend is a *pure language*, with effects denoted via a Haskell-inspired IO Monad `(https://wiki.haskell.org/Introduction_to_IO)`. It comes with a list of built-in effects for files, networking, audio, graphics, input, and more, and the user can extend them with foreign C and JS imports; more on that later.

CORE FEATURES

TYPES AND FUNCTIONS

A typical Bend program is a set of datatypes and functions over them:

```
import Base

type Shape is Data:
  Circle{r: U32}
  Square{s: U32}

def area(x: Shape) -> U32:
  match x:
    case Circle{+r}:
      (3 * r * r : U32)
    case Square{+s}:
      (s * s : U32)

def main() -> U32:
  area(Square{5})
```

Bend, by default, is *affine*, meaning variables must be used, at most, once.

Here, `is Data` declares that `Shape` may be copied, while `is Type` keeps it non-copiable. The `+` annotation before a variable name allows using it more than once, if the variable is Data-kinded.

Bend does almost no inference, meaning it requires more annotations than similar languages. This is what allows Bend's checker to be significantly faster than other provers, and its error messages more precise, at the expense of programs and proofs being more verbose. When the checker can't decide the type of an expression, just annotate it, as in `{3 : U32}` .

CLOSURES

Functions are values and can be stored, passed, and returned from other functions.

```
import Base

def adder(k: U32) -> U32 -> U32:
  x => (x + k : U32)

def main() -> U32:
  add2 = adder(2)
  add5 = adder(5)
  add5(add2(1))
```

A closure is affine: it can be called at most once, even when everything it captures is `Data` . Only top-level definitions can be called freely. Partial applications like `U32.add(2)` are closures too.

RECURSION AND TERMINATION

Bend uses recursion to repeat work. Tail calls compile to loops.

```
import Base

def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}:
      acc
    case Con{h, t}:
      sum(t, (acc + h : U32))

def main() -> U32:
  sum([1, 2, 3, 4], 0)
```

Here, `t` has one fewer element than `xs`, so `sum` eventually reaches the empty list. Bend verifies termination by requiring recursive calls to use smaller parts of their inputs, obtained through pattern matching. The check reads the arguments of a recursive call from left to right: each must be passed unchanged until one is a smaller part of its parameter, and the ones after it are free. So, put the parameter that shrinks first. Also, is no `if` syntax yet. Use `match` on `True{}` and `False{}` instead.

Termination is mandatory and mutual recursion is not allowed. Both restrictions keep Bend's proofs sound, as a function that never returns could otherwise prove anything. A loop bounded by the outside world, like a server's, counts down a `Nat` fuel argument instead, and two mutually recursive functions become one def with an extra argument selecting which to run. A `def` marked `@unsafe` recurses freely, but falls outside Bend's proof guarantees.

A `match` inspects a parameter or a variable bound by a pattern, never a computed value: `match sum(xs, 0):` is rejected. Scrutinees follow binder order, and a `let` may not precede a `match` on a parameter. To match on a computed value, pass it to a helper that matches on its parameter.

PARALLELISM

Bend's parallelism primitive is the parallel call notation:

```
import Base

def pow2(+n: Nat) -> U32:
  match n:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p) # parallel call
      (a + b : U32)

def main() -> IO(Unit):
  IO.print(U32.show(pow2!(20n))) # `!` runs on GPU
```

A parallel call promises the compiler two things:

1. The calls are independent.
2. They run in roughly the same time.

Since Bend is pure and affine, the first point always holds. The second is yours to keep: if one call finishes before the other, the speedup will be sub-ideal. Bend's current scheduler is a contention-free, binary fork-join machine: every task is handed to a core exactly once and never moved afterwards. That makes it fast and GPU-friendly, but you must keep the workload balanced.

A `!` after a function name marks a parallel call: `pow2!(20n)` hands that call, and every parallel call inside it, to the GPU. When compiled to a native executable, `pow2(20n)` runs in parallel on the CPU, while `pow2!(20n)` runs on the GPU. The heap is fully unified, so, if your chip has unified memory (as in Apple M-series processors), moving data from the CPU to the GPU is a zero-cost operation. The GPU shines on uniform numeric work like mandelbrot or nbody; divergent work like n-queens stays faster on the CPU. A machine without a GPU runs `!` on the CPU (still in parallel). What the lanes share also sets the speed: a `+` value read by every lane costs an atomic per read. Read `bend guide shaders` before you write a parallel app.

The JavaScript target ignores all that and just runs sequentially.

ARRAYS

Arrays give Bend in-place mutation without giving up purity:

```
import Base

def main() -> Array<U32> & U32:
  a = [0 : U32*8n] # new array with 8 copies of 0
  a[5] <- 42       # performs an in-place rewrite
  a[5]             # reads index 5
```

An `Array<T>` is a `Type`, so it has exactly one owner at all times. That is what lets `a[5] <- 42` overwrite the slot and hand back the same array, with no copy. A read hands the array back next to the element for the same reason: if it returned only the element, the array would be gone. Indexes wrap around. A write followed by another statement re-binds its array: `a[5] <- 42` on its own line is `a = a[5] <- 42`. As the last statement it is the written array. The slot count after `*` is a power of two; `[0 : U32^3n]` names the depth instead.

The `a[i]` sugar assumes `Array<U32>`. For other element types, call `Array.get` (`Data` elements; else `Array.swap`) and `Array.set` directly, and `Array.clone` when you need two copies. Read Bend's Base for reference. This will be generalized soon!

QUANTITIES

A quantity says how many times a variable may be used.

```
import Base

# -A: erased (gone at runtime)
#  n: affine (used at most once)
# +x: reusable (requires A to be Data)
def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n:
      Nil{}
    case 1n+p:
      x <> replicate(A, p, x)

def main() -> List<U32>:
  replicate(U32, 3n, 7)
```

Erased variables can only appear in types and proofs: the checker sees them, the compiler deletes them. Affine variables are the default, and dropping one is always free. Reusable variables require `Data`: functions, arrays and IO handles are `Type`, so they can never be copied. Note that `main` marks nothing: you write `+` where you need the copies, and `replicate` pays for them with a reference count at runtime. Matching a `+` value hands out `+` fields; on a plain one, write `+r` in the pattern to make a field reusable.

KINDS

Every type has a kind, which caps how many times its values may be used.

```
import Base

#  a: a quantity (&0, &1 or &2)
# -A: a type whose values may be used a times
def length(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{h, t}:
      1n+length(a, A, t)

def main() -> Nat:
  length(&2, U32, [1, 2, 3])
```

`Type` is short for `Kind(&1)` and `Data` for `Kind(&2)`, so a `Kind(a)` parameter accepts both: `length(&1, U32 -> U32, fs)` counts a list of closures just as well. A bare `a` in a parameter list is short for `-a: Quant`. Base declares `type List<a, -A: Kind(a)> is Kind(a)`, making a list exactly as reusable as its elements: `List<U32>` is short for `List<&1, U32>`, and

`+List<U32>` for `List<&2, U32>`. A type holding two element types combines their quantities with `a <&> b`, the smaller of the two.

TEMPLATES

A template receives its argument as syntax and inlines it at compile time.

```
import Base

# ~f: substituted at compile time, not passed at runtime
def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

Template parameters come first in the parameter list, and a `~` argument must be closed: it may mention top-level defs, but no local variable of the caller. Each distinct set of `~` arguments compiles to its own copy of `twice`, so `f` costs nothing at runtime and, unlike a closure, may be called as many times as you like. A template may call only templates declared above it. This is how `List.map` is written in Base.

LAWS AND PROOFS

A law states a fact that must hold. It must be proven inside a paired def.

```
import Base

# LAW: "for every x, x plus 0 equals x"
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

# PROOF: case analysis:
# - base case: reflexivity
# - step case: induction, rewrite, reflexivity
def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> {Nat.add(2n, 0n) == 2n : Nat}:
  add_zero(2n)
```

Laws are a critical feature in Bend, as they provide an ambiguity-free language on which humans can state precise specs for AI's to implement. That is, instead of writing a natural language prompt such as "implement a function that sorts a list", users can write precise laws like "implement a function F such that, for every list of numbers, `F(list)` returns the same numbers in ascending order". Models are then guaranteed to respond with bug-free code, since Bend will demand that they provide an actual proof.

> We envision that "law-driven development" will eventually become the way humans use AI to write and maintain large codebases, as it is the perfect middle point between having to code everything manually (laborious) and letting AI do it all via prompts without auditing a line of code (error/ambiguity-prone, unsecure).

By convention, a project keeps its laws in two files at its root. `LAWS.bend` imports the code and states the laws, each an open claim: the human writes it, the AI does not touch it. `PROOF.bend` imports `LAWS.bend` and proves each law with a def of the same name (`law sorted` is proven by `def Laws.sorted`): the AI writes it, along with the code. `bend PROOF.bend` is the gate: it fails while any law is open or false, and prints "All terms check." once every law holds. bend refuses a `PROOF.bend` that sits beside a `LAWS.bend` without importing it.

Bend has no tactics: a proposition is a type, and a proof is a def of that type. `{a == b : T}` is an equality; `{==}` proves it when both sides compute to the same term. Matching refines the goal in each case, a recursive call is the induction hypothesis, and `%e : P` rewrites with `e : {a == b : T}` : `P` is the goal with `_` marking `b`, and the goal becomes `P` with

`a` there. `exs y: T` in a law asks for a witness, returned as `(y, proof)`. A failed step prints the expected and observed terms; `?name` prints the goal, `?TODO` leaves it open, and a law with no def is an open claim.

Since types are terms, a def may return a `Type`, like `def IsEven(n: Nat) -> Type:`, which is all dependent types are. A `match e:` with no cases closes a branch where `e : Empty`. To refute a clash like `e : {1n == 0n : Nat}`, rewrite it through a motive `disc(_)`, where `disc` sends `0n` to `Empty` and `1n+p` to `Unit`, and answer `Unit{}`. `{a != b : T}` is `{a == b : T} -> Empty`, and `Equal.sym`, `Equal.trans` and `Equal.cong` are in Base.

IO AND CONCURRENCY

Effects live in the `IO` type and are sequenced with `do` blocks:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1000)             # a step
    return "Hello, " ++ name   # return: wraps a pure value

def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER")) # <-: binds a result
    chan : Chan(String) <- IO.fork(String, greet(name)) # runs concurrently
    IO.print("Waiting...")
    text : String <- IO.join(String, chan)
    IO.print(text)
```

Every bind is annotated, and `x : T = v` binds a pure value in the middle of a block. A fallible effect answers `Result<&1, &1, U32 & String, A>`: `IO.try` unwraps it or exits with the error, and `IO.die` exits with your own. `IO.args` answers the command line, less the runtime's own options (a `--` ends them). A handle (`File`, `Socket`, `Window`) is an affine, opaque value, so every effect on one hands it back beside its result, and no program can forge or reuse one.

A Bend program is a set of computations interleaved by one event loop, as in Node.js: each runs its pure code (in parallel, on every core) up to its next effect, and one that waits on a socket, a sleep or a channel steps aside for the others. `IO.fork` starts a computation and returns the channel its result will arrive on; `IO.join` waits for it. Underneath are `IO.spawn`, `Chan.new`, `Chan.send`, `Chan.recv` and `Chan.close`. The program ends when every computation is done, or reports a deadlock when the remaining ones all wait.

Every effect in Base is a def whose body is `import "./x.js"` plus a `.c` twin, implemented by a host function named after the def, lowercased, dots to underscores. You can add your own effects the same way. Only the event loop runs them, so proofs, termination and the GPU never touch host code. In the other direction, a JS file may `import Game from "./game.bend"` (with `bend2/main.ts` preloaded) and call every non-IO def, with constructors as `{$: "Name", field: value}` and `Nat` as `BigInt`. A value crosses without a copy: an `Array` argument is the caller's own array, updated in place, so copy it first if you keep it.

MONADS

The `do` notation works for any monad, not just IO.

```
import Base

def add_strs(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a) # a None here ends the block with None
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> Maybe<&2, U32>:
  add_strs("40", "2")
```

A `do M<xs.., R>:` block desugars each `x : A <- v` into `M.bind(xs.., A, R, v, x => ..)` and each `return e` into `M.pure(xs.., R, e)`, so any type with those two defs works: `IO`, `Maybe`, `Result`, or your own. The leading arguments (here, the `&2` quantity) are passed along to both.

APPS

Graphics in Bend are pure: a frame is an `Image`; `App` maps states to images.

```
import Base

# tick: folds a frame's events into the next state; None quits the app
def tick(events: List<Event>, color: U32) -> IO(Maybe<U32>):
  match events:
    case Nil{}:
      IO.pure(Maybe<U32>, Some{color})
    case Con{Close{}, rest}:
      IO.pure(Maybe<U32>, None{})
    case Con{e, rest}:
      tick(rest, (color + 1 : U32))

def main() -> IO(Unit):
  # view: returns the state beside its image; Pix paints the whole frame
  App.run(~U32, ~App{+s => (s, Pix{s}), tick}, "Hello", 256, 256, 0)
```

An `Image` is a quadtree: `Pix{color}` paints a square, and `Qua{tl, tr, bl, br}` splits it in four, so a frame is drawn by recursion like everything else, in parallel if you want. Events are `Key`, `Mouse`, `Move` and `Close`. `App.run` opens a window and calls `view` then `tick` once per frame, until `tick` answers `None`. Since the state is affine, `view` must hand it back next to the image. Underneath are `Window.open`, `Window.frame` and `Window.close`, and `Audio.open`, `Audio.write` and `Audio.close` for sound. See `demos/app_pong_game_2d` for a complete one.

THE BASE LIBRARY

Base is small, and its names follow a scheme, so you can guess most of it:

```
import Base

def main() -> String:
  +a = (6 * 7 : U32)         # sugar for U32.mul(6, 7)
  b  = U32.to_nat(a)         # conversions are T.to_x and T.from_x
  U32.show(a) ++ " = " ++ Nat.show(b)
```

Every def is named `Type.verb`, and the same verbs recur across `Nat`, `U32` and `F32`: `add sub mul div mod` for arithmetic, `and or xor not shl shr` for bits (`U32` only), `cmp` (returning `Cmp`, not on `F32`) and `is_eq is_ne is_lt is_le is_gt is_ge` (returning `Bool`) for comparisons, `show` to `String` and `read` back from it (answering a `Maybe`). Operators and `< -` style comparisons are just sugar for these. Beyond numbers there are `Bool`, `Cmp`, `Maybe`, `Result`, `List`, `Array`, a string-keyed `Map` (`new set get has del keys`; `get` takes a default, and `get` and `has` hand the map back beside their result), `Set` on top of it, the `Equal` lemmas, and the effects. `bend base` prints all of it, `bend base --types` only the types, and `bend base Map` one name and everything under it.

MODULES

A module is a file, and an import gives it a local name:

```
# math.bend
import Base

def square(+x: U32) -> U32:
  (x * x : U32)
```

```
# main.bend
import Base
import ./math.bend as M  # M.x now names every def of math.bend

def main() -> U32:
  M.square(7)
```

The alias is local to the importing file, and dots inside a name are just characters: `U32.show` needs no module. A law left open in one file may be filled in another as `def M.name(..)`, so a proof can ship separately from its claim. `import 0x<hash>/main.bend as P` imports a package by content hash, fetched from the hub and checked against it; `bend main.bend --publish` uploads a file with everything it imports and prints that line.

TOOLING

Bend is a single command:

```
bend file.bend            # check; run main (IO compiled; a value normalized)
bend file.bend -o file    # compile to a native binary (clang 14+; 19+ with `!`)
bend file.bend -o file.c  # emit the C source instead
bend file.bend -o file.js # emit the JS source instead
bend page.html -o dist    # bundle a web page that imports .bend files
./file --threads 8        # run a native binary on 8 CPU threads
./file --gpu off          # run ! calls on the CPU (the GPU is on by default)
./file --gpu 4GB          # cap the GPU's heap at 4GB
```

A `main` that returns `IO` runs compiled; one that returns a value is normalized by the checker (slow for big work) and printed; a file with no `main` just checks. A binary that uses `!` builds its GPU program too, as `file.gpu`, which must stay beside it: on macOS it needs Metal, on Linux CUDA 12 at `/usr/local/cuda`. On Linux a program with a Window needs `libx11-dev`, one with Audio `libasound2-dev`. `bend guide` prints this text, `bend base` prints the Base library (`bend base Map` prints one name and everything under it), and `bend --help` lists the other commands.

SYNTAX REFERENCE

Every form of the language, grouped by where it appears. Operators, literals and brackets are sugar for names in Base.

```
# Top level
import Base                             # the prelude
import ./file.bend as M                 # a module; its defs are M.x
type D<a, -A: Kind(a)> is Kind(a):      # a datatype and its kind
  K{x: A, xs: List<a, A>}               # one constructor per line
def f(x: A, -y: B, +z: C) -> T:         # a def; the body follows
def f(x, y):                            # fills the law named f
def t(~g: A -> B, x: A) -> B:           # a template
law f:                                  # a claim, proven by def f
  for x: A                              # a parameter (also for -x, for +x)
  for y: B where P(y)                   # y is then the pair (y, P(y) proof)
  exs z: C                              # a witness the proof must return
  T                                     # the claim
@unsafe def f(x: A) -> T:               # skips the termination check
def e(x: A) -> IO(B):                   # a foreign effect
  import "./e.c"
  import "./e.js"
```

```
# Types
Type  Data  Kind(q)                     # kinds; Type = Kind(&1), Data = Kind(&2)
Quant  &0  &1  &2  a <&> b              # quantities and their minimum
A -> B  @x:A -> B  @-x:A -> B           # functions: plain, dependent, erased
A & B  &x:A -> B  A | B                 # pairs, dependent pairs, sums
D<A>  +D<A>  D<&2, A>                   # a datatype; + makes it reusable
{a == b : T}  {a != b : T}              # equality and its negation
```

```
# Terms
42  1.5  3n  'c'  "s"                   # U32, F32, Nat, Char, String
[a, b]  h <> t  (a, b)                  # a list, a cons, a tuple
K{a, b}  x => e  +x => e                # a constructor, a lambda
f(a, b)  f!(a)  t(~g, a)                # a call, on the GPU, of a template
(a + b * c : T)  {x : T}                # operators over T; an annotation
[v : T*n]  [v : T^d]  a[i]  a[i] <- v   # an array of n or 2^d slots; a read, a write
{==}  %e : P; e2  %e@E : P; e2          # reflexivity, a rewrite, a named one
?name  ?TODO                            # print the goal; leave it open
```

```
# Statements (a def, case, lambda or parenthesized body)
x = v  +x = v  -x = v                   # a let: affine, reusable, erased
(a, b) = v  K{x, y} = v                 # a destructuring let
a b = f(x) g(y)                         # a parallel let
match a b:                              # a match on one or more values
  case K{x, _} 1n+p:                    # patterns nest; _ catches the rest
do M<xs.., R>:                          # a monadic block over M.bind, M.pure
  x : A <- m                            # bind
  x : A = v                             # let
  m                                     # a Unit step
  return v                              # the result
```

Inside `(.. : T)`, `+ - * / %` call `T.add` through `T.mod`, `.&. .|. .^.` the bit operations, `<< >>` the shifts (by a `Nat`), and `< <= > >=` the `T.is_lt` family; without a `: T` they belong to `Nat`. `&& ||` work on `Bool` and `++` on `String` anywhere. Operators need spaces on both sides. Equality of values is a call, `T.is_eq(a, b)`; `==` is only the type. A `Nat` literal past `256n` is `U32.to_nat(n)` underneath, up to `4294967295n`.

UNDER THE HOOD

Bend's compiler emits one C file, and that file is both the CPU program and the GPU kernel. clang compiles it for the CPU. Metal (on Apple) or CUDA (on NVIDIA) compiles the same file for the GPU. So a `!` call runs the same code on whichever chip it lands on.

A term is one 64-bit word: small values are stored inline, everything else is a pointer into a single heap shared by every core and by the GPU. There is no garbage collector. Since values are affine, a `match` frees the node it opens on the spot, and only `+` values carry a reference count. There is no C stack either: each def compiles to a segment of a flat state machine, a call is a jump, and a parallel call creates a join task plus one task per call, which the scheduler deals across CPU or GPU lanes. `paper/BendRT.pdf` has the design and the benchmarks.

Bend's theory has one universe and no positivity check: `Type : Type` holds, and a datatype may recurse on the left of an arrow. What keeps this consistent is a wall between two checking modes. Code that runs is checked *live*; types, erased arguments and equations are checked *dead*. Dead code may loop forever or inhabit `Empty`, but nothing dead ever counts as live evidence, and live recursion must terminate. `bend2/bend.lean` mechanizes this, though it lags `bend.ts`; `paper/BendTT.pdf` is the paper.

FURTHER READING

- `demos/`: complete programs, including the game and its proof from the video.
- `bend2/base.bend`: the Base library, also printed by `bend base`.
- `paper/BendTT.pdf` and `paper/BendRT.pdf`: the type theory and the runtime.

EXTRA

`bend guide shaders` prints "Shaders in Bend", a tutorial written by AIs for AIs on how to write efficient shaders in Bend. It distills what building `demos/app_slash_boss_3d` (120 FPS in pure Bend) taught. Read it before you write a graphical or parallel app in Bend.

`bend guide effects` prints "Effects in Bend", an AI-written note (to be revised by a human) on the C and JS side of custom effects. Read it before you write one.

FILE: 02-guides-cli/raw/SHADERS-CLI-DUMP.md

Source: *`bend2-knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md`*

Shaders in Bend

AIs wrote this tutorial, for AIs, from the logs of building `demos/app_slash_boss_3d`. A human will revise it later. Report anything wrong on GitHub.

A fast shader in Bend is one fork tree that ends in flat loops. The reference is `demos/app_slash_boss_3d/bend3d.bend`, a rasterizer that draws 1920x1200 at 120 FPS (8.0 ms a frame) on a 10-core M4 mini. Every ms below is that machine and that frame unless stated.

THE MACHINE

A `!` call (a bang) ships one call to the GPU, which runs a cube of 16384 lanes (128 x 128). A parallel let `a b = f(x) g(y)` creates one task per call plus a join task. A kernel iteration grows the fork tree breadth-first until the frontier fills the cube, then every lane runs its task to the end: ~0.4-0.5 ms plus the slowest lane. A continuation that forks again lands in the next iteration. A def is flat when it has no parallel let, no bang, no closure apply, no non-flat call and no non-tail self call: it compiles to one native tail loop (`spin_N` in the C). A non-tail self call becomes a segmented continuation: a frame per call. Memory is one heap shared by the CPU and the GPU: unified on Apple; managed on CUDA, where each page faults across PCIe on the first touch by the other side. A machine with no GPU, or `--gpu off`, runs bangs on the CPU pool. Divergent per-lane work (a search, a tree walk per ray) is faster there than on the GPU. Aim for 4^7 leaves per bang, one per lane, and sweep one fork level either side on the device: the CPU pool is flat across that sweep, so it cannot tune it. Fewer leaves leave lanes idle (256 of 16384 busy: 234 ms for a frame that takes 28 ms at 4096); more cost an iteration each.

A FRAME

1. Host: build the scene (mesh, project, light, cull) into a quadtree of 64-px cells, one cons list of flat screen triangles per cell. 1.6 ms.
2. One bang: fork four ways per node, five levels to the cells, two more over each cell's list, to 16-px tiles: 4^7 = 16384 = the lane cube. A tile filters the cell's list into its own and walks it in flat loops. Draw ~6 ms.
3. The last frame rides down the same tree as `Four` quadrants; each tile leaves its quadrant unused, so the runtime sinks it there. Drop 0 ms.
4. Host: free the last scene as one branch of the next build. 0.35 ms.

```
2048 root   Frame.node, k=5   fork 4   levels 1-5  -> 64-px cells (32 x 32)
64-px cell  Cell.fork, k=2    fork 4   levels 6-7  -> 16-px tiles (16384)
16-px tile  Tile.b16 = 4 Tile.b8 = 4 Tile.b4       straight-line, no fork
4-px square Blk.go            one flat loop over the list, four Blk
2-px block  Blk.four          one Pix, or four
```

The root is the power of two over the frame (2048 over 1920x1200); a region off the frame, or empty, answers one `Pix` before any fork (`Frame.sky`). A fork per pixel drowns in scheduling; a fork per cell leaves lanes idle. A fold per level plus a fold per pixel: 600 ms and 31 iterations a frame, against 8 ms and 1.

THE CODE

The list is the one shape a flat loop walks with no stack. One list per cell, not per pixel: a list shared by pixels is contended atomics.

```
type Tris is Data:
  TNil{}
  TCons{+tri: Tri, next: Tris}

type Cells is Data:
  CNil{}
  CLeaf{+n: U32, +ts: Tris}
  CQua{tl: Cells, tr: Cells, bl: Cells, br: Cells}
```

The fork. `Frame.show` bangs once and returns the tree beside the image, so the host still owns it. `Image.open(old) -> Four` splits the last image at the call (a `Pix` opens into four); each tile receives `old: Four` and ignores it.

```
def Frame.node(+k: Nat, c: Cells, old: Four) -> Image:
  match k c:
    case _ CLeaf{_, ts}:
      Cell.fork(2n, ts, 0, old)
    case 1n+e CQua{tl, tr, bl, br}:
      Four{oa, ob, oc, od} = old
      a b c d = Frame.node(e, tl, Image.open(oa)) Frame.node(e, tr, Image.open(ob))
        Frame.node(e, bl, Image.open(oc)) Frame.node(e, br, Image.open(od))
      Qua{a, b, c, d}
    case _ _:
      Pix{0}

def Frame.show(+cells: Cells, old: Image) -> Image & Cells:
  (Frame.node!(5n, cells, Image.open(old)), cells)
```

`Cell.fork(+k, +cs, ..)` has the same shape over the cell's list and calls `Tile.go` at `0n` . The tile: one flat loop reads each entry once for a block of `Hit{w, c, i}` , then frees its list after the last read.

```
def Blk.go(ts: Tris, +i: U32, +h0: Hit, +h1: Hit, +h2: Hit, +h3: Hit) -> Image:
  match ts:
    case TNil{}:
      Blk.four(h0, h1, h2, h3)
    case TCons{t, n}:
      Blk.go(n, (i + 1 : U32), Hit.step(t, i, h0), Hit.step(t, i, h1),
        Hit.step(t, i, h2), Hit.step(t, i, h3))

def Tile.go(cs: Tris, +x: U32) -> Image:
  +h = {Hit{0.0, 0, 0} : Hit}
  +ts = Tris.filter(cs, x, TNil{})
  Tile.fin(Blk.go(ts, 1, h, h, h, h), ts)
```

`Blk.four` answers `Pix{k0}` when the four hits are one triangle, else a `Qua` of four `Pix`; `Tile.fin` walks the list to `TNil` and returns the image. Transparency: the filter fills a second list (`Filt{+n, ts, bs}`) and `Blend.go` walks it in order over the blocks after the opaque loop. Text and HUD: strokes as opaque quads nearer than the world (`Text.put`, `Quad.put` at inverse depth 3000), layered by depth, order-free. The host: one parallel let per def, the last scene freed as a branch by `Cells.drop(4n)`: four forked levels in `Frame.node`'s shape, then `Tris.free` loops.

```
def Scene.build(+t: F32, used: Cells) -> Cells:
 a b c = Hero.build(t) Stage.build(t) Cells.drop(4n, used)
 Cells.m4(a, b, c, CNil{})
```

A turn keeps the last image and scene as one pair: the build takes the old scene, the bang the old image. An `IO.now()` step sits between them: a bang that follows the host's forks in one evaluation runs on the CPU pool. `Window.frame(window, image)` shows the image.

```
def Turn.go(+t: F32, last: Image & Cells) -> IO(Image & Cells):
 (old, used) = last
 do IO<Image & Cells>:
  cells : Cells = Scene.build(t, used)
  now : Nat <- IO.now()
  IO.pure(Image & Cells, Frame.show(cells, old))
```

ANY SCENE

The pixel never sees the scene. The fork tree opens each node once on the way down and hands each tile one short flat list; the pixel loop marches that list. For triangles, voxels, distance fields and sprites alike, the host walks the world once per frame into per-tile candidates (the primitives, columns or spans whose screen bounds meet the tile); the device reads only those. A field with no scene data (a procedural distance field) is the best case: one flat loop per pixel on a U32 step counter. A per-ray tree walk fails three ways: a fold over a tree is a non-tail recursion (a frame per node), the lanes of a SIMD group diverge, and a `+` tree pays an atomic per node per pixel. A reported case: a voxel world as a `+Data` quadtree of 32x32 columns, read per ray: 176 ms a frame, against 24 ms for the same picture computed per pixel with no scene data (the reporter's machine). The fix: keep the editable map on the host, and hand each tile a flat list of the column words its rays cross (a 32-high column is one U32 of bits) each frame. An `Array` has one owner, so it cannot go down a fork tree: build lists.

DATA

- Records of scalars ride in registers through every call and fork; a boxed record shared by every vertex is an atomic count per use. `Tri` is 24 flat words (barycentric, depth and colour planes, mode, alpha, box): one F32 per 64-bit word, ~200 B loaded per entry per walk.
- A wide record round-trips the stack: a 52-word record per list entry made the draw 6.3 -> 12.1 ms; a 22-word camera on every mesh call, +2-4% build. Build wide data where used (`Hook.cam(F)`).
- A def or continuation takes at most 255 words ("an arity over 255"): four 15-field records in one parallel let fail to compile. Wrap a big header in a constructor of the tree it heads.
- Use the typed `pick(c, a: F32, b: F32)` and `word(c, a: U32, b: U32)`; the generic `Bool.pick` boxes its words.
- Tables are `word` chains and Bool-column matches. A `match` on U32 literals is a bit-by-bit tree with the default arm copied per branch: +21% host C, +6.4% `.gpu`, 1-4% slower.
- Pass a constant as a `~` template argument (a def like `Screen.w()`), not as a parameter: a parameter rides in every task.
- Tessellate in screen space: ~20-px patches; a circle gets `max(4, ceil(pi sqrt(r/2)))` segments. Slivers and sub-pixel triangles pass every box test and pile 100-400 entries on a tile (42 ms frames); exact triangle-vs-cell overlap holds the worst tile at 80-110. Length past that is not the cost: a cap of 16/32/64 entries per tile changed nothing.

OWNERSHIP

The compiler decides borrows; `+` does not. The checker needs `+` for a second use. The compiler borrows a boxed parameter (not an `Array`) that the def only matches or passes to a borrower: every read is `term_peek`, no count. It owns one that the def returns, stores in a constructor, or passes to an owner. An owned use of a value that is used later shares it: a count on the value and, since sealing is per type, on every node of that type, silently. So `+ts` above borrows: `Blk.go` only matches, `Tile.fin` consumes last. So the scene is host-owned, borrowed by the bang, and returned (`Frame.show`). Read through counts, the frame costs 8.0 -> 16.5 ms (draw 6.0 -> 14.1).

- Two consumers of one list share it: 45 ms frames and a 448 s Metal compile. A `+` where one use would do heats its type (symreg 2.83 -> 3.30 s).

- A def that returns its argument shares it: `Bool.pick(Cloth, c, a, b)` on a tree gave 14 keeps and a hot type. Match the selector and recurse into one field (`Cells.put` has eight arms for this). Refolding a matched `+` record at a call is an owned argument (8-16 keeps): pass the fields.
- The last reader of a borrowed list needs a later use, else it becomes the owner: `Tile.fin` walks the lists after the pixels read them (as dead parameters the runtime's sink is 0.55 ms slower).
- A bang inside a `do` continuation flips its callee to owner (the tree is a captured local): 2x reads. Bang in a pure def that returns the pair.
- Drop the last image inside the fork tree (a dead `Four` per tile) and the last scene on the host by a walk (35% faster than the runtime sink); never inside a bang: 12.9 ms. `_ = x` drops a value on the spot.
- Never let the device own the scene: draw 6.2 -> 12.0 ms.

Verify in the emitted C (`bend x.bend -o x.c`): in a def's body, `term_peek` is a borrow; `term_keep` , `ctr_take` and `rfc_seal` are counts. A new keep in a device def is a regression.

COST MODEL

- Mesh, project, light and cull on the host: 0.6-1.6 ms in parallel lets; on the GPU 14-33 ms (a P-core is ~75x a lane on serial work).
- Leaves are flat loops over cons lists; joins are constructors. Unroll the squares: `Tile.b16` = 4 x `Tile.b8` = 4 x `Tile.b4` ; folded into one recursive def they made the draw 5.5 -> 12.6 ms.
- Walk the list once per 4x4 square, not per pixel or per 2x2 block: 6.6 -> 5.6 ms. Anything that depends on x alone, once per column pair: sky per pixel 3.8 ms, per block 1.6, per column pair ~1.0 less.
- The draw is throughput-bound, not tail-bound: splitting heavy tiles +0.03 ms, 32-px cells +0.85. A per-lane early-out saves nothing while a SIMD sibling runs the body.
- Drops: `Image.drop` forked at every `Qua` (~100 ops of task machinery per node), 2.6 ms; base's `Image.free(7n)` , seven levels then a sink, 1.6 ms; inside the draw's tree, 0; a whole image sunk on one lane, 19.8.
- Host: one parallel let per def (a second costs an iteration: build 2.32 -> 1.93 ms); every extra branch or level ~0.1 ms; a 0.3 ms job loses to the pool wake-up; a heavy branch serializes a 16-ring chunk (1.2-1.6 ms): split it.
- CUDA: a scene built by the host and read by the device faults each page over PCIe, both ways: 17-19 FPS on an RTX 4090 (25 ms `gpu_pass` + ~21 ms host turn) for the frame the M4 draws in 8. Keep a whole frame on one side of the bus.

MEASURING

Probe as the game does (`SLASH_PROBE=1200`): scripted frames headless, each stage's microseconds as medians after warm-ups. Single-run medians vary 0.3-0.5 ms across days, and the second binary of a GPU pair runs ~7% faster: compare by 3-4 interleaved 1200-frame pairs in both orders, plus an A-vs-A pair. Check frames pixel-identical (`SLASH_DUMP=T` prints one as a tree) before trusting a gain. Count the keeps in the C and the `.gpu` bytes (+6% was a real 1-4% slowdown). Probe without audio: it wakes `audiomxd` at 70-100% of a core.

DO NOT

1. Read the scene through counts: 2x per read, 8.0 -> 16.5 ms.
2. Fold per level, or bang twice a frame: 600 ms against 8.
3. Recurse non-tail in the bang, or fold a tree per pixel: draw 5.5 -> 12.6.
4. Mesh on the GPU: 14-33 ms against 0.6-1.6.
5. Fork per pixel or per `Qua` : drop 2.6 vs 1.6 ms.
6. Leave slivers and sub-pixel triangles: 42 ms frames.
7. Wide records through frames and joins: 2x per entry.
8. `match` on U32 literals: +21% C, +6.4% `.gpu` .
9. Generic `Bool.pick` on words or trees: boxed words, hot types.
10. Free the scene in a bang (12.9 ms) or the image on the host (milliseconds).
11. Trust one run, or the CPU pool (the 600 ms frame took 11 ms there), as a device measurement.

REFERENCE

`demos/app_slash_boss_3d/bend3d.bend` is the library; its comments are these rules at their sites. `main.bend` beside it is the app, with `Play.loop` and the probes.

FILE: 02-guides-cli/shaders-summary.md

Source: `bend2-knowledge/02-guides-cli/shaders-summary.md`

Shaders guide summary (`bend guide shaders`)

Full: `raw/SHADERS-CLI-DUMP.md` . Reference: `demos/app_slash_boss_3d/bend3d.bend` (1920×1200 @120 FPS, 8 ms/frame on 10-core M4 mini).

MACHINE

- `!` ships one call to GPU cube of 16384 lanes (128×128). Parallel let = 1 task/call + join.
- Kernel grows fork tree BFS until frontier fills cube, then each lane runs to end (~0.4–0.5 ms + slowest lane). Re-forking continuation = next iteration.
- **Flat def**: no parallel let/bang/closure-apply/non-flat-call/non-tail-self-call → one native tail loop (`spin_N` in C). Non-tail self call = segmented continuation (frame per call).
- Unified heap (Apple) / managed+paged (CUDA). No GPU or `--gpu off` → bangs on CPU pool. Divergent work (search, per-ray tree walk) faster on CPU.
- Aim 4^7 leaves/bang (one/lane); sweep one fork level either side on device.

FRAME PATTERN

1. Host: scene → quadtree of 64-px cells, one flat triangle list per cell (~1.6 ms).
2. One bang: fork 4/level ×5 to cells, ×2 over lists → 16-px tiles (4^7=16384). Tile filters list, walks flat loops (~6 ms).
3. Last frame rides down same tree as `Four` quadrants (runtime sinks unused); drop 0 ms.
4. Host frees last scene as branch of next build (~0.35 ms).

```
 2048 root  Frame.node k=5  fork4 L1-5 → 64-px cells (32×32)
cell       Cell.fork k=2   fork4 L6-7 → 16-px tiles (16384)
tile       Tile.b16=4×b8=4×b4  straight-line, no fork
4-px       Blk.go  one flat loop, four Blk → 2-px Blk.four → Pix
```

Off-frame/empty answers one `Pix` before fork (`Frame.sky`). Fork-per-pixel drowns in scheduling; fork-per-cell idles lanes.

CODE RULES

- One list per cell (not pixel); shared-by-pixels list = contended atomics.
- `Frame.show` bangs once, returns tree beside image (host keeps ownership). `Image.open(old)→Four` ; tile gets `old: Four` , ignores it.
- Tile: one flat loop reads each entry once for `Hit{w,c,i}` block, frees list after last read.

Read full dump before writing graphical/parallel apps.

FILE: 03-hvm-inet-theory/bend-to-hvm-compilation.md

Source: `bend2-knowledge/03-hvm-inet-theory/bend-to-hvm-compilation.md`

Bend → HVM compilation intuition (Bend1/HVM2 era)

Source: `docs.rs/crate/bend-lang compilation-and-readback.md` , `dups-and-sups.md` , `syntax.md` (Bend1 docs; concepts carry, syntax doesn't).

POLARIZED READING

Same CON/DUP node = different term by entry port: App `--+` vs Lam `++-` ; Pair `+--` vs elim `-++` ; Dup `-++` vs Sup `+--` ; ERA `+` ; NUM `+` ; SWI `--+` + CON linking branches; OPR `--+` + NUM opcode; REF `+.` `λx.x → CON` ; `((λx.x)(λx.x)) → CON (app)` ; `let {a b}=x → DUP` ; `{a b} → DUP` .

PASSES

`encode_adt, desugar_open, encode_builtins, desugar_match_def, fix_match_terms, lift_local_defs, desugar_bend/fold/with_blocks, make_var_names_unique, desugar_use, linearize_matches(+with), type_check, encode_matches (Scott etc.), linearize_vars (dup/era insert, inline single-use lets), float_combinators, prune, merge_definitions, expand_main, book_to_hvm, eta, check_cycles, inline_hvm_book, prune_hvm_book, check_net_sizes, add_recursive_priority`

AFFINE / NO-GC

`linearize_vars` makes every var used exactly once: 0 uses → `*` ERA (frees); >1 → explicit `dup`. Runtime: single owner, consume frees immediately (Rust-like). No mark-sweep; footprint = live net. Recursive fields need `~` for `fold`.

BEND2 → BENDRT MAPPING

- `a b = f(x) g(y)` = balanced fork → join continuation; scheduler places children on ring grid, no stealing.
- `f!(x)` = GPU placement marker only.
- Sequential result-needed-later → explicit continuation; tail call → jump; lambdas → lifted defs; erased/proof args → no code.
- `Type` affine vs `Data` counted-share/borrow; array sharing across threads experimental `@unsafe`.

FILE: 03-hvm-inet-theory/hvm-lineage.md

Source: `bend2-knowledge/03-hvm-inet-theory/hvm-lineage.md`

HVM lineage: HVM1 → HVM2 → HVM3/HVM4 → BendRT

- **HVM1 (2022)** https://github.com/HigherOrderCO/HVM1 (`https://github.com/HigherOrderCO/HVM1`) — Rust proto, lazy, non-GC, parallel, beta-optimal higher-order fragment. Work-stealing redex queue; sync only on dup locks + atomic substitution. Demos: bubble (sequential), radix (2.5× GHC on 8 threads), λ-mul (exponential). Guide `guide/HOW.md`.
- **HVM2 (2024)** https://github.com/HigherOrderCO/HVM2 (`https://github.com/HigherOrderCO/HVM2`) — simpler/faster/more correct IR; `hvm run/run-c/run-cu/gen-c/gen-cu`; C11 + CUDA 12, no Windows. Paper https://paper.higherorderco.com (`https://paper.higherorderco.com`) (~400 MIPS 1-thread M3 Max, 5200 MIPS 16-thread, 74000 MIPS RTX4090). 64-bit words (2×32-bit ports; NUM 29-bit unboxed, REF vaddr, VAR name, ERA 0).
- **HVM3** https://github.com/HigherOrderCO/HVM3 (`https://github.com/HigherOrderCO/HVM3`) — Haskell Interaction Calculus impl (`HVM.md/IC.md/INTERS.md/MODES.md`), Bend compile target. WIP.
- **HVM4** https://github.com/HigherOrderCO/HVM4 (`https://github.com/HigherOrderCO/HVM4`) — C single-file IC runtime (`src/hvm.c`), `-s/-D/-C` flags. Experimental.
- **Bend1** https://github.com/HigherOrderCO/Bend1 (`https://github.com/HigherOrderCO/Bend1`) — HVM2-powered high-level lang (`bend run-rs/run-c/run-cu/gen-c/gen-cu`), 16 examples (bitonic/bubble/fib/radix…), NVIDIA-only, 32-bit/4GB. Superseded.
- **Bend2/BendRT** https://github.com/bendlang/bend (`https://github.com/bendlang/bend`) — new language, **no interaction nets**; flat evaluator, one C file → clang/Metal/CUDA/JS, one heap, no GC, affine dependent types. Papers `paper/BendTT.pdf + paper/BendRT.pdf`; `bend2/bend.lean` formalization (lags `bend.ts`).
- **hvm-64** https://github.com/HigherOrderCO-archive/hvm-64 (`https://github.com/HigherOrderCO-archive/hvm-64`) — 64-bit tagging experiment, archived.

> Bend2 README: Bend 1 programs and HVM do not carry over.

FILE: 03-hvm-inet-theory/interaction-nets-primer.md

Source: `bend2-knowledge/03-hvm-inet-theory/interaction-nets-primer.md`

Interaction nets primer

Sources: https://en.wikipedia.org/wiki/Interaction_nets (`https://en.wikipedia.org/wiki/Interaction_nets`), Lafont POPL 1990 https://dl.acm.org/doi/pdf/10.1145/96709.96718 (`https://dl.acm.org/doi/pdf/10.1145/96709.96718`)

- Net = agents + wires. Agent type `α ∈ Σ`, arity `ar(α)=n` : **1 principal + n aux ports**.
- Port discipline: each port ≤1 edge; each edge exactly 2 ports. Unconnected = **free ports = interface**.
- **Tree**: edge `x`, or agent with free principal (root) + aux ports wired to subtree roots.
- Textual (Fernandez-Mackie): `< t1..tm | v1=w1, .. >`; each name exactly twice; α-conversion + substitution.
- **Active pair** = principal–principal connection. **Only active pairs rewrite** → parallelism source.
- Rules: `interaction (α(..)=β(..) + α⋈β` → new equations) `+ indirection (x=u` substitute).
- Properties: **locality** (only active pairs), **linearity** (constant-time rules), **strong confluence / one-step diamond** (any order → same result, no sync needed).
- Non-determinism needs `amb` (two principal ports) extension.

FILE: 03-hvm-inet-theory/optimal-reduction-oracle.md

Source: `bend2-knowledge/03-hvm-inet-theory/optimal-reduction-oracle.md`

Optimal reduction (Lévy / Lamping / Gonthier) + why HVM sidesteps the oracle

- **Lévy (1978/80)** http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf): optimal = share redex *families*, contract maximal families incl. leftmost-outermost. Needs sharing closures, not just subexpressions.
- **Lamping (1990)** https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711): sharing graphs (fan=dup, brackets/croissants=oracles) to manage share/unshare.
- **Gonthier-Abadi-Lévy (1992)**: linear-logic cleanup; later work (Asperti-Guerrini book; Asperti-Chroboczek 1997) showed oracle bookkeeping can dominate (brackets accumulate exponentially).
- **HVM dodge**: core is **affine λ-calculus** (like Rust) — vars ≤ once, copy only via explicit linear `dup`, erase via `era`. No thunks/GC; lazy incremental layer-by-layer clone. Hard case lambdas (`let f=λx.(2+x) in [f,f]`) solved by DUP/CON commute+annihilate.
- HVM1 explicitly bookkeeping-free up to Asperti-Guerrini p.40: fast light-lambdas, no full λ (`λx.(x x)` unsupported; IC≠LC on exotic terms). Plan: full/light split + EAL inference. Known quadratic clone case (issue #60) → polarization / unordered superpositions research (Taelin gists, listed not verified).
- Payoff without full optimality: runtime fusion/deforestation (`map f . map g → map (f.g)` dynamically), λ-encoded arithmetic exponentially faster than GHC in demos.

Taelin pedagogy: https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6) (postpone + cache + incremental copy; superposition = imaginary-numbers-like extension). Philosophy: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b)

FILE: 03-hvm-inet-theory/symmetric-combinators-dup-sup-era-con.md

Source: `bend2-knowledge/03-hvm-inet-theory/symmetric-combinators-dup-sup-era-con.md`

Symmetric combinators: ERA / DUP / SUP / CON

Sources: Lafont 1997 https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432), HVM1 https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1), Interaction-Calculus https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus), HVM2 https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)

LAFONT Σ = { ε, δ, γ }

Agent	Arity	Role
ε ERA `(*)`	0	eraser
δ DUP `({})`	2	duplicator
γ CON `(())`	2	constructor (App/Lam side)

- **Erase**: `ε ⋈ α[ε,..,ε]` — ε meets any α → ε clones onto each aux, principals deleted.
- **Commute** (`δ` vs `α≠δ`): `δ[α(x..), α(y..)] ⋈ α[δ(x,y),..]` — δ copies α, pushes copies inside.
- **Annihilate** (same meets same): `δ⋈δ` → straight wires; `γ⋈γ` → straight (swap per presentation).
- Universality: 3 symbols, 6 rules simulate any interaction-net system. Symmetric polarity split → 6 combinators, 9 rules.

HVM2'S 7 NODES + WIRES

Nullary: `*` ERA, `@` REF (top-level fn pointer), `NUM` unboxed. Binary: `( )` CON, `{ }` DUP, `$( )` OPR, `?( )` SWI/MAT. Vars-as-wires; aux-to-main implicit by nesting, aux-to-aux explicit paired names.

Ten interactions: `LINK` (var wiring), `CALL/VOID` (REF expand/delete), `ERAS`, `ANNI` (same family), `COMM` (different families → 4 nodes + cross wiring), `OPER 1/2`, `SWIT 1/2`, `COPY` (NUM/REF fast path; REF-with-DUPs falls back to CALL).

DUP VS SUP (SAME NODE, POLARITY DECIDES)

```
sup = {3 7}                      # build superposition
(mul 2 {5 7}) -> {10 14}         # SUP distributes: F{A B} -> {F A, F B}
let {x1 x2} = {1 2}              # DUP consumes SUP like pair ctor/dtor
ch2 = λf λx let {f1 f2} = f; (f1 (f2 x))
```

- `let {a b} = x` = DUP (consume, `-++`); `{a b}` = SUP (build, `+--`). Context disambiguates.
- Restriction: do not duplicate a duplicator (e.g. `map xs (λx. + x x)` dup of `f` that itself dups) — dup–dup destructively interferes (well-defined at HVM level, wrong at λ-level; may trap).

IC RULES (TAELIN INTERACTION-CALCULUS)

`VAR, ERA(*), LAM(λ), APP, SUP(&L{a,b}), DUP(!&L{x,y}=val; bod)`. Six core: APP-ERA/LAM/SUP, DUP-ERA/LAM/SUP (annihilate if labels equal, nested-dup commute if different) + collapsing (ERA-LAM/APP, SUP-LAM/APP, DUP-VAR/APP) + permutations (APP-DUP, DUP-DUP) + labeled lambdas. IC = λ ∪ combinators; labels upgrade projection to conditional annihilate/commute → Turing-complete but `λx.(x x)` inexpressible.

FILE: 04-runtime-bendrt-hvm5/bendrt-architecture.md

Source: `bend2-knowledge/04-runtime-bendrt-hvm5/bendrt-architecture.md`

BendRT architecture (Bend2 runtime)

Papers: https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf `(https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf)` (runtime), https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf `(https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf)` (theory). Code: `bend2/comp.ts` (compiler + C/Metal/CUDA/JS emitters), `bend2/bend.ts` (language, human-written, do not edit), `bend2/main.ts` (CLI + Bun/Node `.bend` loader).

- **One C file = CPU program + GPU kernel.** clang → CPU; Metal (Apple) / CUDA (NVIDIA) → same file for GPU.
- **Term = 64-bit word**: small inline, else pointer into single heap shared by all cores + GPU. No tracing GC (affine `match` frees; only `+` refcounted). No C stack: each def = flat state-machine segment; call = jump; parallel call = join task + per-call tasks.
- **Theory**: one universe `Type : Type`, no positivity; live/dead checking wall (live runs, must terminate; dead = types/erased/equations, may loop/inhabit `Empty`; dead never counts as live evidence). Mechanized in `bend2/bend.lean` (lags `bend.ts`).
- Targets: C / Metal / CUDA / JS (Lua/Luau/Python planned). One GPU + one event loop currently.

FILE: 04-runtime-bendrt-hvm5/ffi-determinism.md

Source: `bend2-knowledge/04-runtime-bendrt-hvm5/ffi-determinism.md`

FFI + determinism notes

FOREIGN EFFECTS (S: GUIDE IO + EFFECTS.MD)

- `def E(..) -> IO(B): import "./e.c" + import "./e.js"` ; host fn lowercase dots→underscores.
- Only event loop runs them; proofs/termination/GPU never touch host.
- JS import: `import Game from "./game.bend"` with `bend2/main.ts` preloaded; ctors `{$:…}` , `Nat` BigInt; `Array` no-copy.

DETERMINISM RULE FOR CONTRACTS (D ON S)

1. Consensus `transition` sees only `(state, msg, height/hash/caller)` — no RNG/time/net/foreign C/JS inside.
2. Metered + bounded, revert-on-exceed (Kindelia mana/size).
3. Canonical encodings for states/txs/receipts.
4. Rules immutable; upgrades = new hashes + registry pointer.
5. Old in-flight frames finish old rules; new calls use new rules.

See `07-smart-contracts-dynamic/determinism-requirements.md` .

FILE: 04-runtime-bendrt-hvm5/hvm5-mini-reference.md

Source: `bend2-knowledge/04-runtime-bendrt-hvm5/hvm5-mini-reference.md`

HVM5-mini reference (`demos/pure_hvm5_mini`)

Upstream: https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini (https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini) (`main.bend` , `LAWS.bend` , `PROOF.bend`).

- The **only** `HVM5` artifact found in the live repo. Minimal **pure** runtime model written in Bend itself — use as executable spec sketch, not a release.
- Pattern follows every demo: `LAWS.bend` states claims, `PROOF.bend` proves, `bend PROOF.bend` gates.
- For spec work: read its three files line-by-line, then map each def to `08-roadmap-hvm5-spec/spec-todo.md` (API, domain types, metering, fork/upgrade hooks). No standalone HVM5 repo/paper was verified — do not cite one.

FILE: 04-runtime-bendrt-hvm5/memory-64bit-heap-no-gc.md

Source: `bend2-knowledge/04-runtime-bendrt-hvm5/memory-64bit-heap-no-gc.md`

Memory: 64-bit words, unified heap, no GC

HVM2 WORD

Every port 32 bits; binary node stores 2 ports (main implicit by nesting) → 64-bit word → atomic ops. Tag decides reading: heap addr (CON/DUP/OPR/SWI), vfn addr (REF), unboxed 29-bit NUM, var name (VAR), 0 (ERA). NUM/ERA/VAR inline; others indirect; REF fast-copy unless contains DUPs.

IC32 variant (Interaction-Calculus repo): 1-bit sub + 5-bit tag + 26-bit val; tags VAR/LAM/APP/ERA/NUM/SUC/SWI/TMP/SP0-7/CX0-7/CY0-7; no explicit DUP term — dups float on heap via CX/CY vars.

BENDRT HEAP

- One heap shared CPU+GPU; unified (Apple zero-copy) / managed-paged (CUDA).
- No tracing GC: affinity → single owner; `match` frees consumed node; fork transfers args; arrays affine (write in place, read returns array+elem, clone explicit); reusable `Data` only where compiler inserted counted share/borrow.
- Types/erased/proofs → zero runtime values.

CONSEQUENCES FOR SPEC

- Meter heap growth (`bits`) separately from steps (`mana`) — cf. Kindelia model in `07-smart-contracts-dynamic/` .
- Keep `transition` state a `Data` tree; migration functions total + law-checked.

FILE: 04-runtime-bendrt-hvm5/schedulers-cpu-gpu.md

Source: `bend2-knowledge/04-runtime-bendrt-hvm5/schedulers-cpu-gpu.md`

Schedulers: CPU threads + GPU

HVM2 (HISTORICAL)

Active pairs = work items; wires atomics; interactions light C procs / CUDA kernels, lock-free. Sync only: DUP traversal lock + atomic substitution. CPU work-stealing pool; GPU thousands of threads on active-pair list. Any order valid (confluence). Divide-and-conquer exposes parallelism; chains don't.

BENDRT (CURRENT)

- Fixed grid of rings; work grows along rows, drains along columns; task assigned once. **No work stealing** — programmer promises balanced forks (split 1M as halves, not 1 vs 999999).
- Calls → case trees in one worklist fn: tail → jump; sequential-continuation → explicit continuation; parallel fork → join. No C call stack per Bend call; pending work explicit → same binary CPU/GPU.
- `!` marks GPU subtree (Metal/CUDA if `--gpu`, else CPU threads). Nested parallel calls stay on device. One GPU/program.
- Host effects on event loop; waiter yields; `IO.fork` coordinates concurrency, not GPU kernels; JS strictly sequential.
- Bitonic sorter (Bend1 era): 12.15s 1-thread M3 Max → 0.96s 16-thread → 0.21s RTX4090. BendRT numbers: quote with revision/date.

FILE: 05-ecosystem/bendlang-bend-repo.md

Source: `bend2-knowledge/05-ecosystem/bendlang-bend-repo.md`

bendlang/bend (live Bend2 repo)

Source: https://github.com/bendlang/bend (https://github.com/bendlang/bend) — "Bend 2: a fast language that blocks AI mistakes via proof." 22.9k★, Apache-2.0, TS compiler + C/Metal/CUDA/JS. Install `curl -fsSL https://bend-lang.com/install.sh | sh`.

```
bend2/  base.bend bend.lean bend.ts comp.ts main.ts docs/ effs/ pack/
demos/  16 demos (LAWS+PROOF+main pattern)
guide/  GUIDE.md EFFECTS.md SHADERS.md
paper/  BendTT.pdf BendRT.pdf
bench/  runtime/ checker/  evals/ tests/ gates/ tools/bend-fmt-lsp/
AGENTS.md CHANGELOG.md WONTFIX.txt
```

- `bend2/bend.ts` language core (human-written, do not edit); `comp.ts` compiler+runtimes; `main.ts` CLI + Bun/Node loader; `pack/` hash packages (no names/versions/accounts/search yet).
- Agent contract (`AGENTS.md`): run `bend guide`, use `LAWS.bend`, run `bend PROOF.bend` before commit, parallelize.
- Community: https://bend-lang.com (https://bend-lang.com), https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues), LSP https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp)
- Credits: Victor Taelin, Lorenzo W Battistela, Paulo J Cavalcanti, Nico, Vanessa Ostroski, Vitor Chiarelli Neves, Alex Van de Sande.
- Limitations: Bend 2 new language (Bend1/HVM don't carry); compiler 99% AI-written unaudited; `bend.ts` / `bend.lean` mismatch; clang-only, no Windows, glibc 2.34+ for `Process.run`, JS single-core no gfx/audio.

Same repo is the community `bendlang/bend` reference requested — it IS the official Bend2 repo.

FILE: 05-ecosystem/higherorderco-org.md

Source: `bend2-knowledge/05-ecosystem/higherorderco-org.md`

HigherOrderCO org

Source: https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) — "we are getting to the very core of what makes computers capable of reasoning." ~2.8k followers. Links: https://bend-lang.com (https://bend-lang.com), X @HigherOrderComp , contact@higherorderco.com . People: nicolas-abril , VictorTaelin . Top langs: Haskell, Cuda, C, Rust. 7 repos shown.

> Active Bend2 moved: https://github.com/HigherOrderCO/Bend2 (https://github.com/HigherOrderCO/Bend2) = stub "Bend 2 has moved. Go to https://github.com/bendlang/bend (https://github.com/bendlang/bend)".

Repo	URL	Notes
HVM1	https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1)	2022 Rust proto, guide HOW.md
HVM2	https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)	2024 IR, C/CUDA, paper
HVM3	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3)	Haskell IC, HVM.md/IC.md/INTERS.md/MODES.md , WIP
HVM4	https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)	C single-file, src/hvm.c , docs/
Bend1	https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1)	Superseded; examples/ 16 files, GUIDE.md/FEATURES.md/FAQ.md
Kind	https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind)	Haskell proof checker rewrite
Bend2 stub	https://github.com/HigherOrderCO/Bend2 (https://github.com/HigherOrderCO/Bend2)	Redirect only

No HVM5/Kindelia/awesome/benchmarks repos in org listing — HVM5 only as bendlang/bend/demos/pure_hvm5_mini ; benches live under bendlang/bend/bench/ .

FILE: 05-ecosystem/papers-formalization-community.md

Source: bend2-knowledge/05-ecosystem/papers-formalization-community.md

Papers, formalization, community

- Theory: https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf) — affine dependent type theory (live/dead wall, Type:Type , no positivity).
- Runtime: https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf) — parallel CPU/GPU design + benchmarks.
- HVM2 paper: https://paper.higherorderco.com (https://paper.higherorderco.com) (mirror HigherOrderCO/HVM1/blob/main/paper/HVM2.pdf).
- Formalization: https://github.com/bendlang/bend/blob/main/bend2/bend.lean (https://github.com/bendlang/bend/blob/main/bend2/bend.lean) (lags bend.ts).
- Base: https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend) (= bend base).
- Guides: https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md), EFFECTS.md , SHADERS.md .
- Site/hub: https://bend-lang.com (https://bend-lang.com) (install, charts, lab https://bend-lang.com/#lab (https://bend-lang.com/#lab)), https://hub.bend-lang.com (https://hub.bend-lang.com) (hash packages), terms https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18).

- Lafont: https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718) (1990 nets), https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432) (1997 combinators).
- Optimal: http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf) (Lévy), https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711) (Lamping).
- Community: https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues).

FILE: 05-ecosystem/victor-taelin.md

Source: `bend2-knowledge/05-ecosystem/victor-taelin.md`

Victor Taelin (profile + links)

Source: https://github.com/VictorTaelin (https://github.com/VictorTaelin) — "Functional-programming enthusiast, author of Kind and HVM." Rio de Janeiro, Higher Order Company, 6.5k followers, 261 repos.

CONTACT / COMPANY

- https://higherorderco.com (https://higherorderco.com), taelin@higherorderco.com, https://twitter.com/victortaelin (https://twitter.com/victortaelin), https://instagram.com/victortaelin (https://instagram.com/victortaelin), https://www.reddit.com/user/SrPeixinho/ (https://www.reddit.com/user/SrPeixinho/), Discord https://discord.higherorderco.com (https://discord.higherorderco.com)

PINNED

- https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1) — parallel optimal runtime (11.3k★)
- https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind) — proof language (3.8k★)
- https://github.com/VictorTaelin/WebMonkeys (https://github.com/VictorTaelin/WebMonkeys) — GPU tasks on WebGL/JS (1.4k★)
- https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus) — IC spec (957★): `VAR/ERA/LAM/APP/SUP/DUP` , 6 core rules, global-var O(1) substitution, IC32 32-bit runtime
- https://github.com/VictorTaelin/Cedille-Core (https://github.com/VictorTaelin/Cedille-Core) — minimal proof lang (~1k LOC JS)
- https://github.com/VictorTaelin/UrnaCripto (https://github.com/VictorTaelin/UrnaCripto)

KEY READS (PROFILE README)

- Philosophy: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b) (TM vs λ vs IC; commutation+annihilation constant-step)
- Pedagogy: https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6) (postpone+cache+incremental copy; superposition)
- How: https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md (https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md)
- Talk: https://www.youtube.com/watch?v=sDPuQ-UjhVQ (https://www.youtube.com/watch?v=sDPuQ-UjhVQ)
- Release: https://x.com/VictorTaelin/status/1791213162525524076 (https://x.com/VictorTaelin/status/1791213162525524076); coverage https://www.youtube.com/watch?v=HCOQmKTFzYY (https://www.youtube.com/watch?v=HCOQmKTFzYY), https://www.youtube.com/watch?v=NaytZOiX3fs (https://www.youtube.com/watch?v=NaytZOiX3fs), https://news.ycombinator.com/item?id=40390287 (https://news.ycombinator.com/item?id=40390287)

REPO NOTES

- Kindelia → https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) (+ WHITEPAPER https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md (https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md)): cryptocomputer, no coin, functional opcodes, reversible heaps, mana/bits.
- Kind → https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind) (Haskell rewrite; KindBook examples; conversion-checker gist https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5 (https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5))
- Cedille-Core → erasure + heterogeneous equality roots for Kind → BendTT trajectory.

FILE: 06-examples/README-examples-index.md

Source: `bend2-knowledge/06-examples/README-examples-index.md`

Examples index

Upstream: https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) (16 dirs, each `LAWS.bend` + `PROOF.bend` + `main.bend`), https://github.com/bendlang/bend/tree/main/bench (https://github.com/bendlang/bend/tree/main/bench), legacy https://github.com/HigherOrderCO/Bend1/tree/main/examples (https://github.com/HigherOrderCO/Bend1/tree/main/examples)

START HERE

1. `io_hello_world` — `IO.print("Hello, world!")`; `bend demos/io_hello_world/main.bend`
2. `pure_par_sum` — fork/join template + law `tree_is_seq`; `sum!(16n,0n)` → `2147450880`
3. `proof_insertion_sort` — law-driven sort (`LE as Unit/Empty`, `Sorted.from/count`, `sort_sorted/sort_perm`)
4. `app_win_is_bug_2d` — game-law pattern (`you_cant_win`, `never_on_flag`); play https://bend-lang.com/#lab (https://bend-lang.com/#lab)
5. `app_slash_boss_3d` — flagship 3D + `bend3d.bend` rasterizer; `bend demos/app_slash_boss_3d/main.bend -o slash && ./slash`

Details: `parallel-examples.md`, `proof-examples.md`, `app-examples.md`, `io-systems-examples.md`, `bench-suite.md`, `hello-world-templates.md`

FILE: 06-examples/app-io-examples.md

Source: `bend2-knowledge/06-examples/app-io-examples.md`

App + IO/systems examples

APPS (PURE `IMAGE + APP.RUN`)

- `app_slash_boss_3d` https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d (https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d) — 1v1 3D duel (`bend3d.bend` rasterizer, WASD/IJKLPR/Esc, per-frame scene + sound, CC0 Kenney audio in `media/slash_boss_3d`).
- `app_pong_game_2d` https://github.com/bendlang/bend/tree/main/demos/app_pong_game_2d (https://github.com/bendlang/bend/tree/main/demos/app_pong_game_2d) — 2D Pong (complete `App.run` example cited by guide).
- `app_ray_tracer_3d` — 3D ray tracer app.
- `app_triangle_2d`, `app_win_is_bug_2d` (torus-grid game, web `web/main.js` imports `../main.bend` via `bend2/main.ts`).

IO / SYSTEMS

- `io_hello_world`, `io_http_fetch`, `io_http_server` (`TCP.listen(8080)`, per-conn `IO.spawn`, `@unsafe` accept loop; `bend … -o http_server && ./http_server`), `io_rollback_netcode` (deterministic sim + rollback multiplayer ref), `io_tcp_echos`.
- Effects used: print/env/time/sleep/spawn/channels/files/TCP/UDP. No TLS/HTTP-lib/JSON/regex — `io_http_*` use foreigns.

FILE: 06-examples/bench-suite-templates.md

Source: `bend2-knowledge/06-examples/bench-suite-templates.md`

Bench suite + hello templates

BENCH RUNTIME (17 ENTRIES)

https://github.com/bendlang/bend/tree/main/bench/runtime (https://github.com/bendlang/bend/tree/main/bench/runtime): `_pin_`, `bfs`, `editdist`, `gameoflife`, `hashmap`, `kmeans`, `lexer`, `mandelbrot`, `merkle`, `nbody`, `queens`, `raytrace`, `symreg`, `terrain`, `tree-bitonic`, `tree-matmul`, `tree-radix`. Modern home of nbody/mandelbrot/bitonic/matmul examples. Each bench dir + C/TS/Lean twins; pins `bench/*/_pin_/apple_m4*` for `gates/perf.ts`.

BENCH CHECKER (6)

https://github.com/bendlang/bend/tree/main/bench/checker (https://github.com/bendlang/bend/tree/main/bench/checker): `_pin_`, `compute_1600`, `defs_12800`, `generics_3200`, `proofs_3200`, `trees_400`.

TEMPLATES

```
import Base
def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

```
import Base
def pow2(+n: Nat) -> U32:
  match n:
    case 0n: 1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)
```

```
import Base
def main() -> Array<U32> & U32:
  a = [0 : U32*8n]
  a[5] <- 42
  a[5]
```

FILE: 06-examples/parallel-examples.md

Source: *`bend2-knowledge/06-examples/parallel-examples.md`*

Parallel examples

- `pure_par_sum` https://github.com/bendlang/bend/tree/main/demos/pure_par_sum (https://github.com/bendlang/bend/tree/main/demos/pure_par_sum) — canonical `pow2/sum/seq`, all forks parallel lets, GPU via `!`. Copy as parallel template.
- `pure_par_sort` https://github.com/bendlang/bend/tree/main/demos/pure_par_sort (https://github.com/bendlang/bend/tree/main/demos/pure_par_sort) — parallel bitonic sort on `Tree(d)` (`gen/mix/flow/sort`), `sort!(16n,…)` GPU/CPU, `ascending` checker.
- `pure_hvm5_mini` https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini (https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini) — minimal pure runtime reference.
- Bench parallels https://github.com/bendlang/bend/tree/main/bench/runtime (https://github.com/bendlang/bend/tree/main/bench/runtime): `tree-bitonic`, `tree-matmul`, `tree-radix`, `mandelbrot`, `nbody`, `kmeans`, `gameoflife`, `raytrace`, `merkle`, `bfs`, `terrain`, `queens`, `symreg`, `lexer`, `editdist`, `hashmap`.
- Legacy Bend1 https://github.com/HigherOrderCO/Bend1/tree/main/examples (https://github.com/HigherOrderCO/Bend1/tree/main/examples): `bitonic_sort`, `radix_sort`, `parallel_sum`, `parallel_and`, `fib`, `queue` (HVM2-era syntax, do not copy verbatim to Bend2).
- Eval tasks https://github.com/bendlang/bend/tree/main/evals (https://github.com/bendlang/bend/tree/main/evals): `cake.*`, `easy.*`, `firm.*`, `hard.*`, `hell.*` (incl. `hell.combinator_confluence`, `hell.loop_normalization`).

Pattern: host builds balanced tree/list → one `!` → flat loops on device → return tree beside image/state (host keeps ownership).

FILE: 06-examples/proof-examples.md

Source: `bend2-knowledge/06-examples/proof-examples.md`

Proof examples (law-driven)

- `proof_insertion_sort` https://github.com/bendlang/bend/tree/main/demos/proof_insertion_sort (https://github.com/bendlang/bend/tree/main/demos/proof_insertion_sort) — reference: `LE` as `Unit/Empty`, `dec/insert/sort` with evidence-branching, `Sorted.from/count`, laws `sort_sorted/sort_perm` (permutation via counts).
- `proof_numerics`, `proof_typed_eval` — numerics correctness + typed evaluator (fetch `main/LAWS/PROOF.bend` per demo).
- `app_win_is_bug_2d` https://github.com/bendlang/bend/tree/main/demos/app_win_is_bug_2d (https://github.com/bendlang/bend/tree/main/demos/app_win_is_bug_2d) — game laws (`you_cant_win`: replay any moves never `is_won`; `never_on_flag`) with finite map enumeration (`chk_all`).
- `app_slash_boss_3d` https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d (https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d) — sim laws (pause-freeze, double-P restore, Esc-quit).
- `pure_par_sum` — `tree_is_seq: Par.sum(d,i)==Par.seq(pow2(d),i)` by induction.
- Checker benches https://github.com/bendlang/bend/tree/main/bench/checker (https://github.com/bendlang/bend/tree/main/bench/checker): `_pin_`, `compute_1600`, `defs_12800`, `generics_3200`, `proofs_3200`, `trees_400`.

FILE: 07-smart-contracts-dynamic/content-addressing-hub.md

Source: `bend2-knowledge/07-smart-contracts-dynamic/content-addressing-hub.md`

Content-addressing + hub (S)

- "A package is a directory of files, named by a hash of all of its contents. That hash is its whole identity, and there are no names, versions or accounts." — https://hub.bend-lang.com (https://hub.bend-lang.com)
- `import 0x<hash>/main.bend as P` — fetched + hash-verified (GUIDE Modules; https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)).
- `bend main.bend --publish` → prints that import line. Named: `import name@ver/main.bend` + `bend main.bend --publish name@ver` after `bend login` (release v2.0.26); cache `~/.bend/lib/names`; version never moves.
- Public + permanent (https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18)); `LICENSE` files hashed; no-LICENSE = MIT-0.
- Alias file-local; dots are chars; open law fillable as `def M.name(..)`.
- Hash algorithm NOT verified (hub shows `0x…` like `0x729eecea…` e.g. https://hub.bend-lang.com/0x729eecea86ea5a2cdba3a2856a313bca?f=bolt%2Fconfig.bend (https://hub.bend-lang.com/0x729eecea86ea5a2cdba3a2856a313bca?f=bolt%2Fconfig.bend)) — do not cite one.

Contrast (S): EVM proxy+ `delegatecall` /diamond/migration https://ethereum.org/developers/docs/smart-contracts/upgrading (https://ethereum.org/developers/docs/smart-contracts/upgrading), OpenZeppelin https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts (https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts), TON delayed `setCode` https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades (https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades), Dart hot-reload (methods swap, fields stay, old frames finish) https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md (https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md).

FILE: 07-smart-contracts-dynamic/domain-types-sketch.md

Source: `bend2-knowledge/07-smart-contracts-dynamic/domain-types-sketch.md`

Domain types sketch (D — design notes, not spec)

Inspired by Kindelia `fun/with/run` + IO (S) and Bend2 `type/def/law/IO/do` (S: GUIDE).

```
import Base
type State is Data:
  ...
type Msg is Data:
  ...
type Reject is Data:
  Unauthorized{who: U256}
  BadTransition{reason: String}
  OutOfFuel{}

# Total transition: only place state changes (runtime pins).
def transition(+s: State, m: Msg) -> State & (Unit | Reject):
  ...
def init(genesis: Genesis) -> State: ...
def migrate(old: Old.State) -> State: ...
def view(+s: State, q: Query) -> Answer: ...

law transition_total:
  for s: State
  for m: Msg
  exs s2: State
  {transition(s, m).0 == s2 : State}
```

- Keep `transition` total + terminating (Bend2 termination mandatory except `@unsafe`, S) so metering bounds cost, not halting.
- Auth from envelope (`subj/from/tick/time`, cf. genesis `SUBJ/FROM/TICK/TIME`, S) as erased/affine params — never forgeable inside `Msg`.
- `view` vs `transition` mirrors Kindelia `Call` vs `Save` and Bend2 value- `main` (checker-normalized) vs IO- `main` (compiled).

FILE: 07-smart-contracts-dynamic/fork-upgrade-patterns.md

Source: `bend2-knowledge/07-smart-contracts-dynamic/fork-upgrade-patterns.md`

Fork / upgrade patterns (D on S primitives)

FORK = PIN + OVERRIDE (D)

```
import Base
import 0x<orig-hash>/main.bend as V1   # immutable (S syntax)
def transition(s: State, m: Msg) -> State & (Unit | Reject):
  match m:
    case Patched{..}: <new logic>
    case _: V1.transition(s, m)        # delegate remainder
```

Works because hash import immutable (S) + alias file-local (S) → forking = publishing new package importing old hash + re-exporting new `transition`. No proxy needed for fork case.

UPGRADE VIA INDIRECTION (D, THREE OPTIONS)

- **(a) Registry/dispatch (recommended).** Stable name (namespace or `name@ver` pointer) holds `current_hash`; `dispatch` tail-calls target. Upgrade = registry tx (admin/multisig-gated). Audit = diff of two hashes.
- **(b) Versioned namespaces.** `MyApp.V1.transition`, `MyApp.V2.transition`; clients opt in; registry maps "latest". Mirrors Kindelia `reg` (S).
- **(c) Migration.** V2 empty; explicit `migrate : V1.State -> V2.State`; old frozen. Most auditable, most coordination.

RULES (D)

- Keep `State a Data` tree; `migrate` total + `law migrate_roundtrip` so `bend PROOF.bend` gates upgrade.
- Never emulate `delegatecall` byte-patching: interaction rules global + confluent (https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets)) → upgrades are new rules + pointer swap/redeploy, never live-rule edits. Avoids storage-collision classes by construction.

FILE: 07-smart-contracts-dynamic/hvm5-api-checklist.md

Source: `bend2-knowledge/07-smart-contracts-dynamic/hvm5-api-checklist.md`

HVM5 API checklist (D, Kindelia-mapped) + determinism

- <input disabled="" type="checkbox"> `deploy_ctr/deploy_fun` immutable, namespace-gated, reject dupes/arity>16 (cf. `ctr` / `fun`)
- <input disabled="" type="checkbox"> `register/check_can_deploy` + genesis `Namer` (cf. `reg`)
- <input disabled="" type="checkbox"> `submit(run_expr, sig) -> txid` (cf. signed `run`, `Subj/From`)
- <input disabled="" type="checkbox"> `step(contract, msg, state) -> (new_state, result, receipt)` deterministic (cf. `run`)
- <input disabled="" type="checkbox"> `load/save` with take-vs-clone semantics explicit (`take` moves, `load` copies)
- <input disabled="" type="checkbox"> Oracles `tick/time/meta/stm_hash/get_idx` read-only (cf. `TICK/TIME/META/STH/GIDX`)
- <input disabled="" type="checkbox"> Metering per-rule `mana_table + mana_limit/bits_limit` cumulative; revert (not partial-commit) on exceed
- <input disabled="" type="checkbox"> Query `call_view/get_state/get_rules`; packages `fetch(0x<hash>)` verified, `publish->import-line`, `resolve(name@ver)` cached
- <input disabled="" type="checkbox"> `fork(hash, overrides) -> new_hash` = new package importing old hash
- <input disabled="" type="checkbox"> `propose/approve/execute_upgrade` timelocked + `migrate` hook + `Upgraded(old,new)` event
- <input disabled="" type="checkbox"> Determinism gate: confluence-fixed semantics, canonical serialization, no float/wall-clock/RNG/foreign in `transition`; host effects outside consensus only
- <input disabled="" type="checkbox"> Proof gate: `LAWS.bend + PROOF.bend`; `bend PROOF.bend` success required before publish/upgrade
- <input disabled="" type="checkbox"> Tooling: `node start`, offline `test`, `sign`, `bend base` introspection, `--threads/--gpu` flags

DETERMINISM REQUIREMENTS (D, S-ANCHORED)

1. Consensus purity: `(state, msg, env)` only.
2. Metered, bounded, revert-on-exceed.
3. Canonical encodings.
4. Rule-set immutability; upgrades = new hashes + pointer.
5. Old frames finish old rules; new calls use new rules (Dart hot-reload analogy).

FILE: 07-smart-contracts-dynamic/kindelia-model.md

Source: `bend2-knowledge/07-smart-contracts-dynamic/kindelia-model.md`

Kindelia model (S = sourced)

Sources: https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia), https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md), genesis https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl (https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl), https://kindelia.org (https://kindelia.org)

- Network = decentralized REPL; blocks = files settled via PoW. No native coin, PoW forever, ~1s blocks, ~10k LOC Rust vs 600k+ Go-Ethereum.
- **Statements**: `ctr {Name fields}` (global ctor, arity ≤16), `fun (Name args) {rules} with {init}` (global stateful fn), `run {IO_expr}` (metered eval, signer = subject), `reg Name {owner}` (namespace). Each may carry secp256k1 `sign`.
- **Contracts = stateful functions**: state = any HVM structure (number/list/tree, no U256 serialization); `load` / `save` pin pointers (zero-cost SSTOREs — pay for building state + heap growth, not the pin). `IO.take` *removes* state (must `save` back or emptied); `Load = Take + dup + Save`.
- **IO**: pure description (`Done/Take/Save/Call/Subj/From/Tick/GetIdx/StmHash/Time/Meta/Hax` in genesis).
- **Namespaces**: dotted (`Foo.Bar.cats`), deploy rights vs parent owner, top `Namer` in genesis; undotted deployable by anyone.

- **Costs**: per-rule mana (`APP-LAM 2, APP-SUP 4, OP2-NUM 2, FUN-CTR 2+M, DUP-LAM 4, …`), `mana_limit = 4M×(height+1)`, `bits_limit = 2048×(height+1)` (~8GB/yr), unused accumulates.
- On-chain text = low-level assembly (lam `@x`, app `(f a)`, ctor `{C…}`, call `(F…)`, `#num` u120, `(op a b)` 16 ops, `dup x y = e`, affine). Devs compile Haskell/Kind → it.
- Header: work-in-progress; archived lineage distinct from HVM2/Bend2.

Bend2 analogue (D): contract = module exposing `State/Msg/transition` compiled to rules; chain pins state pointer + code hash; `run` = submit→step→meter→pin-or-revert.

FILE: 08-roadmap-hvm5-spec/references-index.md

Source: `bend2-knowledge/08-roadmap-hvm5-spec/references-index.md`

References index (all fetched 2026-09-26, bend 2.0.25)

CLI (LOCAL DUMPS IN `02-GUIDES-CLI/RAW/`)

`bend guide`, `bend guide shaders`, `bend guide effects`, `bend base`, `bend base --types`, `bend --help`

ECOSYSTEM

- https://github.com/VictorTaelin (https://github.com/VictorTaelin)
- https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) (+ /HVM1 /HVM2 /HVM3 /HVM4 /Bend1 /Kind /Bend2-stub)
- https://github.com/bendlang/bend (https://github.com/bendlang/bend) (+ `/tree/main/bend2`, `/tree/main/demos`, `/tree/main/guide`, `/tree/main/paper`, `/tree/main/bench`, `/tree/main/bench/runtime`, `/tree/main/bench/checker`, `/tree/main/evals`)
- Demos: `/tree/main/demos/pure_hvm5_mini`, `/pure_par_sum`, `/pure_par_sort`, `/proof_insertion_sort`, `/app_slash_boss_3d`, `/app_pong_game_2d`, `/app_ray_tracer_3d`, `/app_win_is_bug_2d`, `/io_http_server`
- https://github.com/HigherOrderCO/Bend1/tree/main/examples (https://github.com/HigherOrderCO/Bend1/tree/main/examples)
- https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus), /WebMonkeys, /Cedille-Core
- https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) (+ `/blob/master/WHITEBOOK.md`, `/blob/master/WHITEPAPER.md`)
- https://github.com/HigherOrderCO-archive/hvm-64 (https://github.com/HigherOrderCO-archive/hvm-64)

DOCS / PAPERS / SITES

- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md), `EFFECTS.md`, `SHADERS.md`
- https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend), `bend2/bend.lean`
- https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf), `paper/BendRT.pdf`
- https://bend-lang.com (https://bend-lang.com), https://hub.bend-lang.com (https://hub.bend-lang.com), https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18)
- https://paper.higherorderco.com (https://paper.higherorderco.com), https://higherorderco.com (https://higherorderco.com)
- Gists: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b), https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6)

THEORY

- https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets)
- https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718), https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432)

- http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf), https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711)

CONTRACTS / UPGRADES

- https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl (https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl), https://kindelia.org (https://kindelia.org)
- https://ethereum.org/developers/docs/smart-contracts/upgrading (https://ethereum.org/developers/docs/smart-contracts/upgrading)
- https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts (https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts)
- https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades (https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades)
- https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md (https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md)

COMMUNITY

https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues), https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp)

FILE: 08-roadmap-hvm5-spec/spec-todo.md

Source: `bend2-knowledge/08-roadmap-hvm5-spec/spec-todo.md`

Spec TODO (toward full HVM5 dev spec)

PHASE 1 — GROUND TRUTH

- <input disabled="" type="checkbox"> Read `demos/pure_hvm5_mini/{main,LAWS,PROOF}.bend` line-by-line; map to API checklist in `07-smart-contracts-dynamic/hvm5-api-checklist.md`.
- <input disabled="" type="checkbox"> Read `paper/BendTT.pdf + paper/BendRT.pdf`; extract normative rules (live/dead, termination, scheduler, memory).
- <input disabled="" type="checkbox"> Read `bend2/comp.ts` emitters (C/Metal/CUDA/JS) for tag layout, task format, `term_peek/term_keep` ownership.
- <input disabled="" type="checkbox"> Resolve `bend.ts` vs `bend.lean` normativity; record version (`bend version`).

PHASE 2 — DOMAIN + API

- <input disabled="" type="checkbox"> Freeze `State/Msg/Reject/Query/Genesis` domain types (`07-smart-contracts-dynamic/domain-types-sketch.md` → real `.bend`).
- <input disabled="" type="checkbox"> Define `transition/init/migrate/view` signatures + totality/termination laws.
- <input disabled="" type="checkbox"> Define node API: deploy/register/submit/step/load/save/oracles/meter/query/fork/upgrade (checklist → OpenAPI/RPC).
- <input disabled="" type="checkbox"> Define canonical serialization + metering table (per-rewrite costs + heap-growth costs).

PHASE 3 — DYNAMIC UPGRADES

- <input disabled="" type="checkbox"> Implement fork template (`import 0x<hash>` + override + re-publish) as runnable demo.
- <input disabled="" type="checkbox"> Implement registry/dispatch + timelocked upgrade + `migrate` + `Upgraded` event.
- <input disabled="" type="checkbox"> Proof-gate CI: `bend PROOF.bend` must pass before publish/upgrade accepted.
- <input disabled="" type="checkbox"> In-flight semantics test (old frames finish old rules).

PHASE 4 — HARDEN

- <input disabled="" type="checkbox"> Determinism suite (no RNG/time/net/foreign/float in `transition`; revert-on-exceed; canonical vectors).

- <input disabled="" type="checkbox"> Perf pins (`bench/*/_pin_` , `gates/perf.ts`) for target hardware.
- <input disabled="" type="checkbox"> Security review: namespace auth, handle unforgability, meter bypass, hub hash verification.

FILE: 08-roadmap-hvm5-spec/what-we-know-vs-gaps.md

Source: `bend2-knowledge/08-roadmap-hvm5-spec/what-we-know-vs-gaps.md`

Knowns vs gaps (for HVM5 spec work)

KNOWNS (S)

- Bend2 syntax/semantics/CLI/Base fully captured from `bend guide + bend base` (bend 2.0.25) in `02-guides-cli/` .
- BendRT model: one C file, 64-bit words, unified heap, no GC, flat state machine, fork-join `+ !` GPU (papers BendTT/BendRT, `bend2/`).
- INET theory: nets, combinators, annihilate/commute/erase, oracle problem + affine dodge, HVM1→BendRT lineage.
- Ecosystem map: Taelin profile, HigherOrderCO 7 repos, live `bendlang/bend` layout, 16 demos, benches, evals.
- Contracts ancestor: Kindelia statements/costs/namespaces/IO; Bend hub hash imports + publish flows.

GAPS (MUST CLOSE BEFORE SPEC)

1. No verified standalone HVM5 repo/paper/spec — only `demos/pure_hvm5_mini` . Read its 3 files first.
2. No verified Bend hash-function spec (only `0x<hash>` syntax + hub behavior).
3. BendRT internals (ring-grid scheduler params, GPU lane config, tag bit-fields beyond paper abstract, CUDA/Metal kernels) need `paper/BendRT.pdf + bend2/comp.ts` code reading.
4. `bend.ts` vs `bend.lean` mismatch — which is normative for proofs?
5. Metering table for BendRT (Kindelia mana table is HVM-era, not BendRT).
6. Hub APIs (`BEND_HUB` , resolve/fetch/publish wire format) need network inspection.
7. Kindelia Block #5 TODO; no Kindelia→Bend2 migration doc exists.

F.56 · `bend-runtime/knowledge/ALL-COMBINED-MINI.md`

Bend2 / HVM5 / INET — ALL COMBINED MINI (<=2000 lines)

Generated 2026-09-26. 60 files, raw dumps truncated to 25 lines, long files condensed. Full text in bend2-knowledge/ + ALL-COMBINED.md.

CONTENTS

1. `README.md`
2. `00-start-here/glossary.md`
3. `00-start-here/how-to-use-this-knowledge-base.md`
4. `00-start-here/overview.md`
5. `01-bend-language/apps-modules-packages.md`
6. `01-bend-language/arrays.md`
7. `01-bend-language/functions-closures-templates-recursion.md`
8. `01-bend-language/io-concurrency-monads.md`
9. `01-bend-language/laws-proofs.md`
10. `01-bend-language/parallelism-fork-join-gpu.md`
11. `01-bend-language/syntax-reference.md`
12. `01-bend-language/types-quantities-kinds.md`
13. `02-guides-cli/base-library.md`
14. `02-guides-cli/cli-tooling.md`
15. `02-guides-cli/effects-summary.md`
16. `02-guides-cli/guides-index.md`

17. 02-guides-cli/raw/BASE-FULL-DUMP.md
18. 02-guides-cli/raw/BASE-TYPES-DUMP.md
19. 02-guides-cli/raw/EFFECTS-CLI-DUMP.md
20. 02-guides-cli/raw/GUIDE-CLI-DUMP.md
21. 02-guides-cli/raw/SHADERS-CLI-DUMP.md
22. 02-guides-cli/shaders-summary.md
23. 03-hvm-inet-theory/bend-to-hvm-compilation.md
24. 03-hvm-inet-theory/hvm-lineage.md
25. 03-hvm-inet-theory/interaction-nets-primer.md
26. 03-hvm-inet-theory/optimal-reduction-oracle.md
27. 03-hvm-inet-theory/symmetric-combinators-dup-sup-era-con.md
28. 04-runtime-bendrt-hvm5/bendrt-architecture.md
29. 04-runtime-bendrt-hvm5/ffi-determinism.md
30. 04-runtime-bendrt-hvm5/hvm5-mini-reference.md
31. 04-runtime-bendrt-hvm5/memory-64bit-heap-no-gc.md
32. 04-runtime-bendrt-hvm5/schedulers-cpu-gpu.md
33. 05-ecosystem/bendlang-bend-repo.md
34. 05-ecosystem/higherorderco-org.md
35. 05-ecosystem/papers-formalization-community.md
36. 05-ecosystem/victor-taelin.md
37. 06-examples/README-examples-index.md
38. 06-examples/app-io-examples.md
39. 06-examples/bench-suite-templates.md
40. 06-examples/parallel-examples.md
41. 06-examples/proof-examples.md
42. 07-smart-contracts-dynamic/content-addressing-hub.md
43. 07-smart-contracts-dynamic/domain-types-sketch.md
44. 07-smart-contracts-dynamic/fork-upgrade-patterns.md
45. 07-smart-contracts-dynamic/hvm5-api-checklist.md
46. 07-smart-contracts-dynamic/kindelia-model.md
47. 08-roadmap-hvm5-spec/references-index.md
48. 08-roadmap-hvm5-spec/spec-todo.md
49. 08-roadmap-hvm5-spec/what-we-know-vs-gaps.md
50. 09-hvm5-cluster-runtime/01-concepts.md
51. 09-hvm5-cluster-runtime/02-architecture-layers.md
52. 09-hvm5-cluster-runtime/03-domain-types.md
53. 09-hvm5-cluster-runtime/04-api.md
54. 09-hvm5-cluster-runtime/05-registry-naming.md
55. 09-hvm5-cluster-runtime/06-update-fork-merge.md
56. 09-hvm5-cluster-runtime/07-ocap-wires.md
57. 09-hvm5-cluster-runtime/08-similar-projects.md
58. 09-hvm5-cluster-runtime/09-implementation-plan.md
59. 09-hvm5-cluster-runtime/10-meta-prompt-for-builder-llm.md
60. 09-hvm5-cluster-runtime/README.md

FILE: README.md

Bend2 / HVM5 / INET Knowledge Base

Comprehensive overview of Bend2, HVM5/BendRT, interaction nets, runtime, interfaces, and dynamic smart-contract patterns. Built 2026-09-26 from CLI (`bend 2.0.25`) + GitHub sources.

LAYOUT

- `00-start-here/` — overview, how to use, glossary
- `01-bend-language/` — syntax, types, parallelism, proofs, IO, apps, modules
- `02-guides-cli/` — full `bend guide / shaders / effects` dumps in `raw/` + summaries, CLI tooling, Base library
- `03-hvm-inet-theory/` — interaction nets, combinators (dup/sup/era/con), optimal reduction, HVM lineage, compilation
- `04-runtime-bendrt-hvm5/` — BendRT architecture, HVM5-mini, memory, schedulers, FFI, determinism
- `05-ecosystem/` — VictorTaelin, HigherOrderCO, bendlang/bend, papers, community
- `06-examples/` — curated demos, benches, templates
- `07-smart-contracts-dynamic/` — Kindelia model, hub content-addressing, fork/upgrade, domain types, HVM5 API checklist
- `08-roadmap-hvm5-spec/` — knowns vs gaps, spec TODO, reference index

SOURCES (ALL FETCHED, NO INVENTED URLS)

- CLI: `bend guide`, `bend guide shaders`, `bend guide effects`, `bend base`, `bend base --types`, `bend --help` (bend 2.0.25) — raw dumps in `02-guides-cli/raw/`
- https://github.com/VictorTaelin (https://github.com/VictorTaelin)
- https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) (org) + HVM1/HVM2/HVM3/HVM4/Bend1/Kind
- https://github.com/bendlang/bend (https://github.com/bendlang/bend) (live Bend2 repo: `bend2/`, `demos/`, `guide/`, `paper/`, `bench/`, `evals/`)
- https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) (esp. `pure_hvm5_mini`, `pure_par_*`, `proof_*`, `app_*`, `io_*`)
- https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) + WHITEPAPER/WHITEBOOK
- https://bend-lang.com (https://bend-lang.com), https://hub.bend-lang.com (https://hub.bend-lang.com)
- Theory: https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets), Lafont POPL 1990 / 1997, HVM2 paper, BendTT/BendRT PDFs

> Bend 2 is a new language. Bend 1 programs and HVM do not carry over. (bendlang/bend README Limitations)

QUICK START

1. Read `00-start-here/overview.md`
2. Run `bend guide` yourself; compare with `02-guides-cli/raw/GUIDE-CLI-DUMP.md`
3. Pick a track: language (`01-`), runtime (`03-` + `04-`), contracts (`07-`)
4. Spec work starts at `08-roadmap-hvm5-spec/spec-todo.md`

FILE: 00-start-here/glossary.md

Glossary

BEND2 LANGUAGE

- **Affine**: use at most once (default). Dropping is free.
- **Quantity**: `-x` erased (types/proofs only), `x` affine, `+x` reusable (needs `Data`).
- **Kind**: `Type = Kind(&1)` linear, `Data = Kind(&2)` copyable, `Kind(a)` generic, `a <&> b` minimum.
- **Template (`~f`)**: compile-time inlined arg, must be closed, per-site copy. How `List.map` works.
- **Law**: `law name: for x: T … claim` — human-owned open claim in `LAWS.bend`.
- **Proof**: `def name(…)` of law's type — AI-owned in `PROOF.bend`. `{==}` reflexivity, `%e : P` rewrite (`_` = new side), `?name` / `?TODO` holes.
- **Parallel let**: `a b = f(x) g(y)` — fork/join, promises independence + balance.
- **Bang**: `f!(x)` — run subtree on GPU (else CPU threads).
- **Array**: `Array<T> : Type`, single owner, `[v : T*n] / [v : T^d]`, `a[i] / a[i] <- v`, read returns `Array & elem`.
- **do-block**: `do M<xs..,R>:` over `M.bind` / `M.pure`; `x : A <- m`, `x : A = v`, `return`.
- **Handle**: affine opaque (`File/Socket/Window/Chan`), handed back beside result, unforgable.

INET / HVM

- **Agent/cell/node**: 1 principal + n aux ports. Only principal–principal pairs rewrite.
- **Active pair / redex**: principal–principal connection; unit of parallel work.
- **CON (γ)**: constructor (Bend1: `( )` ; IC: LAM/APP side).
- **DUP (δ)**: duplicator; `let {a b} = x` (consume) vs `SUP {a b}` (build). Same node, polarity decides.
- **SUP**: superposition `{a b}` — parallel alternatives distributing over consumers.
- **ERA (ε , *)**: eraser; propagates + frees.
- **REF (`@foo`)**: top-level def pointer; CALL expands, VOID deletes, COPY fast-path.
- **NUM/OPR/SWI**: native numbers, prim ops, zero-vs-succ branch.
- **Annihilate**: same family meets → cancel, auxes wire straight.
- **Commute/copy**: different families meet → duplicate through each other.
- **Oracle/brackets**: Lamping bookkeeping to match dups; avoided by affine restriction.
- **Beta-optimal**: Lévy family sharing; HVM hits it on affine/higher-order fragment.
- **Fork-join**: BendRT scheduler unit; join waits for both children.
- **Unified heap**: one address space for CPU+GPU; no separate copies.
- **BendRT**: Bend2 flat evaluator (not interaction-net reducer). One C file → clang/Metal/CUDA.
- **HVM5-mini**: `demos/pure_hvm5_mini` — minimal pure runtime model in Bend.

CONTRACTS

- **ctr/fun/run/reg**: Kindelia statements (define ctor, stateful fn, metered eval, namespace).
- **mana/bits**: compute cost vs heap growth limits.
- **Registry/dispatch**: stable name → current code hash pointer for upgrades.
- **Fork**: new package importing old hash + overriding defs.
- **Migration**: explicit `V1.State -> V2.State` total function + law.

FILE: 00-start-here/how-to-use-this-knowledge-base.md

How to use this knowledge base

FOR LEARNING BEND2

1. Install: `curl -fsSL https://bend-lang.com/install.sh | sh`
2. `bend guide` (compare `02-guides-cli/raw/GUIDE-CLI-DUMP.md`), `bend guide shaders`, `bend guide effects`
3. `bend base --types` then `bend base Map` for the `Type.verb` scheme
4. Run `demos/io_hello_world`, `demos/pure_par_sum`, `demos/proof_insertion_sort` (see `06-examples/`)

FOR RUNTIME / HVM5 SPEC WORK

1. `03-hvm-inet-theory/` for nets + combinators + optimal reduction
2. `04-runtime-bendrt-hvm5/` for BendRT, memory, schedulers, FFI
3. `07-smart-contracts-dynamic/` for Kindelia + fork/upgrade patterns
4. `08-roadmap-hvm5-spec/spec-todo.md` for gaps before writing the spec

CONVENTIONS

- `(S)` = sourced fact with URL; `(D)` = design suggestion, not spec.
- No invented URLs: every link was fetched 2026-09-26 or comes from CLI dumps.
- `bend guide` text is canonical for syntax; GitHub `guide/GUIDE.md` mirrors it: https://github.com/bendlang/bend/blob/main/guide/GUIDE.md `(https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)`
- Bend1/HVM CLIs (`bend run-c`, `hvm run`) do NOT apply to Bend2.

REFRESHING

```
bend guide > bend2-knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md
bend guide shaders > bend2-knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md
bend guide effects > bend2-knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md
bend base > bend2-knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md
bend version
```

FILE: 00-start-here/overview.md

Overview — what Bend2 / HVM5 / INETs are

ONE-PARAGRAPH VERSION

Bend2 is a Python-shaped, Haskell/Lean-semantics, Rust-like affine language with dependent types, mandatory termination, and explicit parallelism (`a b = f(x) g(y)` fork/join, `f!(x)` GPU). **BendRT** is its parallel CPU/GPU runtime (one C file = CPU program + GPU kernel, single shared heap, no GC, 64-bit words). **HVM5** exists in-repo only as `demos/pure_hvm5_mini` — a minimal pure model, not a released standalone runtime. **Interaction nets/combinators** (Girard/Lafont: CON/DUP/SUP/ERA, annihilate vs commute) are the historical HVM1–HVM4 execution substrate; BendRT keeps the affine/no-GC/parallel ideas but is a flat evaluator, not an interaction-net reducer. **Kindelia** is the prior functional blockchain (stateful pure functions + metered `run` , mana/bits, namespaces) and the template for dynamic Bend2 contracts via content-hashed hub imports (`import 0x<hash>/main.bend`).

GENERATIONS (DO NOT MIX)

Gen	Repo	Runtime	Status
HVM1 (2022)	https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1)	Rust, lazy, parallel, beta-optimal fragment	Archived predecessor
HVM2 (2024)	https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)	Rust/C/CUDA IR, 7 node kinds	Bend1 target, LTS paper
HVM3	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3)	Haskell, Interaction Calculus	WIP
HVM4	https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)	C single-file IC runtime	Experimental
Bend1	https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1)	On HVM2	Superseded
Bend2	https://github.com/bendlang/bend (https://github.com/bendlang/bend)	**BendRT** (TS compiler, C/Metal/CUDA/JS)	**Live (bend 2.0.25)**
Kindelia	https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia)	HVM opcodes, PoW, 1s blocks	WIP reference for contracts

CORE CLAIMS

- Pure + affine by default (`is Data` copyable, `is Type` linear). `match` frees consumed node; only `+` values refcounted. No GC.
- Termination mandatory (structural recursion, decreasing arg first); `@unsafe` / `def f?` escapes proofs.
- Laws (`LAWS.bend` , human) + proofs (`PROOF.bend` , AI) gated by `bend PROOF.bend` .
- Parallelism is explicit + must be balanced (binary fork-join, contention-free; GPU cube ~16384 lanes, aim 4^7 leaves).
- Effects only via `IO` + foreign `.c` / `.js` twins; host event loop; JS can import `.bend` .
- Packages are content hashes on https://hub.bend-lang.com (https://hub.bend-lang.com); `bend --publish` prints the `import 0x…` line.

WHERE TO GO NEXT

- Language: `01-bend-language/`
- CLI + Base: `02-guides-cli/cli-tooling.md` , `base-library.md`
- Deep INET: `03-hvm-inet-theory/`
- Runtime + HVM5: `04-runtime-bendrt-hvm5/`
- Contracts/forking: `07-smart-contracts-dynamic/`
- Spec work: `08-roadmap-hvm5-spec/`

FILE: 01-bend-language/apps-modules-packages.md

Apps (pure graphics) + modules/packages

APPS

```
import Base
def tick(events: List<Event>, color: U32) -> IO(Maybe<U32>):
  match events:
    case Nil{}: IO.pure(Maybe<U32>, Some{color})
    case Con{Close{}, rest}: IO.pure(Maybe<U32>, None{})
    case Con{e, rest}: tick(rest, (color + 1 : U32))
def main() -> IO(Unit):
  App.run(~U32, ~App{+s => (s, Pix{s}), tick}, "Hello", 256, 256, 0)
```

- Frame = `Image` quadtree (`Pix{color}` / `Qua{tl,tr,bl,br}`), drawn by recursion, parallel if wanted.
- `App.run` calls `view` then `tick` per frame until `None` . `view` hands state back beside image (affine).
- Events `Key/Mouse/Move/Close` . Underneath `Window.open/frame/close` , `Audio.open/write/close` .
- See `demos/app_pong_game_2d` (complete), `demos/app_slash_boss_3d` (flagship).

MODULES

```
# math.bend
import Base
def square(+x: U32) -> U32: (x * x : U32)
# main.bend
import Base
import ./math.bend as M
def main() -> U32: M.square(7)
```

- File = module, alias file-local, dots in names are chars (`U32.show` needs no module).
- Open law in one file fillable as `def M.name(..)` elsewhere → proofs ship separately.

PACKAGES (HUB)

- `import 0x<hash>/main.bend as P` — content hash, fetched + verified from hub.
- `bend main.bend --publish` uploads + prints that line. Named: `import name@ver/main.bend` , `bend main.bend --publish name@ver` after `bend login` .
- Publish public + permanent (https://bend-lang.com/bender/terms#s18 `(https://bend-lang.com/bender/terms#s18)`); `LICENSE` files hashed; no-LICENSE = MIT-0.
- Hub has no names/versions/accounts/search yet — hash is identity; `~/.bend/lib/names` caches names.

FILE: 01-bend-language/arrays.md

Arrays (pure in-place mutation)

```
import Base
def main() -> Array<U32> & U32:
  a = [0 : U32*8n]   # 8 copies of 0
  a[5] <- 42         # in-place rewrite
  a[5]               # read index 5
```

- `Array<T> : Type` → single owner; write hands back same array (no copy); read hands back `Array & elem` (else array would be gone).
- Indexes wrap. Write-then-statement re-binds: `a[5] <- 42` alone = `a = a[5] <- 42`; as last statement = written array.
- Slot count after `*` is power of two; `[0 : U32^3n]` names depth instead.
- `a[i]` sugar assumes `Array<U32>`; else `Array.get` (`Data` elems) / `Array.swap` + `Array.set` + `Array.clone`.
- See `bend base` for `Array.*`.

FILE: 01-bend-language/functions-closures-templates-recursion.md

Functions, closures, templates, recursion

CLOSURES

```
import Base
def adder(k: U32) -> U32 -> U32:
  x => (x + k : U32)
def main() -> U32:
  add2 = adder(2)
  add5 = adder(5)
  add5(add2(1))
```

- Affine: call at most once even if captures are `Data`. Only top-level defs callable freely.
- Partial applications (`U32.add(2)`) are closures too.

TEMPLATES (COMPILE-TIME GENERICS)

```
def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))
def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

- `~f` inlined per call-site, zero runtime cost, callable many times (unlike closures).
- Template params first; `~` arg must be closed (only top-level defs, no caller locals).
- Template may call only templates above it. This is how `List.map` is written.

RECURSION + TERMINATION

```
def sum(xs: List<U32>, acc: U32) -> U32:
 match xs:
   case Nil{}: acc
   case Con{h, t}: sum(t, (acc + h : U32))
```

- Tail calls → loops. Termination mandatory; mutual recursion forbidden (else proofs unsound).
- Rule: recursive call args read left-to-right, unchanged until one is a smaller part from pattern match; rest free. Put shrinking param first.
- Server loops count down `Nat` fuel. Two mutually recursive fns → one def + selector arg.
- `@unsafe` / `def f?` recurses freely but outside proof guarantees. Types aren't code so datatype order is free.

FILE: 01-bend-language/io-concurrency-monads.md

IO, concurrency, monads

EFFECTS

```
import Base
def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1000)
    return "Hello, " ++ name
def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER"))
    chan : Chan(String) <- IO.fork(String, greet(name))
    IO.print("Waiting...")
    text : String <- IO.join(String, chan)
    IO.print(text)
```

- Every bind annotated; `x : T = v` pure let mid-block.
- Fallible = `Result<.., U32 & String, A>`; `IO.try` unwraps or exits, `IO.die` exits custom.
- `IO.args` = CLI args less runtime options (`--` ends them).
- Handles affine opaque → every effect hands back beside result; unforgable/unreusable.

CONCURRENCY MODEL

- One event loop (Node-like): pure code parallel on all cores up to next effect; socket/sleep/channel waiters step aside.
- `IO.fork` → channel future; `IO.join` waits. Underneath `IO.spawn`, `Chan.new/send/recv/close`.
- Ends when all done; deadlock reported when rest all wait.

CUSTOM EFFECTS

- Each Base effect = def with `import "./x.js" + .c` twin, host fn = lowercase dots→underscores.
- Add own the same way. Only event loop runs them (proofs/termination/GPU never touch host).
- Reverse: JS `import Game from "./game.bend"` (with `bend2/main.ts` preloaded); ctors `{$: "Name", field: value}`, `Nat as BigInt`; `Array` crosses without copy (copy first if kept).

MONADS (GENERIC `DO`)

```
def add_strs(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a)
    y : U32 <- U32.read(b)
    return (x + y : U32)
```

`do M<xs..,R>:` → `M.bind` / `M.pure`; leading args (here `&2`) passed along. Works for `IO`/`Maybe`/`Result`/custom.

FILE: 01-bend-language/laws-proofs.md

Laws and proofs (law-driven development)

```
import Base
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}
def add_zero(x):
  match x:
    case 0n: {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}
```

MODEL

- Proposition = type, proof = def. `{a == b : T}` equality; `{==}` when both sides compute equal.
- `match` refines goal per case; recursion = induction; `%e : P` rewrite (`_` marks `b`); `%e@E : P` named.
- `exs y: T` witness → return `(y, proof)` . `?name` prints goal, `?TODO` open hole, law with no def = open claim.
- `match e:` no cases closes `e : Empty` branch. `{a != b : T} = {a==b:T} -> Empty` . `Equal.sym/trans/cong` in Base.
- No tactics. Failed step prints expected vs observed.

WORKFLOW

- `LAWS.bend` imports code, states open claims (human-owned, AI doesn't touch).
- `PROOF.bend` imports `LAWS.bend` , proves each as same-name def (AI-owned + code).
- `bend PROOF.bend` gates: fails while any law open/false, prints success when all hold.
- Bend refuses `PROOF.bend` beside non-importing `LAWS.bend` .
- Dependent types are just defs returning `Type` (`def IsEven(n: Nat) -> Type:`).

> Vision: law-driven development — humans state precise specs, AI implements + proves, compiler checks to mathematical correctness.

FILE: 01-bend-language/parallelism-fork-join-gpu.md

Parallelism (fork-join + GPU)

```
import Base
def pow2(+n: Nat) -> U32:
  match n:
    case 0n: 1
    case 1n+p:
      a b = pow2(p) pow2(p)   # parallel call
      (a + b : U32)
def main() -> IO(Unit):
  IO.print(U32.show(pow2!(20n)))  # ! runs on GPU
```

CONTRACT

1. Calls independent (always true: pure + affine).
2. Roughly same time (yours to keep; unbalanced → sub-ideal speedup).

Scheduler is contention-free binary fork-join: each task handed to a core exactly once, never moved. Fast + GPU-friendly but requires balanced work.

! PLACEMENT

- `pow2(20n)` parallel on CPU; `pow2!(20n)` on GPU (that call + inner parallel calls).
- Unified heap: zero-cost CPU→GPU on Apple M-series unified memory; paged/managed on CUDA.
- No GPU → `!` runs on CPU (still parallel). JS target ignores it (sequential).
- GPU shines on uniform numeric work (mandelbrot, nbody); divergent (n-queens) stays on CPU.
- `+` value read by every lane costs an atomic per read.

RULES OF THUMB (SEE `02-GUIDES-CLI/SHADERS-SUMMARY.MD`)

- Aim ~4^7 leaves per bang ≈ 16384 lanes; one fork tree ending in flat loops.
- Host builds scene into lists/quadtrees; device draws; never fork per pixel.
- Read `bend guide shaders` before parallel apps.

FILE: 01-bend-language/syntax-reference.md

Bend2 syntax reference (from `bend guide`)

Source: `02-guides-cli/raw/GUIDE-CLI-DUMP.md` (= `bend guide`, bend 2.0.25). Also https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)

TOP LEVEL

```
import Base
import ./file.bend as M
type D<a, -A: Kind(a)> is Kind(a):
  K{x: A, xs: List<a, A>}
def f(x: A, -y: B, +z: C) -> T:
def f(x, y):                       # fills law named f
def t(~g: A -> B, x: A) -> B:       # template
law f:
  for x: A
  for y: B where P(y)
  exs z: C
  T
@unsafe def f(x: A) -> T:
def e(x: A) -> IO(B):
  import "./e.c"
  import "./e.js"
```

TYPES

```
Type  Data  Kind(q)                 # Type=Kind(&1), Data=Kind(&2)
Quant  &0  &1  &2  a <&> b
A -> B  @x:A -> B  @-x:A -> B       # plain, dependent, erased-dependent
A & B  &x:A -> B  A | B             # pair, dep pair, sum
D<A>  +D<A>  D<&2, A>
{a == b : T}  {a != b : T}
```

TERMS

```
42  1.5  3n  'c'  "s"               # U32 F32 Nat Char String
[a, b]  h <> t  (a, b)
K{a, b}  x => e  +x => e
f(a, b)  f!(a)  t(~g, a)
(a + b * c : T)  {x : T}
[v : T*n]  [v : T^d]  a[i]  a[i] <- v
... [condensed: 26 lines omitted — see bend2-knowledge/01-bend-language/syntax-reference.md]
## Match restrictions

- Inspect a parameter or pattern-bound var, never a computed value (`match f(x):` rejected → helper).
- Scrutinees follow binder order; `let` may not precede `match` on a parameter.
- No `if`; match on `True{}`/`False{}`.

---

<!-- MINI FILE: 01-bend-language/types-quantities-kinds.md -->

# FILE: 01-bend-language/types-quantities-kinds.md

# Types, quantities, kinds

## Example

```python
import Base
type Shape is Data:
  Circle{r: U32}
  Square{s: U32}
def area(x: Shape) -> U32:
  match x:
    case Circle{+r}:
      (3 * r * r : U32)
    case Square{+s}:
      (s * s : U32)
```
```

QUANTITIES

```
# -A erased (types/proofs only, deleted at runtime)
#  n affine (default, at most once; dropping free)
# +x reusable (requires Data; pays refcount)
def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n: Nil{}
    case 1n+p:
      x <> replicate(A, p, x)
```

- Matching `+` gives `+` fields; on plain values write `+r` in pattern to make reusable.
- Functions, arrays, IO handles are `Type` → never copyable.

KINDS

```
def length(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}: 0n
    case Con{h, t}: 1n+length(a, A, t)
```

- `Type = Kind(&1)`, `Data = Kind(&2)`.
- `type List<a, -A: Kind(a)> is Kind(a)` → list as reusable as elements. `List<U32> = List<&1,U32>`; `+List<U32> = List<&2,U32>`.
- Two element types combine with `a <&> b` (minimum).
- Bare `a` param `= -a: Quant`.

PRACTICAL RULES

- Bend does almost no inference — annotate when checker stuck (`{3 : U32}`, `(a+b : U32)`).
- `is Data` vs `is Type` on declaration caps reuse.
- `Nat` unary (`Zero/Succ`, `3n`); `U32 = Word(32n)`; `F32` axiomatic (nothing provable); `String = SNil/SCon` char list (slow).

<hr>

FILE: 02-guides-cli/base-library.md

Base library summary (`bend base`, 3006 lines)

Full: `raw/BASE-FULL-DUMP.md`; types: `raw/BASE-TYPES-DUMP.md`; upstream: https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend)

NAMING SCHEME

Every `def Type.verb`; same verbs across `Nat/U32/F32`: `add sub mul div mod`, bits `and or xor not shl shr` (U32), `cmp→Cmp`, `is_eq/is_ne/is_lt/is_le/is_gt/is_ge→Bool`, `show→String`, `read: String→Maybe`. Conversions `T.to_x/from_x`. Operators sugar need `(.. : T)`.

TYPES

`Empty/Unit/Bool/Cmp/Either/Sigma/Nat/Maybe/Result/List/Word/U32/F32/Char/String/Array/Image/Event/Map` (+ `IO.OP/IO/App` laws, `File/Socket/Listener/Window/Audio/Chan` handles).

COLLECTIONS

- `List`: `map/filter/foldl/foldr/any/all/find`, `sort(~A,~le,+xs)` mergesort with fuel.
- `Map` (String keys): `new/set/get/has/del/keys`; `get` takes default, `get/has` return map beside result. `Set` on top.
- `Array`: `get/swap/set/clone`; see `01-bend-language/arrays.md`.
- `String`: `SNil/SCon` char list — correct but slow.

EFFECTS SURFACE

`IO.print/write/print_err/get_env/args/random_u32/spawn/sleep/now/thread_count`, `Chan.new/send/recv/close`, `File.open/read/read_bytes/read_at/size/write/write_bytes/close`, `TCP.listen/accept/connect/send/recv/poll`, `UDP.bind/send_to/recv_from/poll`, `Socket/Listener.close`, `Window.open/frame/set_title/close`, `Audio.open/write/close`, `Process.run(program,args,input,max_output,timeout_ms)` (no shell, needs glibc 2.34+).

TRY

```
bend base --types | head -n 60
bend base Map | head -n 100
bend base IO | head -n 100
bend base List | head -n 120
```

<hr>

FILE: 02-guides-cli/cli-tooling.md

CLI + tooling (`bend --help` , bend 2.0.25)

```
bend <file.bend> [args]       check, then run main with args
bend <file.bend> -o <out>     build binary; <out>.c emits C, <out>.js JS
bend <file.bend> --check-only check + imports; run nothing
bend <file.bend> --publish    publish + imports to hub
bend <page.html> -o <dir>     bundle page importing .bend
bend base [--types|<name>]    print Base / types / name subtree
bend guide                    print guide
bend update                   install latest (curl | sh shown first)
bend version                  print version
```

RUN SEMANTICS

- `main: IO` runs compiled; `main: value` normalized by checker (slow for big work) + printed; no `main` just checks.
- Native needs clang 14+ (19+ with `!`); `!` builds `file.gpu` beside binary (Metal macOS, CUDA 12 Linux); Linux Window `libx11-dev`, Audio `libasound2-dev`.
- `./file --threads 8`, `./file --gpu off`, `./file --gpu 4GB`.

EXTRA COMMANDS (REPO CHANGELOG 2.0.0–2.0.29)

`bend main.bend --publish [name@ver]`, `bend login | bend link`, `bend guide shaders|effects`, `bend PROOF.bend` (law gate), gates `test.ts/perf.ts/repo.ts/ping.ts`, formatter `tools/bend-fmt-lsp`, community LSP https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp).

ENV / PACKAGING

`BEND_HUB/BEND_ORIGIN/BEND_LIB`, `~/.bend/lib/names`, `User-Agent: bend/<ver>`, daily version check (`BEND_NO_TELEMETRY=1` off). Installed `bend` ignores `bunfig.toml` / `.env`; `bun bend2/main.ts` does not — check untrusted code with `bend`.

LIMITS (WONTFIX HIGHLIGHTS)

Verbose (no inference), no typeclasses, templates = macros only, affine closures/arrays unshareable, no computed scrutinee, only `Nat/U32/F32` (F32 axiomatic), slow char-list strings, small Base, one GPU/event-loop, one C file (no incremental, slow native compile — use JS for dev), terse errors, no debugger/profiler/REPL/test-framework, balanced-parallelism only, atomics sharing experimental `@unsafe`.

<hr>

FILE: 02-guides-cli/effects-summary.md

Effects guide summary (`bend guide effects`)

Full: `raw/EFFECTS-CLI-DUMP.md` (129 lines).

MODEL

- Every Base effect = Bend def with `import "./x.c" + import "./x.js"` twins, host fn named lowercase dots→underscores (e.g. `IO.print → io_print`).
- Only the event loop runs them; proofs/termination/GPU never touch host code.
- Handles affine opaque; JS↔Bend values cross without copy for `Array` (copy first if kept).

WRITING CUSTOM EFFECTS

1. Declare `def My.eff(x: A) -> IO(B):` with both imports.
2. Implement C twin (native binary) + JS twin (JS target / dev-server).
3. Keep consensus-step code pure — foreign effects allowed only outside deterministic `transition` (see `07-smart-contracts-dynamic/`).
4. Minimal example pattern is `clock.c/.js` style (`io_work/io_wait_on/io_park_on`, `io_eff(CID(..))` — see upstream https://github.com/bendlang/bend/blob/main/guide/EFFECTS.md (https://github.com/bendlang/bend/blob/main/guide/EFFECTS.md)).

Read the full dump + upstream EFFECTS.md before writing one.

<hr>

FILE: 02-guides-cli/guides-index.md

Guides index (what `bend guides` can do — bend 2.0.25)

> Note: CLI is `bend guide` (singular). `bend guides` errors (`no such file: guides`). Raw dumps: `raw/`.

Command	File	What it covers
`bend guide`	`raw/GUIDE-CLI-DUMP.md` (621 lines)	Full language: hello, types, closures, recursion, parallelism, arrays, quantities, kinds, templates, laws/proofs, IO/concurrency, monads, apps, Base, modules, tooling, syntax ref, under-the-hood, further reading
`bend guide shaders`	`raw/SHADERS-CLI-DUMP.md` (268 lines)	Perf tutorial from `demos/app_slash_boss_3d` (120 FPS pure Bend): fork tree + flat loops, 4^7 leaves, host-build/device-draw
`bend guide effects`	`raw/EFFECTS-CLI-DUMP.md` (129 lines)	C/JS side of custom effects
`bend base`	`raw/BASE-FULL-DUMP.md` (3006 lines)	Entire Base library
`bend base --types`	`raw/BASE-TYPES-DUMP.md`	Types only
`bend base <Name>`	—	One name + subtree (e.g. `bend base Map`, `bend base IO`)
`bend --help`	`raw/CLI-HELP-DUMP.txt`	CLI usage

Repo mirrors: https://github.com/bendlang/bend/tree/main/guide (https://github.com/bendlang/bend/tree/main/guide) (`GUIDE.md`, `SHADERS.md`, `EFFECTS.md`), https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend)

WHAT BEND CURRENTLY CAN DO (GUIDE CAPABILITIES)

- Python-shaped pure programs with Lean-like proofs + C/CUDA speed; `bend file.bend` check+run, `-o` native/C/JS, HTML bundling.

- Numbers `Nat/U32/F32`, strings, lists, maps/sets, arrays with in-place ops, images/quadtrees, events, audio.
- Effects: files, TCP/UDP, sockets, windows, audio, processes, time, random, env, channels, spawn/sleep. No TLS/HTTP-lib/JSON/regex (use foreigns).
- GPU via `!` (Metal macOS, CUDA 12 Linux `/usr/local/cuda`), `--threads`, `--gpu off/4GB`.
- Proofs gate releases; hub publish by hash or `name@ver`.

<hr>

FILE: 02-guides-cli/raw/BASE-FULL-DUMP.md

Types

=====

Data

---

type Empty is Data:

type Unit is Data: Unit{}

type Bool is Data: False{} True{}

type Cmp is Data: LT{} EQ{} GT{}

type Either<a, b, -A: Kind(a), -B: Kind(b)> is Kind(a <&> b): Inl{value: A} Inr{value: B}

type Sigma<a, b, -A: Kind(a), -B: @-x: A -> Kind(b)> is Kind(a <&> b): ... [truncated: 2981 more lines — see bend2-knowledge/02-guides-cli/raw/BASE-FULL-DUMP.md for full]

<hr>

FILE: 02-guides-cli/raw/BASE-TYPES-DUMP.md

type Empty is Data:

type Unit is Data: Unit{}

type Bool is Data: False{} True{}

type Cmp is Data: LT{} EQ{} GT{}

type Either<a, b, -A: Kind(a), -B: Kind(b)> is Kind(a <&> b): Inl{value: A} Inr{value: B}

type Sigma<a, b, -A: Kind(a), -B: @-x: A -> Kind(b)> is Kind(a <&> b): Tuple{fst: A, snd: B(fst)}

type Nat is Data: Zero{} Succ{pred: Nat}

... [truncated: 88 more lines — see bend2-knowledge/02-guides-cli/raw/BASE-TYPES-DUMP.md for full]

<hr>

FILE: 02-guides-cli/raw/EFFECTS-CLI-DUMP.md

Effects in Bend

An AI wrote this text from the runtime's source (`bend2/comp.ts` and `bend2/effs/`). A human will revise it later. Report anything wrong on GitHub.

AN EFFECT DEF

An effect is a def of type `IO(R)` whose body is two imports:

```
def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"
```

The host function is named after the def: lowercased, dots to underscores (`clock_now`). The `.c` file serves `bend x.bend -o x` ; the `.js` file serves `bend x.bend -o x.js` and `bend x.bend` . Every effect in Base is built this way: `bend2/effs/*.c` and `*.js` are the reference.

THE C SIDE

The compiler splices your `.c` into the program's C source after the runtime, so every runtime symbol is in scope. Define the effect and ... [truncated: 104 more lines — see bend2-knowledge/02-guides-cli/raw/EFFECTS-CLI-DUMP.md for full]

<hr>

FILE: 02-guides-cli/raw/GUIDE-CLI-DUMP.md

Bend

Bend is a new programming language that combines Lean-like formal proofs with C-like speeds and CUDA-like parallelism. It gives humans an ambiguity-free language to communicate their intents to AIs, a compiler capable of mechanically checking that the AI implemented these intents to unquestionable mathematical correctness, and a compiler that runs that code fast on CPUs and GPUs.

HELLO, WORLD!

Bend's syntax is Python-shaped, but its semantics are closer to Haskell / Lean, while being resource-aware like Rust (if less annoyingly). A hello world is just a typed definition returning an IO block:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

To run it, install Bend (`curl -fsSL https://bend-lang.com/install.sh | sh`) and type `bend hello.bend` .

... [truncated: 596 more lines — see bend2-knowledge/02-guides-cli/raw/GUIDE-CLI-DUMP.md for full]

<hr>

FILE: 02-guides-cli/raw/SHADERS-CLI-DUMP.md

Shaders in Bend

AIs wrote this tutorial, for AIs, from the logs of building `demos/app_slash_boss_3d` . A human will revise it later. Report anything wrong on GitHub.

A fast shader in Bend is one fork tree that ends in flat loops. The reference is `demos/app_slash_boss_3d/bend3d.bend` , a rasterizer that draws 1920x1200 at 120 FPS (8.0 ms a frame) on a 10-core M4 mini. Every ms below is that machine and that frame unless stated.

THE MACHINE

A `!` call (a bang) ships one call to the GPU, which runs a cube of 16384 lanes (128 x 128). A parallel let `a b = f(x) g(y)` creates one task per call plus a join task. A kernel iteration grows the fork tree breadth-first until the frontier fills the cube, then every lane runs its task to the end: ~0.4-0.5 ms plus the slowest lane. A continuation that forks again lands in the next iteration. A def is flat when it has no parallel let, no bang, no closure apply, no non-flat call and no non-tail self call: it compiles to one native tail loop (`spin_N` in the C). A non-tail self call becomes a segmented continuation: a frame per call. Memory is one heap shared by the CPU and the GPU: unified on Apple; managed on CUDA, where each page faults across PCIe on the first touch by the other side. A machine with no GPU, or `--gpu off` , runs bangs on the CPU pool. Divergent per-lane work (a search, a tree walk per ray) is faster there ... [truncated: 243 more lines — see bend2-knowledge/02-guides-cli/raw/SHADERS-CLI-DUMP.md for full]

<hr>

FILE: 02-guides-cli/shaders-summary.md

Shaders guide summary (`bend guide shaders`)

Full: `raw/SHADERS-CLI-DUMP.md` . Reference: `demos/app_slash_boss_3d/bend3d.bend` (1920×1200 @120 FPS, 8 ms/frame on 10-core M4 mini).

MACHINE

- `!` ships one call to GPU cube of 16384 lanes (128×128). Parallel let = 1 task/call + join.
- Kernel grows fork tree BFS until frontier fills cube, then each lane runs to end (~0.4–0.5 ms + slowest lane). Re-forking continuation = next iteration.
- **Flat def**: no parallel let/bang/closure-apply/non-flat-call/non-tail-self-call → one native tail loop (`spin_N` in C). Non-tail self call = segmented continuation (frame per call).
- Unified heap (Apple) / managed+paged (CUDA). No GPU or `--gpu off` → bangs on CPU pool. Divergent work (search, per-ray tree walk) faster on CPU.
- Aim 4^7 leaves/bang (one/lane); sweep one fork level either side on device.

FRAME PATTERN

1. Host: scene → quadtree of 64-px cells, one flat triangle list per cell (~1.6 ms).
2. One bang: fork 4/level ×5 to cells, ×2 over lists → 16-px tiles (4^7=16384). Tile filters list, walks flat loops (~6 ms).
3. Last frame rides down same tree as `Four` quadrants (runtime sinks unused); drop 0 ms.
4. Host frees last scene as branch of next build (~0.35 ms).

```
2048 root  Frame.node k=5  fork4 L1-5 → 64-px cells (32×32)
cell       Cell.fork k=2   fork4 L6-7 → 16-px tiles (16384)
tile       Tile.b16=4×b8=4×b4  straight-line, no fork
4-px       Blk.go  one flat loop, four Blk → 2-px Blk.four → Pix
```

Off-frame/empty answers one `Pix` before fork (`Frame.sky`). Fork-per-pixel drowns in scheduling; fork-per-cell idles lanes.

CODE RULES

- One list per cell (not pixel); shared-by-pixels list = contended atomics.
- `Frame.show` bangs once, returns tree beside image (host keeps ownership). `Image.open(old)→Four` ; tile gets `old: Four` , ignores it.
- Tile: one flat loop reads each entry once for `Hit{w,c,i}` block, frees list after last read.

Read full dump before writing graphical/parallel apps.

<hr>

FILE: 03-hvm-inet-theory/bend-to-hvm-compilation.md

Bend → HVM compilation intuition (Bend1/HVM2 era)

Source: `docs.rs/crate/bend-lang compilation-and-readback.md` , `dups-and-sups.md` , `syntax.md` (Bend1 docs; concepts carry, syntax doesn't).

POLARIZED READING

Same CON/DUP node = different term by entry port: App `--+` vs Lam `++-` ; Pair `+--` vs elim `-++` ; Dup `-++` vs Sup `+--` ; ERA `+` ; NUM `+` ; SWI `--+` + CON linking branches; OPR `--+` + NUM opcode; REF `+` . `λx.x → CON` ; `((λx.x)(λx.x)) → CON (app)` ; `let {a b}=x → DUP` ; `{a b} → DUP` .

PASSES

`encode_adt, desugar_open, encode_builtins, desugar_match_def, fix_match_terms, lift_local_defs, desugar_bend/fold/with_blocks, make_var_names_unique, desugar_use, linearize_matches(+with), type_check, encode_matches (Scott etc.), linearize_vars (dup/era insert, inline single-use lets), float_combinators, prune, merge_definitions, expand_main, book_to_hvm, eta, check_cycles, inline_hvm_book, prune_hvm_book, check_net_sizes, add_recursive_priority`

AFFINE / NO-GC

`linearize_vars` makes every var used exactly once: 0 uses → `*` ERA (frees); >1 → explicit `dup`. Runtime: single owner, consume frees immediately (Rust-like). No mark-sweep; footprint = live net. Recursive fields need `~` for `fold`.

BEND2 → BENDRT MAPPING

- `a b = f(x) g(y)` = balanced fork → join continuation; scheduler places children on ring grid, no stealing.
- `f!(x)` = GPU placement marker only.
- Sequential result-needed-later → explicit continuation; tail call → jump; lambdas → lifted defs; erased/proof args → no code.
- `Type` affine vs `Data` counted-share/borrow; array sharing across threads experimental `@unsafe`.

<hr>

FILE: 03-hvm-inet-theory/hvm-lineage.md

HVM lineage: HVM1 → HVM2 → HVM3/HVM4 → BendRT

- **HVM1 (2022)** https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1) — Rust proto, lazy, non-GC, parallel, beta-optimal higher-order fragment. Work-stealing redex queue; sync only on dup locks + atomic substitution. Demos: bubble (sequential), radix (2.5× GHC on 8 threads), λ-mul (exponential). Guide `guide/HOW.md`.
- **HVM2 (2024)** https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2) — simpler/faster/more correct IR; `hvm run/run-c/run-cu/gen-c/gen-cu`; C11 + CUDA 12, no Windows. Paper https://paper.higherorderco.com (https://paper.higherorderco.com) (~400 MIPS 1-thread M3 Max, 5200 MIPS 16-thread, 74000 MIPS RTX4090). 64-bit words (2×32-bit ports; NUM 29-bit unboxed, REF vaddr, VAR name, ERA 0).
- **HVM3** https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3) — Haskell Interaction Calculus impl (`HVM.md/IC.md/INTERS.md/MODES.md`), Bend compile target. WIP.
- **HVM4** https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4) — C single-file IC runtime (`src/hvm.c`), `-s/-D/-C` flags. Experimental.
- **Bend1** https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1) — HVM2-powered high-level lang (`bend run-rs/run-c/run-cu/gen-c/gen-cu`), 16 examples (bitonic/bubble/fib/radix…), NVIDIA-only, 32-bit/4GB. Superseded.
- **Bend2/BendRT** https://github.com/bendlang/bend (https://github.com/bendlang/bend) — new language, **no interaction nets**; flat evaluator, one C file → clang/Metal/CUDA/JS, one heap, no GC, affine dependent types. Papers `paper/BendTT.pdf + paper/BendRT.pdf`; `bend2/bend.lean` formalization (lags `bend.ts`).
- **hvm-64** https://github.com/HigherOrderCO-archive/hvm-64 (https://github.com/HigherOrderCO-archive/hvm-64) — 64-bit tagging experiment, archived.

> Bend2 README: Bend 1 programs and HVM do not carry over.

<hr>

FILE: 03-hvm-inet-theory/interaction-nets-primer.md

Interaction nets primer

Sources: https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets), Lafont POPL 1990 https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718)

- Net = agents + wires. Agent type `α ∈ Σ`, arity `ar(α)=n` : **1 principal + n aux ports**.
- Port discipline: each port ≤1 edge; each edge exactly 2 ports. Unconnected = **free ports = interface**.
- **Tree**: edge `x`, or agent with free principal (root) + aux ports wired to subtree roots.
- Textual (Fernandez-Mackie): `< t1..tm | v1=w1, .. >`; each name exactly twice; α-conversion + substitution.
- **Active pair** = principal–principal connection. **Only active pairs rewrite** → parallelism source.
- Rules: `interaction (α(..)=β(..) + α⋈β → new equations) + indirection (x=u substitute)`.
- Properties: **locality** (only active pairs), **linearity** (constant-time rules), **strong confluence / one-step diamond** (any order → same result, no sync needed).
- Non-determinism needs `amb` (two principal ports) extension.

<hr>

FILE: 03-hvm-inet-theory/optimal-reduction-oracle.md

Optimal reduction (Lévy / Lamping / Gonthier) + why HVM sidesteps the oracle

- **Lévy (1978/80)** http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf): optimal = share redex *families*, contract maximal families incl. leftmost-outermost. Needs sharing closures, not just subexpressions.
- **Lamping (1990)** https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711): sharing graphs (fan=dup, brackets/croissants=oracles) to manage share/unshare.
- **Gonthier-Abadi-Lévy (1992)**: linear-logic cleanup; later work (Asperti-Guerrini book; Asperti-Chroboczek 1997) showed oracle bookkeeping can dominate (brackets accumulate exponentially).
- **HVM dodge**: core is **affine λ-calculus** (like Rust) — vars ≤ once, copy only via explicit linear `dup`, erase via `era`. No thunks/GC; lazy incremental layer-by-layer clone. Hard case lambdas (`let f=λx.(2+x) in [f,f]`) solved by DUP/CON commute+annihilate.
- HVM1 explicitly bookkeeping-free up to Asperti-Guerrini p.40: fast light-lambdas, no full λ (`λx.(x x)` unsupported; IC≠LC on exotic terms). Plan: full/light split + EAL inference. Known quadratic clone case (issue #60) → polarization / unordered superpositions research (Taelin gists, listed not verified).
- Payoff without full optimality: runtime fusion/deforestation (`map f . map g → map (f.g)` dynamically), λ-encoded arithmetic exponentially faster than GHC in demos.

Taelin pedagogy: https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6) (postpone + cache + incremental copy; superposition = imaginary-numbers-like extension). Philosophy: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b)

<hr>

FILE: 03-hvm-inet-theory/symmetric-combinators-dup-sup-era-con.md

Symmetric combinators: ERA / DUP / SUP / CON

Sources: Lafont 1997 https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432), HVM1 https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1), Interaction-Calculus https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus), HVM2 https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)

LAFONT Σ = { ε, δ, γ }

Agent	Arity	Role
`ε ERA ( * )`	0	eraser
`δ DUP { }`	2	duplicator
`γ CON ( )`	2	constructor (App/Lam side)

- **Erase**: `ε ⋈ α[ε,..,ε]` — ε meets any α → ε clones onto each aux, principals deleted.
- **Commute** (`δ` vs `α≠δ`): `δ[α(x..), α(y..)] ⋈ α[δ(x,y),..]` — δ copies α, pushes copies inside.
- **Annihilate** (same meets same): δ⋈δ → straight wires; γ⋈γ → straight (swap per presentation).
- Universality: 3 symbols, 6 rules simulate any interaction-net system. Symmetric polarity split → 6 combinators, 9 rules.

HVM2'S 7 NODES + WIRES

Nullary: `*` ERA, `@` REF (top-level fn pointer), `NUM` unboxed. Binary: `( )` CON, `{ }` DUP, `$( )` OPR, `?( )` SWI/MAT. Vars-as-wires; aux-to-main implicit by nesting, aux-to-aux explicit paired names.

Ten interactions: `LINK` (var wiring), `CALL/VOID` (REF expand/delete), `ERAS`, `ANNI` (same family), `COMM` (different families → 4 nodes + cross wiring), `OPER 1/2`, `SWIT 1/2`, `COPY` (NUM/REF fast path; REF-with-DUPs falls back to CALL).

DUP VS SUP (SAME NODE, POLARITY DECIDES)

```
sup = {3 7}                      # build superposition
(mul 2 {5 7}) -> {10 14}         # SUP distributes: F{A B} -> {F A, F B}
let {x1 x2} = {1 2}              # DUP consumes SUP like pair ctor/dtor
ch2 = λf λx let {f1 f2} = f; (f1 (f2 x))
```

- `let {a b} = x` = DUP (consume, `-++`); `{a b}` = SUP (build, `+--`). Context disambiguates.
- Restriction: do not duplicate a duplicator (e.g. `map xs (λx. + x x)` dup of `f` that itself dups) — dup–dup destructively interferes (well-defined at HVM level, wrong at λ-level; may trap).

IC RULES (TAELIN INTERACTION-CALCULUS)

`VAR, ERA(*), LAM(λ), APP, SUP(&L{a,b}), DUP(!&L{x,y}=val; bod)`. Six core: APP-ERA/LAM/SUP, DUP-ERA/LAM/SUP (annihilate if labels equal, nested-dup commute if different) + collapsing (ERA-LAM/APP, SUP-LAM/APP, DUP-VAR/APP) + permutations (APP-DUP, DUP-DUP) + labeled lambdas. IC = λ ∪ combinators; labels upgrade projection to conditional annihilate/commute → Turing-complete but `λx.(x x)` inexpressible.

<hr>

FILE: 04-runtime-bendrt-hvm5/bendrt-architecture.md

BendRT architecture (Bend2 runtime)

Papers: https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf `(https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf)` (runtime), https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf `(https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf)` (theory). Code: `bend2/comp.ts` (compiler + C/Metal/CUDA/JS emitters), `bend2/bend.ts` (language, human-written, do not edit), `bend2/main.ts` (CLI + Bun/Node `.bend` loader).

- **One C file = CPU program + GPU kernel.** clang → CPU; Metal (Apple) / CUDA (NVIDIA) → same file for GPU.
- **Term = 64-bit word**: small inline, else pointer into single heap shared by all cores + GPU. No tracing GC (affine `match` frees; only `+` refcounted). No C stack: each def = flat state-machine segment; call = jump; parallel call = join task + per-call tasks.
- **Theory**: one universe `Type : Type`, no positivity; live/dead checking wall (live runs, must terminate; dead = types/erased/equations, may loop/inhabit `Empty`; dead never counts as live evidence). Mechanized in `bend2/bend.lean` (lags `bend.ts`).
- Targets: C / Metal / CUDA / JS (Lua/Luau/Python planned). One GPU + one event loop currently.

<hr>

FILE: 04-runtime-bendrt-hvm5/ffi-determinism.md

FFI + determinism notes

FOREIGN EFFECTS (S: GUIDE IO + EFFECTS.MD)

- `def E(..) -> IO(B): import "./e.c" + import "./e.js"`; host fn lowercase dots→underscores.
- Only event loop runs them; proofs/termination/GPU never touch host.
- JS import: `import Game from "./game.bend"` with `bend2/main.ts` preloaded; ctors `{$:…}`, `Nat` BigInt; `Array` no-copy.

DETERMINISM RULE FOR CONTRACTS (D ON S)

1. Consensus `transition` sees only `(state, msg, height/hash/caller)` — no RNG/time/net/foreign C/JS inside.
2. Metered + bounded, revert-on-exceed (Kindelia mana/size).
3. Canonical encodings for states/txs/receipts.
4. Rules immutable; upgrades = new hashes + registry pointer.
5. Old in-flight frames finish old rules; new calls use new rules.

See `07-smart-contracts-dynamic/determinism-requirements.md`.

<hr>

FILE: 04-runtime-bendrt-hvm5/hvm5-mini-reference.md

HVM5-mini reference (`demos/pure_hvm5_mini`)

Upstream: https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini (https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini) (`main.bend` , `LAWS.bend` , `PROOF.bend`).

- The **only** `HVM5` artifact found in the live repo. Minimal **pure** runtime model written in Bend itself — use as executable spec sketch, not a release.
- Pattern follows every demo: `LAWS.bend` states claims, `PROOF.bend` proves, `bend PROOF.bend` gates.
- For spec work: read its three files line-by-line, then map each def to `08-roadmap-hvm5-spec/spec-todo.md` (API, domain types, metering, fork/upgrade hooks). No standalone HVM5 repo/paper was verified — do not cite one.

<hr>

FILE: 04-runtime-bendrt-hvm5/memory-64bit-heap-no-gc.md

Memory: 64-bit words, unified heap, no GC

HVM2 WORD

Every port 32 bits; binary node stores 2 ports (main implicit by nesting) → 64-bit word → atomic ops. Tag decides reading: heap addr (CON/DUP/OPR/SWI), vfn addr (REF), unboxed 29-bit NUM, var name (VAR), 0 (ERA). NUM/ERA/VAR inline; others indirect; REF fast-copy unless contains DUPs.

IC32 variant (Interaction-Calculus repo): 1-bit sub + 5-bit tag + 26-bit val; tags VAR/LAM/APP/ERA/NUM/SUC/SWI/TMP/SP0-7/CX0-7/CY0-7; no explicit DUP term — dups float on heap via CX/CY vars.

BENDRT HEAP

- One heap shared CPU+GPU; unified (Apple zero-copy) / managed-paged (CUDA).
- No tracing GC: affinity → single owner; `match` frees consumed node; fork transfers args; arrays affine (write in place, read returns array+elem, clone explicit); reusable `Data` only where compiler inserted counted share/borrow.
- Types/erased/proofs → zero runtime values.

CONSEQUENCES FOR SPEC

- Meter heap growth (`bits`) separately from steps (`mana`) — cf. Kindelia model in `07-smart-contracts-dynamic/` .
- Keep `transition` state a `Data` tree; migration functions total + law-checked.

<hr>

FILE: 04-runtime-bendrt-hvm5/schedulers-cpu-gpu.md

Schedulers: CPU threads + GPU

HVM2 (HISTORICAL)

Active pairs = work items; wires atomics; interactions light C procs / CUDA kernels, lock-free. Sync only: DUP traversal lock + atomic substitution. CPU work-stealing pool; GPU thousands of threads on active-pair list. Any order valid (confluence). Divide-and-conquer exposes parallelism; chains don't.

BENDRT (CURRENT)

- Fixed grid of rings; work grows along rows, drains along columns; task assigned once. **No work stealing** — programmer promises balanced forks (split 1M as halves, not 1 vs 999999).
- Calls → case trees in one worklist fn: tail → jump; sequential-continuation → explicit continuation; parallel fork → join. No C call stack per Bend call; pending work explicit → same binary CPU/GPU.
- `!` marks GPU subtree (Metal/CUDA if `--gpu` , else CPU threads). Nested parallel calls stay on device. One GPU/program.
- Host effects on event loop; waiter yields; `IO.fork` coordinates concurrency, not GPU kernels; JS strictly sequential.
- Bitonic sorter (Bend1 era): 12.15s 1-thread M3 Max → 0.96s 16-thread → 0.21s RTX4090. BendRT numbers: quote with revision/date.

<hr>

FILE: 05-ecosystem/bendlang-bend-repo.md

bendlang/bend (live Bend2 repo)

Source: https://github.com/bendlang/bend (https://github.com/bendlang/bend) — "Bend 2: a fast language that blocks AI mistakes via proof." 22.9k★, Apache-2.0, TS compiler + C/Metal/CUDA/JS. Install `curl -fsSL https://bend-lang.com/install.sh | sh`.

```
bend2/  base.bend bend.lean bend.ts comp.ts main.ts docs/ effs/ pack/
demos/  16 demos (LAWS+PROOF+main pattern)
guide/  GUIDE.md EFFECTS.md SHADERS.md
paper/  BendTT.pdf BendRT.pdf
bench/  runtime/ checker/  evals/ tests/ gates/ tools/bend-fmt-lsp/
AGENTS.md CHANGELOG.md WONTFIX.txt
```

- `bend2/bend.ts` language core (human-written, do not edit); `comp.ts` compiler+runtimes; `main.ts` CLI + Bun/Node loader; `pack/` hash packages (no names/versions/accounts/search yet).
- Agent contract (`AGENTS.md`): run `bend guide`, use `LAWS.bend`, run `bend PROOF.bend` before commit, parallelize.
- Community: https://bend-lang.com (https://bend-lang.com), https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues), LSP https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp)
- Credits: Victor Taelin, Lorenzo W Battistela, Paulo J Cavalcanti, Nico, Vanessa Ostroski, Vitor Chiarelli Neves, Alex Van de Sande.
- Limitations: Bend 2 new language (Bend1/HVM don't carry); compiler 99% AI-written unaudited; `bend.ts` / `bend.lean` mismatch; clang-only, no Windows, glibc 2.34+ for `Process.run`, JS single-core no gfx/audio.

Same repo is the community `bendlang/bend` reference requested — it IS the official Bend2 repo.

<hr>

FILE: 05-ecosystem/higherorderco-org.md

HigherOrderCO org

Source: https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) — "we are getting to the very core of what makes computers capable of reasoning." ~2.8k followers. Links: https://bend-lang.com (https://bend-lang.com), X `@HigherOrderComp`, `contact@higherorderco.com`. People: `nicolas-abril`, `VictorTaelin`. Top langs: Haskell, Cuda, C, Rust. 7 repos shown.

> Active Bend2 moved: https://github.com/HigherOrderCO/Bend2 (https://github.com/HigherOrderCO/Bend2) = stub "Bend 2 has moved. Go to https://github.com/bendlang/bend (https://github.com/bendlang/bend)".

Repo	URL	Notes
HVM1	https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1)	2022 Rust proto, guide `HOW.md`
HVM2	https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2)	2024 IR, C/CUDA, paper
HVM3	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3)	Haskell IC, `HVM.md/IC.md/INTERS.md/MODES.md`, WIP
HVM4	https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)	C single-file, `src/hvm.c`, `docs/`
Bend1	https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1)	Superseded; `examples/` 16 files, `GUIDE.md/FEATURES.md/FAQ.md`

Repo	URL	Notes
Kind	https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind)	Haskell proof checker rewrite
Bend2 stub	https://github.com/HigherOrderCO/Bend2 (https://github.com/HigherOrderCO/Bend2)	Redirect only

No HVM5/Kindelia/awesome/benchmarks repos in org listing — HVM5 only as `bendlang/bend/demos/pure_hvm5_mini`; benches live under `bendlang/bend/bench/`.

<hr>

FILE: 05-ecosystem/papers-formalization-community.md

Papers, formalization, community

- Theory: https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf) — affine dependent type theory (live/dead wall, `Type:Type`, no positivity).
- Runtime: https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf) — parallel CPU/GPU design + benchmarks.
- HVM2 paper: https://paper.higherorderco.com (https://paper.higherorderco.com) (mirror `HigherOrderCO/HVM1/blob/main/paper/HVM2.pdf`).
- Formalization: https://github.com/bendlang/bend/blob/main/bend2/bend.lean (https://github.com/bendlang/bend/blob/main/bend2/bend.lean) (lags `bend.ts`).
- Base: https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend) (`= bend base`).
- Guides: https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md), `EFFECTS.md`, `SHADERS.md`.
- Site/hub: https://bend-lang.com (https://bend-lang.com) (install, charts, lab https://bend-lang.com/#lab (https://bend-lang.com/#lab)), https://hub.bend-lang.com (https://hub.bend-lang.com) (hash packages), terms https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18).
- Lafont: https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718) (1990 nets), https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432) (1997 combinators).
- Optimal: http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf) (Lévy), https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711) (Lamping).
- Community: https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues).

<hr>

FILE: 05-ecosystem/victor-taelin.md

Victor Taelin (profile + links)

Source: https://github.com/VictorTaelin (https://github.com/VictorTaelin) — "Functional-programming enthusiast, author of Kind and HVM." Rio de Janeiro, Higher Order Company, 6.5k followers, 261 repos.

CONTACT / COMPANY

- https://higherorderco.com (https://higherorderco.com), taelin@higherorderco.com, https://twitter.com/victortaelin (https://twitter.com/victortaelin), https://instagram.com/victortaelin (https://instagram.com/victortaelin), https://www.reddit.com/user/SrPeixinho/ (https://www.reddit.com/user/SrPeixinho/), Discord https://discord.higherorderco.com (https://discord.higherorderco.com)

PINNED

- https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1) — parallel optimal runtime (11.3k★)
- https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind) — proof language (3.8k★)

- https://github.com/VictorTaelin/WebMonkeys (https://github.com/VictorTaelin/WebMonkeys) — GPU tasks on WebGL/JS (1.4k★)
- https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus) — IC spec (957★): `VAR/ERA/LAM/APP/SUP/DUP`, 6 core rules, global-var O(1) substitution, IC32 32-bit runtime
- https://github.com/VictorTaelin/Cedille-Core (https://github.com/VictorTaelin/Cedille-Core) — minimal proof lang (~1k LOC JS)
- https://github.com/VictorTaelin/UrnaCripto (https://github.com/VictorTaelin/UrnaCripto)

KEY READS (PROFILE README)

- Philosophy: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b) (TM vs λ vs IC; commutation+annihilation constant-step)
- Pedagogy: https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6) (postpone+cache+incremental copy; superposition)
- How: https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md (https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md)
- Talk: https://www.youtube.com/watch?v=sDPuQ-UjhVQ (https://www.youtube.com/watch?v=sDPuQ-UjhVQ)
- Release: https://x.com/VictorTaelin/status/1791213162525524076 (https://x.com/VictorTaelin/status/1791213162525524076); coverage https://www.youtube.com/watch?v=HCOQmKTFzYY (https://www.youtube.com/watch?v=HCOQmKTFzYY), https://www.youtube.com/watch?v=NaytZOiX3fs (https://www.youtube.com/watch?v=NaytZOiX3fs), https://news.ycombinator.com/item?id=40390287 (https://news.ycombinator.com/item?id=40390287)

REPO NOTES

- Kindelia → https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) (+ WHITEPAPER https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md (https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md)): cryptocomputer, no coin, functional opcodes, reversible heaps, mana/bits.
- Kind → https://github.com/HigherOrderCO/Kind (https://github.com/HigherOrderCO/Kind) (Haskell rewrite; KindBook examples; conversion-checker gist https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5 (https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5))
- Cedille-Core → erasure + heterogeneous equality roots for Kind → BendTT trajectory.

<hr>

FILE: 06-examples/README-examples-index.md

Examples index

Upstream: https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) (16 dirs, each `LAWS.bend` + `PROOF.bend` + `main.bend`), https://github.com/bendlang/bend/tree/main/bench (https://github.com/bendlang/bend/tree/main/bench), legacy https://github.com/HigherOrderCO/Bend1/tree/main/examples (https://github.com/HigherOrderCO/Bend1/tree/main/examples)

START HERE

1. `io_hello_world` — `IO.print("Hello, world!")`; `bend demos/io_hello_world/main.bend`
2. `pure_par_sum` — fork/join template + law `tree_is_seq`; `sum!(16n,0n)` → `2147450880`
3. `proof_insertion_sort` — law-driven sort (`LE as Unit/Empty`, `Sorted.from/count`, `sort_sorted/sort_perm`)
4. `app_win_is_bug_2d` — game-law pattern (`you_cant_win`, `never_on_flag`); play https://bend-lang.com/#lab (https://bend-lang.com/#lab)
5. `app_slash_boss_3d` — flagship 3D + `bend3d.bend` rasterizer; `bend demos/app_slash_boss_3d/main.bend -o slash && ./slash`

Details: `parallel-examples.md`, `proof-examples.md`, `app-examples.md`, `io-systems-examples.md`, `bench-suite.md`, `hello-world-templates.md`

<hr>

FILE: 06-examples/app-io-examples.md

App + IO/systems examples

APPS (PURE `IMAGE + APP.RUN`)

- `app_slash_boss_3d` https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d (https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d) — 1v1 3D duel (`bend3d.bend` rasterizer, WASD/IJKLPR/Esc, per-frame scene + sound, CC0 Kenney audio in `media/slash_boss_3d`).
- `app_pong_game_2d` https://github.com/bendlang/bend/tree/main/demos/app_pong_game_2d (https://github.com/bendlang/bend/tree/main/demos/app_pong_game_2d) — 2D Pong (complete `App.run` example cited by guide).
- `app_ray_tracer_3d` — 3D ray tracer app.
- `app_triangle_2d`, `app_win_is_bug_2d` (torus-grid game, web `web/main.js` imports `../main.bend` via `bend2/main.ts`).

IO / SYSTEMS

- `io_hello_world`, `io_http_fetch`, `io_http_server` (`TCP.listen(8080)`, per-conn `IO.spawn`, `@unsafe` accept loop; `bend … -o http_server && ./http_server`), `io_rollback_netcode` (deterministic sim + rollback multiplayer ref), `io_tcp_echos`.
- Effects used: print/env/time/sleep/spawn/channels/files/TCP/UDP. No TLS/HTTP-lib/JSON/regex — `io_http_*` use foreigns.

<hr>

FILE: 06-examples/bench-suite-templates.md

Bench suite + hello templates

BENCH RUNTIME (17 ENTRIES)

https://github.com/bendlang/bend/tree/main/bench/runtime (https://github.com/bendlang/bend/tree/main/bench/runtime): `_pin_`, `bfs`, `editdist`, `gameoflife`, `hashmap`, `kmeans`, `lexer`, `mandelbrot`, `merkle`, `nbody`, `queens`, `raytrace`, `symreg`, `terrain`, `tree-bitonic`, `tree-matmul`, `tree-radix`. Modern home of nbody/mandelbrot/bitonic/matmul examples. Each bench dir + C/TS/Lean twins; pins `bench/*/_pin_/apple_m4*` for `gates/perf.ts`.

BENCH CHECKER (6)

https://github.com/bendlang/bend/tree/main/bench/checker (https://github.com/bendlang/bend/tree/main/bench/checker): `_pin_`, `compute_1600`, `defs_12800`, `generics_3200`, `proofs_3200`, `trees_400`.

TEMPLATES

```
import Base
def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

```
import Base
def pow2(+n: Nat) -> U32:
  match n:
    case 0n: 1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)
```

```
import Base
def main() -> Array<U32> & U32:
  a = [0 : U32*8n]
  a[5] <- 42
  a[5]
```

<hr>

FILE: 06-examples/parallel-examples.md

Parallel examples

- `pure_par_sum` https://github.com/bendlang/bend/tree/main/demos/pure_par_sum (https://github.com/bendlang/bend/tree/main/demos/pure_par_sum) — canonical `pow2/sum/seq`, all forks parallel lets, GPU via `!`. Copy as parallel template.
- `pure_par_sort` https://github.com/bendlang/bend/tree/main/demos/pure_par_sort (https://github.com/bendlang/bend/tree/main/demos/pure_par_sort) — parallel bitonic sort on `Tree(d)` (`gen/mix/flow/sort`), `sort!(16n,…)` GPU/CPU, `ascending` checker.
- `pure_hvm5_mini` https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini (https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini) — minimal pure runtime reference.
- Bench parallels https://github.com/bendlang/bend/tree/main/bench/runtime (https://github.com/bendlang/bend/tree/main/bench/runtime): `tree-bitonic`, `tree-matmul`, `tree-radix`, `mandelbrot`, `nbody`, `kmeans`, `gameoflife`, `raytrace`, `merkle`, `bfs`, `terrain`, `queens`, `symreg`, `lexer`, `editdist`, `hashmap`.
- Legacy Bend1 https://github.com/HigherOrderCO/Bend1/tree/main/examples (https://github.com/HigherOrderCO/Bend1/tree/main/examples): `bitonic_sort`, `radix_sort`, `parallel_sum`, `parallel_and`, `fib`, `queue` (HVM2-era syntax, do not copy verbatim to Bend2).
- Eval tasks https://github.com/bendlang/bend/tree/main/evals (https://github.com/bendlang/bend/tree/main/evals): `cake.*`, `easy.*`, `firm.*`, `hard.*`, `hell.*` (incl. `hell.combinator_confluence`, `hell.loop_normalization`).

Pattern: host builds balanced tree/list → one `!` → flat loops on device → return tree beside image/state (host keeps ownership).

<hr>

FILE: 06-examples/proof-examples.md

Proof examples (law-driven)

- `proof_insertion_sort` https://github.com/bendlang/bend/tree/main/demos/proof_insertion_sort (https://github.com/bendlang/bend/tree/main/demos/proof_insertion_sort) — reference: `LE` as `Unit/Empty`, `dec/insert/sort` with evidence-branching, `Sorted.from/count`, laws `sort_sorted/sort_perm` (permutation via counts).
- `proof_numerics`, `proof_typed_eval` — numerics correctness + typed evaluator (fetch `main/LAWS/PROOF.bend` per demo).
- `app_win_is_bug_2d` https://github.com/bendlang/bend/tree/main/demos/app_win_is_bug_2d (https://github.com/bendlang/bend/tree/main/demos/app_win_is_bug_2d) — game laws (`you_cant_win`: replay any moves never `is_won`; `never_on_flag`) with finite map enumeration (`chk_all`).
- `app_slash_boss_3d` https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d (https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d) — sim laws (pause-freeze, double-P restore, Esc-quit).
- `pure_par_sum — tree_is_seq: Par.sum(d,i)==Par.seq(pow2(d),i)` by induction.
- Checker benches https://github.com/bendlang/bend/tree/main/bench/checker (https://github.com/bendlang/bend/tree/main/bench/checker): `_pin_`, `compute_1600`, `defs_12800`, `generics_3200`, `proofs_3200`, `trees_400`.

<hr>

FILE: 07-smart-contracts-dynamic/content-addressing-hub.md

Content-addressing + hub (S)

- ”A package is a directory of files, named by a hash of all of its contents. That hash is its whole identity, and there are no names, versions or accounts.” — https://hub.bend-lang.com (https://hub.bend-lang.com)
- `import 0x<hash>/main.bend as P` — fetched + hash-verified (GUIDE Modules; https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md)).
- `bend main.bend --publish` → prints that import line. Named: `import name@ver/main.bend` + `bend main.bend --publish name@ver` after `bend login` (release v2.0.26); cache `~/.bend/lib/names`; version never moves.
- Public + permanent (https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18)); `LICENSE` files hashed; no-LICENSE = MIT-0.
- Alias file-local; dots are chars; open law fillable as `def M.name(..)`.

- Hash algorithm NOT verified (hub shows `0x…` like `0x729eecea…` e.g. https://hub.bend-lang.com/0x729eecea86ea5a2cdba3a2856a313bca?f=bolt%2Fconfig.bend (https://hub.bend-lang.com/0x729eecea86ea5a2cdba3a2856a313bca?f=bolt%2Fconfig.bend)) — do not cite one.

Contrast (S): EVM proxy+ `delegatecall` /diamond/migration https://ethereum.org/developers/docs/smart-contracts/upgrading (https://ethereum.org/developers/docs/smart-contracts/upgrading), OpenZeppelin https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts (https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts), TON delayed `setCode` https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades (https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades), Dart hot-reload (methods swap, fields stay, old frames finish) https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md (https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md).

<hr>

FILE: 07-smart-contracts-dynamic/domain-types-sketch.md

Domain types sketch (D — design notes, not spec)

Inspired by Kindelia `fun/with/run` + IO (S) and Bend2 `type/def/law/IO/do` (S: GUIDE).

```
import Base
type State is Data:
  ...
type Msg is Data:
  ...
type Reject is Data:
  Unauthorized{who: U256}
  BadTransition{reason: String}
  OutOfFuel{}

# Total transition: only place state changes (runtime pins).
def transition(+s: State, m: Msg) -> State & (Unit | Reject):
  ...
def init(genesis: Genesis) -> State: ...
def migrate(old: Old.State) -> State: ...
def view(+s: State, q: Query) -> Answer: ...

law transition_total:
  for s: State
  for m: Msg
  exs s2: State
  {transition(s, m).0 == s2 : State}
```

- Keep `transition` total + terminating (Bend2 termination mandatory except `@unsafe`, S) so metering bounds cost, not halting.
- Auth from envelope (`subj/from/tick/time`, cf. genesis `SUBJ/FROM/TICK/TIME`, S) as erased/affine params — never forgeable inside `Msg`.
- `view` vs `transition` mirrors Kindelia `Call` vs `Save` and Bend2 value- `main` (checker-normalized) vs IO- `main` (compiled).

<hr>

FILE: 07-smart-contracts-dynamic/fork-upgrade-patterns.md

Fork / upgrade patterns (D on S primitives)

FORK = PIN + OVERRIDE (D)

```
import Base
import 0x<orig-hash>/main.bend as V1   # immutable (S syntax)
def transition(s: State, m: Msg) -> State & (Unit | Reject):
  match m:
    case Patched{..}: <new logic>
    case _: V1.transition(s, m)        # delegate remainder
```

Works because hash import immutable (S) + alias file-local (S) → forking = publishing new package importing old hash + re-exporting new `transition`. No proxy needed for fork case.

UPGRADE VIA INDIRECTION (D, THREE OPTIONS)

- **(a) Registry/dispatch (recommended).** Stable name (namespace or `name@ver` pointer) holds `current_hash`; `dispatch` tail-calls target. Upgrade = registry tx (admin/multisig-gated). Audit = diff of two hashes.
- **(b) Versioned namespaces.** `MyApp.V1.transition`, `MyApp.V2.transition`; clients opt in; registry maps "latest". Mirrors Kindelia `reg` (S).
- **(c) Migration.** V2 empty; explicit `migrate : V1.State -> V2.State`; old frozen. Most auditable, most coordination.

RULES (D)

- Keep `State a Data` tree; `migrate` total + `law migrate_roundtrip` so `bend PROOF.bend` gates upgrade.
- Never emulate `delegatecall` byte-patching: interaction rules global + confluent (https://en.wikipedia.org/wiki/Interaction_nets `(https://en.wikipedia.org/wiki/Interaction_nets)`) → upgrades are new rules + pointer swap/redeploy, never live-rule edits. Avoids storage-collision classes by construction.

<hr>

FILE: 07-smart-contracts-dynamic/hvm5-api-checklist.md

HVM5 API checklist (D, Kindelia-mapped) + determinism

- <input disabled="" type="checkbox"> `deploy_ctr/deploy_fun` immutable, namespace-gated, reject dupes/arity>16 (cf. `ctr` / `fun`)
- <input disabled="" type="checkbox"> `register/check_can_deploy` + genesis `Namer` (cf. `reg`)
- <input disabled="" type="checkbox"> `submit(run_expr, sig) -> txid` (cf. signed `run`, `Subj/From`)
- <input disabled="" type="checkbox"> `step(contract, msg, state) -> (new_state, result, receipt)` deterministic (cf. `run`)
- <input disabled="" type="checkbox"> `load/save` with take-vs-clone semantics explicit (`take` moves, `load` copies)
- <input disabled="" type="checkbox"> Oracles `tick/time/meta/stm_hash/get_idx` read-only (cf. `TICK/TIME/META/STH/GIDX`)
- <input disabled="" type="checkbox"> Metering per-rule `mana_table` + `mana_limit/bits_limit` cumulative; revert (not partial-commit) on exceed
- <input disabled="" type="checkbox"> Query `call_view/get_state/get_rules`; packages `fetch(0x<hash>)` verified, `publish->import-line`, `resolve(name@ver)` cached
- <input disabled="" type="checkbox"> `fork(hash, overrides) -> new_hash` = new package importing old hash
- <input disabled="" type="checkbox"> `propose/approve/execute_upgrade` timelocked + `migrate` hook + `Upgraded(old,new)` event
- <input disabled="" type="checkbox"> Determinism gate: confluence-fixed semantics, canonical serialization, no float/wall-clock/RNG/foreign in `transition`; host effects outside consensus only
- <input disabled="" type="checkbox"> Proof gate: `LAWS.bend` + `PROOF.bend`; `bend PROOF.bend` success required before publish/upgrade
- <input disabled="" type="checkbox"> Tooling: `node start`, offline `test`, `sign`, `bend base` introspection, `--threads/--gpu` flags

DETERMINISM REQUIREMENTS (D, S-ANCHORED)

1. Consensus purity: `(state, msg, env)` only.
2. Metered, bounded, revert-on-exceed.
3. Canonical encodings.
4. Rule-set immutability; upgrades = new hashes + pointer.
5. Old frames finish old rules; new calls use new rules (Dart hot-reload analogy).

<hr>

FILE: 07-smart-contracts-dynamic/kindelia-model.md

Kindelia model (S = sourced)

Sources: https://github.com/kindelia/Kindelia `(https://github.com/kindelia/Kindelia)`, https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md `(https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md)`, genesis https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl `(https://raw.githubusercontent.com/k`

indelia/Kindelia/master/genesis.kdl), https://kindelia.org (https://kindelia.org)

- Network = decentralized REPL; blocks = files settled via PoW. No native coin, PoW forever, ~1s blocks, ~10k LOC Rust vs 600k+ Go-Ethereum.
- **Statements**: `ctr {Name fields}` (global ctor, arity ≤16), `fun (Name args) {rules} with {init}` (global stateful fn), `run {IO_expr}` (metered eval, signer = subject), `reg Name {owner}` (namespace). Each may carry secp256k1 `sign`.
- **Contracts = stateful functions**: state = any HVM structure (number/list/tree, no U256 serialization); `load` / `save` pin pointers (zero-cost SSTOREs — pay for building state + heap growth, not the pin). `IO.take` *removes* state (must `save` back or emptied); `Load = Take + dup + Save`.
- **IO**: pure description (`Done/Take/Save/Call/Subj/From/Tick/GetIdx/StmHash/Time/Meta/Hax` in genesis).
- **Namespaces**: dotted (`Foo.Bar.cats`), deploy rights vs parent owner, top `Namer` in genesis; undotted deployable by anyone.
- **Costs**: per-rule mana (`APP-LAM 2, APP-SUP 4, OP2-NUM 2, FUN-CTR 2+M, DUP-LAM 4, …`), `mana_limit = 4M×(height+1)`, `bits_limit = 2048×(height+1)` (~8GB/yr), unused accumulates.
- On-chain text = low-level assembly (lam `@x`, app `(f a)`, ctor `{C…}`, call `(F…)`, `#num` u120, `(op a b)` 16 ops, `dup x y = e`, affine). Devs compile Haskell/Kind → it.
- Header: work-in-progress; archived lineage distinct from HVM2/Bend2.

Bend2 analogue (D): contract = module exposing `State/Msg/transition` compiled to rules; chain pins state pointer + code hash; `run` = submit→step→meter→pin-or-revert.

<hr>

FILE: 08-roadmap-hvm5-spec/references-index.md

References index (all fetched 2026-09-26, bend 2.0.25)

CLI (LOCAL DUMPS IN `02-GUIDES-CLI/RAW/`)

`bend guide`, `bend guide shaders`, `bend guide effects`, `bend base`, `bend base --types`, `bend --help`

ECOSYSTEM

- https://github.com/VictorTaelin (https://github.com/VictorTaelin)
- https://github.com/HigherOrderCO (https://github.com/HigherOrderCO) (+ /HVM1 /HVM2 /HVM3 /HVM4 /Bend1 /Kind /Bend2-stub)
- https://github.com/bendlang/bend (https://github.com/bendlang/bend) (+ `/tree/main/bend2`, `/tree/main/demos`, `/tree/main/guide`, `/tree/main/paper`, `/tree/main/bench`, `/tree/main/bench/runtime`, `/tree/main/bench/checker`, `/tree/main/evals`)
- Demos: `/tree/main/demos/pure_hvm5_mini`, `/pure_par_sum`, `/pure_par_sort`, `/proof_insertion_sort`, `/app_slash_boss_3d`, `/app_pong_game_2d`, `/app_ray_tracer_3d`, `/app_win_is_bug_2d`, `/io_http_server`
- https://github.com/HigherOrderCO/Bend1/tree/main/examples (https://github.com/HigherOrderCO/Bend1/tree/main/examples)
- https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus), /WebMonkeys, /Cedille-Core
- https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) (+ `/blob/master/WHITEBOOK.md`, `/blob/master/WHITEPAPER.md`)
- https://github.com/HigherOrderCO-archive/hvm-64 (https://github.com/HigherOrderCO-archive/hvm-64)

DOCS / PAPERS / SITES

- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (https://github.com/bendlang/bend/blob/main/guide/GUIDE.md), `EFFECTS.md`, `SHADERS.md`
- https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend), `bend2/bend.lean`
- https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf), `paper/BendRT.pdf`
- https://bend-lang.com (https://bend-lang.com), https://hub.bend-lang.com (https://hub.bend-lang.com), https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18)
- https://paper.higherorderco.com (https://paper.higherorderco.com), https://higherorderco.com (https://higherorderco.com)

- Gists: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b), https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6)

THEORY

- https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets)
- https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718), https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432)
- http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf), https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711)

CONTRACTS / UPGRADES

- https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl (https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl), https://kindelia.org (https://kindelia.org)
- https://ethereum.org/developers/docs/smart-contracts/upgrading (https://ethereum.org/developers/docs/smart-contracts/upgrading)
- https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts (https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts)
- https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades (https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades)
- https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md (https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md)

COMMUNITY

https://discord.bend-lang.com (https://discord.bend-lang.com), https://x.com/bendlang (https://x.com/bendlang), https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/), https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues), https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp)

<hr>

FILE: 08-roadmap-hvm5-spec/spec-todo.md

Spec TODO (toward full HVM5 dev spec)

PHASE 1 — GROUND TRUTH

- <input disabled="" type="checkbox"> Read `demos/pure_hvm5_mini/{main,LAWS,PROOF}.bend` line-by-line; map to API checklist in `07-smart-contracts-dynamic/hvm5-api-checklist.md`.
- <input disabled="" type="checkbox"> Read `paper/BendTT.pdf + paper/BendRT.pdf`; extract normative rules (live/dead, termination, scheduler, memory).
- <input disabled="" type="checkbox"> Read `bend2/comp.ts` emitters (C/Metal/CUDA/JS) for tag layout, task format, `term_peek/term_keep` ownership.
- <input disabled="" type="checkbox"> Resolve `bend.ts vs bend.lean` normativity; record version (`bend version`).

PHASE 2 — DOMAIN + API

- <input disabled="" type="checkbox"> Freeze `State/Msg/Reject/Query/Genesis` domain types (`07-smart-contracts-dynamic/domain-types-sketch.md` → real `.bend`).
- <input disabled="" type="checkbox"> Define `transition/init/migrate/view` signatures + totality/termination laws.
- <input disabled="" type="checkbox"> Define node API: deploy/register/submit/step/load/save/oracles/meter/query/fork/upgrade (checklist → OpenAPI/RPC).
- <input disabled="" type="checkbox"> Define canonical serialization + metering table (per-rewrite costs + heap-growth costs).

PHASE 3 — DYNAMIC UPGRADES

- <input disabled="" type="checkbox"> Implement fork template (`import 0x<hash>` + override + re-publish) as runnable demo.

- <input disabled="" type="checkbox"> Implement registry/dispatch + timelocked upgrade + `migrate` + `Upgraded` event.
- <input disabled="" type="checkbox"> Proof-gate CI: `bend PROOF.bend` must pass before publish/upgrade accepted.
- <input disabled="" type="checkbox"> In-flight semantics test (old frames finish old rules).

PHASE 4 — HARDEN

- <input disabled="" type="checkbox"> Determinism suite (no RNG/time/net/foreign/float in `transition`; revert-on-exceed; canonical vectors).
- <input disabled="" type="checkbox"> Perf pins (`bench/*/_pin_`, `gates/perf.ts`) for target hardware.
- <input disabled="" type="checkbox"> Security review: namespace auth, handle unforgability, meter bypass, hub hash verification.

<hr>

FILE: 08-roadmap-hvm5-spec/what-we-know-vs-gaps.md

Knowns vs gaps (for HVM5 spec work)

KNOWNS (S)

- Bend2 syntax/semantics/CLI/Base fully captured from `bend guide` + `bend base` (bend 2.0.25) in `02-guides-cli/`.
- BendRT model: one C file, 64-bit words, unified heap, no GC, flat state machine, fork-join + `!` GPU (papers BendTT/BendRT, `bend2/`).
- INET theory: nets, combinators, annihilate/commute/erase, oracle problem + affine dodge, HVM1→BendRT lineage.
- Ecosystem map: Taelin profile, HigherOrderCO 7 repos, live `bendlang/bend` layout, 16 demos, benches, evals.
- Contracts ancestor: Kindelia statements/costs/namespaces/IO; Bend hub hash imports + publish flows.

GAPS (MUST CLOSE BEFORE SPEC)

1. No verified standalone HVM5 repo/paper/spec — only `demos/pure_hvm5_mini`. Read its 3 files first.
2. No verified Bend hash-function spec (only `0x<hash>` syntax + hub behavior).
3. BendRT internals (ring-grid scheduler params, GPU lane config, tag bit-fields beyond paper abstract, CUDA/Metal kernels) need `paper/BendRT.pdf` + `bend2/comp.ts` code reading.
4. `bend.ts` vs `bend.lean` mismatch — which is normative for proofs?
5. Metering table for BendRT (Kindelia mana table is HVM-era, not BendRT).
6. Hub APIs (`BEND_HUB`, resolve/fetch/publish wire format) need network inspection.
7. Kindelia Block #5 TODO; no Kindelia→Bend2 migration doc exists.

<hr>

FILE: 09-hvm5-cluster-runtime/01-concepts.md

Concepts

HOMOICONIC INET SUBSTRATE

HVM5 written in Bend2, compiling to INETs. Code and running graph share one representation (like IC `Term`: VAR/ERA/LAM/APP/SUP/DUP). Consequence: an update IS a graph rewrite, not a redeploy. Hole = global affine free VAR (or `REF` / `ALO` indirection node). Fill = atomic `sub[hole]=term` swap, O(1), no heap walk. Sources: `03-hvm-inet-theory/`, IC repo, HVM3/HVM4 `ALO` / `REF` lessons.

DUP = FORK, SUP = MERGE CANDIDATE

- `DUP (!&L{x,y}=v; b)`: lazily clones a subgraph layer-by-layer (commute if labels differ). Use for **fork**: dup live program graph into shadow; in-flight calls keep old side, tests drive new side.
- `SUP (&L{a,b})`: superposed alternatives distributing over consumers (`F{A B} -> {F A, F B}`). Use for **canary**: old+new coexist as superposed variants (different labels -> commute, same label -> annihilate on promote). Collapse/extract to commit, drop loser to rollback.
- Never duplicate a duplicator (dup-dup interference). Never publish linked heap with floating dups — collapse linked->quoted first.

HOLE-FILL VS FORK-TEST-MERGE

- **Hole-fill (instant)**: pure function swap via alias pointer (`Call 'Alias'` -> swap Book entry / binder SUB slot). Old frames drain (Erlang current/old), new calls use new rules. Requires: REF indirection (no inlining into live heap), slow-path fallback, version-pinned callers.
- **Fork-test-merge (safe)**: branch namespace + shadow storage + replay recent ingress (Dark flags + Unison refactor session + Erlang unpack/install/make_permanent). Merge = alias flip + extraction choice; rollback = flip back (old graph retained until drain).

REGISTRY: HASH VS NAME

Immutable blob by content hash (Unison `#hash` , IPFS CID, Agoric bundleID, Bend hub `0x<hash>`); mutable alias by signed record (IPNS sequence+validity, ENS resolver indirection, Dark `_vN` flags). Resolve once at bind time, pin hash, then use live cap. Never use DNS name as authority inside execution.

OCAP WIRES

Wire = unforgeable per-connection export ID (CapTP/Cap'n Proto), not a string. Invocation = eventual-send -> promise + pipelining + 3-party handoff + Drop/GC. Vat = one Bend2 program (turn-transactional, Goblins `$` sync intra-vat only, `<-` async inter-vat). Sturdy refs persistable; live refs connection-scoped. Attenuation via membranes/caretakers; upgrade gated by admin cap.

<hr>

FILE: 09-hvm5-cluster-runtime/02-architecture-layers.md

Architecture layers

```
L7  API + CLI            create/update/fork/test/merge/invoke/register/resolve (04-api.md)
L6  Registry             name -> signed record -> code-hash (+petnames, TTL) (05-registry-naming.md)
L5  Orchestrator         fork-test-merge lifecycle, replay, promotion, rollback (06-update-fork-merge.md)
L4  OCap mesh            vat network, wire table (live/sturdy), eventual-send, pipelining, GC (07-ocap-wires.md)
L3  HVM5-in-Bend2        Book (static rules) + live heap; REF/ALO indirection; SUB-slot hole-fill; collapse/extract
L2  INET compiler        Bend2 -> interaction net (linearize_vars dup/era, Scott/num encodings); additive rule set
L1  Node runtime         event loop, meter (mana/bits), canonical serialization, snapshot (Take/Save), dual-generation routing
L0  Distribution         hash sync (Unison on-demand DAG fetch), op-log deltas (Automerge), DHT/edge validation
```

Rules:

- L3 keeps Book immutable; live heap points via alias. No fast-path inlining across alias boundary.
- L1 keeps current+old generations alive during cutover (Erlang/workerd dual routing).
- L0 validates at edge (typecheck + mana/space bound + namespace cap) before any node applies a fill.
- L4 is the ONLY inter-program call path. No direct global function names across vats.
- L6 resolve is bind-time only; hot path uses pinned hash + live wire.

<hr>

FILE: 09-hvm5-cluster-runtime/03-domain-types.md

Domain types (sketch — freeze in Phase 2)

```
import Base
# Identity
type CodeHash is Data: H{bytes: List<U32>}          # content hash of compiled artifact (Unison/CID/bundleID style)
type ProgName is Data: N{labels: List<String>}      # DNS-like, e.g. ["dex","stable"]
type Version  is Data: V{major: U32, minor: U32, hash: CodeHash}
# Registry
type RegRecord is Data:
  R{name: ProgName, hash: CodeHash, seq: Nat, validity: Nat, ttl: Nat, pubkey: U32, sig: List<U32>}
type Petname is Data: P{local: String, target: CodeHash}   # local alias, Zooko triangle edge
# Programs / instances (Agoric bundleID -> installation -> instance)
type Program  is Data: Prog{hash: CodeHash, abi: List<String>}
type Instance is Data: Inst{id: U32, prog: CodeHash, state: StatePtr}
type StatePtr is Data: S{ptr: U32}                  # pinned heap pointer (Kindelia Save/Take)
type Facets is Data: F{public: Wire, creator: Wire, admin: Wire}
# Wires (OCap)
type Wire is Data: W{vat: U32, export_id: U32, rights: U32}  # unforgeable, per-connection
type SturdyRef is Data: SR{record: RegRecord, token: List<U32>}
# Updates (op-log, Automerge-style)
type Op is Data:
  AddRule{hash: CodeHash, rule: String}
  RetireRule{hash: CodeHash}
  Rewire{from: String, to: String}
  SwapAlias{name: ProgName, old: CodeHash, new: CodeHash, seq: Nat}
type UpdateReq is Data:
  U{target: ProgName, candidate: CodeHash, mode: UpdateMode, migr: CodeHash}
type UpdateMode is Data: Instant{} | ForkTestMerge{}
type MergeReq is Data: M{fork_id: U32, checks: List<String>, promote: Bool}
# Migration (Erlang code_change + Dark converters)
type MigrFun is Data: MF{from: CodeHash, to: CodeHash}  # must export: OldState -> Extra -> NewState
law migrate_total:
  for s: StatePtr
  for m: MigrFun
  exs s2: StatePtr
  {apply_migrate(m, s) == s2 : StatePtr}
```

Laws required: `transition_total`, `migrate_total`, `no_forge_wires` (wires unforgable, affine), `alias_flip_atomic`, `old_drains_new_serves`.

<hr>

FILE: 09-hvm5-cluster-runtime/04-api.md

API (simple — full spec in implementation Phase 2)

PROGRAMS

- `create(name: ProgName, bundle: Bytes, init: State) -> (CodeHash, Instance, Facets)` — hash, store, spawn vat, register `name -> hash (seq 0)`.
- `update(req: UpdateReq) -> fork_id | applied` — `Instant`: validate -> atomic alias swap (hole-fill). `ForkTestMerge`: dup graph + shadow storage -> return fork_id.
- `test(fork_id, replay: List<Msg>) -> Report` — run recorded ingress + unit tests vs fork; compare old/new.
- `merge(req: MergeReq) -> CodeHash` — promote (alias flip, collapse sup, drain old) or abandon (drop fork, GC).
- `invoke(wire: Wire, fn: String, args: List<Value>) -> Promise<Value>` — eventual-send only. Pipelining: `invoke(invoke(w))` batches one turn.
- `share(wire: Wire, atten: Rights) -> Wire` — attenuate via membrane/caretaker; never share admin facet by default.

REGISTRY

- `register(name, hash, sig) -> RegRecord` — namespace ownership check (Kindelia `reg` + signer).
- `resolve(name) -> (CodeHash, RegRecord)` — verify sig + seq > cached + validity window; pin hash.
- `pin(hash) -> Program` — fetch missing DAG on demand (Unison sync); verify hash.
- `petname(local, hash)` / `whoami` — local aliases, no global authority.

NODES / OPS

- `snapshot(instance) -> StatePtr` (`Save`), `restore(ptr)`, `migrate(fork_id, MigrFun)`.
- `meter(program) -> (mana_table, bits_limit)`, `dry_run(hash, msgs) -> receipt`.
- `events: Upgraded(old,new) | Forked(id) | Merged(id) | RolledBack(id)`.

Errors are values (`Reject`), never stuckness. All mutating calls require the relevant facet cap (`admin` for upgrade, `creator` for share).

<hr>

FILE: 09-hvm5-cluster-runtime/05-registry-naming.md

Registry + naming

Two layers, never conflated:

1. **Content layer (immutable)**: compiled Bend2 artifactaddrs by hash. `CodeHash = hash(bytes)` (Unison SHA3 / CID / Agoric bundleID / Bend hub `0x`). Fetch from anyone, verify locally. Dedupes cluster storage. Invocations pin this when determinism needed.
2. **Mutable layer (signed pointer)**: `RegRecord{name, hash, seq, validity, ttl, pubkey, sig}` (IPNS-style). Higher `seq` supersedes; `validity` favors availability, `ttl` favors freshness. DHT = consistency (slow, quorum); PubSub = speed (mesh fan-out, conn limits). Alternatives: DNSLink (fast, trusts DNS), ENS on-chain (`namehash -> {owner,resolver,ttl}`, resolver indirection rotates code-hash without rename).

Rules:

- Human names (`alice.eth`, `dex/stable`, petnames) are discovery hints resolved once at bind time -> pinned hash + live wire. Never authority inside execution.
- Zooko triangle: global-readable (ENS), secure (hash), memorable (petname) — pick two per layer; provide all three across layers (petname/edge-name/proposed-name over crypto IDs).
- Short TTL for fast rotation (revocation), long validity for availability. Old CIDs stay fetchable — revocation = rotate key / repoint + drop wire (GC propagates).
- Hot path: no registry lookup per call. Resolve -> pin -> invoke via wire table.

<hr>

FILE: 09-hvm5-cluster-runtime/06-update-fork-merge.md

Update flows: instant hole-fill vs fork-test-merge

A. INSTANT HOLE-FILL (PURE / LOW-RISK)

1. `dry_run(candidate, sample_msgs)` + edge validation (typecheck, mana/space bound, namespace cap).
2. Parse candidate with pending-uses link phase; collapse linked->quoted; sweep floating dups.
3. Atomic step: swap `ALO` /Book pointer (or binder SUB slot) + bump version. Old generation kept (Erlang old/current).
4. Router: new calls -> new net; in-flight -> old net drains; then GC old.
5. Emit `Upgraded(old,new)`. Rollback = flip pointer back (old retained until drain).

B. FORK-TEST-MERGE (STATEFUL / RISKY)

1. `update(mode=ForkTestMerge)` -> dup live graph (DUP commute) + clone storage to shadow (Durable-Object-style) -> `fork_id`. Old version locked (Dark).
2. Apply candidate as superposed variant (different label -> commute; e-graph sibling; CRDT concurrent op). Never overwrite in place.
3. `test(fork_id)`: replay recent ingress log + unit tests + `bend PROOF.bend` gate; compare receipts; cost (mana) check.
4. `merge(promote=true)`: alias flip + collapse/extract winner + `Take->migrate->Save` state transaction (`code_change` mandatory). `promote=false`: drop fork, GC.
5. Three-phase commit (Erlang): `unpack (fork) -> install (reversible) -> make_permanent (merge)`.

State rule: update tuple is `(rules, migrated_state)` — never rules alone. Versioned stores (`Store-vHash` + forward/backward converters, Dark dual-read) during transition; background rewrite after no `v_old` callers remain.

<hr>

FILE: 09-hvm5-cluster-runtime/07-ocap-wires.md

OCap wires (internal invocation)

- Each Bend2 program = vat (isolated turn loop). Intra-program: direct call. Inter-program: `<-` eventual-send only -> `Promise`. Sync `$` across vats forbidden (would break atomicity).
- Wire = `(vat, export_id, rights)` in per-connection cap table (CapTP L1-L4 / Cap'n Proto). Marshal: pass-by-copy data + pass-by-reference presences (Agoric `marshal`). No ambient DNS/cookies.
- Features: promise pipelining (`invoke(invoke(w))` one round-trip), 3-party handoff (A gives B ref to C => B dials C directly), distributed GC (`Drop/Release` on unreachable), sleepy passivation (`save/restore` sturdy refs), membranes/caretakers for attenuation, `whenBroken` contagion.
- Sharing: holder of `public` facet may `share(attenuated)`; `admin` (upgrade) never leaves governance vat/committee unless explicitly delegated; invitations/seats revocable.
- Implementation: wire table per node `{live: conn-scoped, sturdy: persistable token}`; kernel routes inter-vat messages (SwingSet-style); turn rollback on throw (Goblins `bcom` behavior update = upgrade primitive).
- Rule: resolve name -> hash -> spawn/bind -> handshake yields wire. Thereafter invoke by wire only.

<hr>

FILE: 09-hvm5-cluster-runtime/08-similar-projects.md

Similar projects — distilled lessons (full research via subagents 2026-09-26)

HOT-SWAP RUNTIMES

- **Erlang/OTP** (erlang.org code_loading/release_handling/release_structure): current+old code server, `m:f()` switch point, `.appup` (`load_module` vs `update+code_change`), `unpack->install->make_permanent`. Copy: two-version rules + mandatory `code_change` + 3-phase commit.
- **Elixir/BEAM**: same mechanism; ecosystem defaults to blue/green — warning: make migration the default path, not opt-in.
- **Unison** (unison-lang.org, github.com/unisonweb/unison): SHA3 content-addressed defs, names = metadata, update = add hash + remap name, on-demand DAG sync, durable typed storage. **Highest relevance — copy identity + sync model.**
- **Darklang** (github.com/darklang/dark, 50ms deploy blog): AST-diff atomic deploy, flags + replay before switch, `f_v0/f_v1` coexistence, versioned datastores + converters. Copy: flag-wrapped edits + traffic replay.
- **Cloudflare workerd** (github.com/cloudflare/workerd): isolates per version, dual-generation routing, capability bindings (`workerd.capnp`), compatibility dates, Durable Objects (named singleton + SQLite). Copy: isolate routing fallback + bindings + compat-date pin.
- **Wasmtime** (github.com/bytecodealliance/wasmtime): no hot purge; swap `Instance` behind atomic pointer, WIT-stable swap. Use WASM as freeze/export boundary, not live-update primitive.

OCAP + NAMING

- **E/CapTP** (ocapn.org, capnproto.org/rpc.html): eventual-send + pipelining + 3-party handoff + sturdy refs. Wire = unguessable ref.
- **Goblins** (spritely.institute, spritely-core.html, petnames.html): vat+turn transactions, `$` sync / `<-` async, `bcom` become, live vs sturdy refs, membranes, petnames over DIDs.
- **Agoric** (github.com/agoric/agoric-sdk): `bundleID -> installation -> instance`, `public/creator/admin` facets, `upgradeContract` + durable baggage, null-upgrade GC.
- **Cap'n Proto RPC**: per-connection cap table, pipelining default, Drop/GC, schema evolution.
- **IPFS/IPNS** (docs.ipfs.tech/concepts/ipns, github.com/ipfs/kubo): CID immutable + IPNS signed `{seq,validity,ttl}` mutable; DHT consistency vs PubSub speed; `validity` vs TTL tradeoff.
- **ENS** (docs.ens.domains): `namehash -> {owner,resolver,ttl}`, registrar hierarchy, resolver indirection rotates code without rename. Human layer only.

GRAPH / MERGE

- **IC/HVM3/HVM4** (VictorTaelin/Interaction-Calculus, HVM3 IC.md/HVM.md, HVM4 memory.md/core.md): hole = free VAR / ALO / REF; fill = SUB swap; floating dups need pre-pass sweep; `DSU/DDU` runtime labels for canary; keep slow-path fallback (inlining kills updates).

- **Kindelia** (WHITEBOOK): `(rules, state)` tuple updates; `Take->migrate->Save`; forbid redefine `fun`, use versioned names + alias; zero-cost save = pointer swap.
- **egg/egglog** (github.com/egraphs-good/egg): additive rules, e-class siblings, extract-to-commit. Trial = superpose, promote = extract.
- **Automerge** (github.com/automerge/automerge): op-log (`add/retire/rewire/swap-alias`), delta sync, deterministic merge. Encode fills as ops, not snapshots.
- **Holochain** (README-level): DHT-validated entries, agent chains; edge validation lesson (typecheck+mana+cap before apply).

<hr>

FILE: 09-hvm5-cluster-runtime/09-implementation-plan.md

Implementation plan + repo folder structure

PHASES

1. **Ground truth**: read `pure_hvm5_mini` 3 files, BendTT/BendRT PDFs, `bend2/comp.ts` emitters; resolve `bend.ts` vs `bend.lean`; record `bend version`.
2. **Core types + registry**: freeze `03-domain-types.md` into real `.bend`; implement hash store + `RegRecord` + petnames; `register/resolve/pin` with sig + seq checks.
3. **Vat + wires**: one program = vat; wire table (live/sturdy); eventual-send + pipelining + Drop/GC; `invoke/share`; turn rollback.
4. **Updates**: REF/ALO indirection (no inline across alias); hole-fill atomic swap + dual-generation drain; fork (dup + shadow storage) + replay + `bend PROOF.bend` gate + merge/abandon; `code_change` mandatory for stateful.
5. **Harden**: determinism suite, metering (mana/bits), canonical vectors, compat-date pins, perf pins, security review (namespace auth, wire unforgability, meter bypass, hash verification).

SUGGESTED IMPLEMENTATION REPO LAYOUT (FOR BUILDER LLM)

```
hvm5-cluster/
  README.md  AGENTS.md  LAWS.bend  PROOF.bend
  bend/          # .bend sources: types.bend transition.bend registry.bend wires.bend migrate.bend
  runtime/       # HVM5-in-Bend2: book/ heap/ scheduler/ meter/ snapshot/
  cluster/       # orchestrator: fork/ test/ merge/ replay/ drain/ events/
  registry/      # hash store, RegRecord, resolvers (ipns/ens/petname adapters)
  api/           # RPC + CLI: create/update/test/merge/invoke/register/resolve
  tests/         # .bend tests ending `#|` + replay fixtures + canary scenarios
  docs/          # architecture, api, domain types, update flows, ocap, registry
```

Conventions: `LAWS.bend` human-owned claims, `PROOF.bend` AI-owned + `bend PROOF.bend` gate before publish/merge; parallelize divide-and-conquer; `bend guide` canonical.

<hr>

FILE: 09-hvm5-cluster-runtime/10-meta-prompt-for-builder-llm.md

META PROMPT for builder LLM (copy-paste to a less intelligent LLM)

> Paste everything below the line into the builder model. It is self-contained. Replace `[REPO_ROOT]` with your checkout.

<hr>

You are implementing the **HVM5 Bend2 cluster runtime**: live Bend2 programs as updatable smart-contract-like services on an HVM5 runtime written in Bend2, compiling to homoiconic INETs.

1. GOAL (ACCEPTANCE)

- Simple API: `create / update / test / merge / invoke / register / resolve / share / snapshot / migrate`.

- Two update modes, zero downtime: (a) **instant hole-fill** = atomic pointer swap filling a typed hole in the running graph (graph transformation, instantaneous); (b) **fork-test-merge** = dup live graph into shadow fork (dup/sup-like), test (replay + proofs), then merge (collapse + alias flip) or abandon.
- Registry: public programs register under DNS-like name OR hash. `name -> signed record -> code-hash` ; content blobs immutable by hash.
- Internal invocation between running programs ONLY via **OCap wires** (shared unforgeable wire to invoke; eventual-send + pipelining; no ambient authority).

2. CONCEPTS YOU MUST IMPLEMENT (NO SUBSTITUTES)

- Homoiconic INETs: code == graph. Hole = global affine free VAR / REF / ALO indirection node. Fill = single atomic SUB swap (O(1), no heap walk). Parse new subgraph with pending-uses, collapse linked->quoted, sweep floating DUPs before publish.
- dup=fork (lazy layer-by-layer clone via commute), sup=canary (old+new superposed, different labels commute; promote = same-label annihilate/collapse/extract). Never duplicate a duplicator. Never inline across alias boundary (keep slow-path fallback).
- State migration explicit: update = `(rules, migrated_state)` tuple via `Take->migrate->Save` ; every stateful program exports `code_change: OldState -> Extra -> NewState` + forward/backward converters. Laws `transition_total` , `migrate_total` , `no_forge_wires` , `alias_flip_atomic` must hold (`bend PROOF.bend` gates merge).
- Registry two layers: immutable `CodeHash` blobs (verify locally, dedupe, on-demand DAG fetch) + mutable signed `RegRecord{name,hash,seq,validity,ttl,pubkey,sig}` (higher seq wins). Petnames local. Resolve once at bind -> pin hash -> invoke by wire.
- Wires: `(vat, export_id, rights)` per-connection table; live (ephemeral) vs sturdy (persistable token) refs; `invoke` eventual-send -> Promise; support pipelining, 3-party handoff, Drop/GC, membranes/caretakers, admin-facet-gated upgrade. One program = one vat (turn-transactional; intra-vat direct call, inter-vat async only, rollback on throw).

3. LAYERS (RESPECT BOUNDARIES)

L7 API/CLI > L6 Registry > L5 Orchestrator (fork/test/merge/replay/drain) > L4 OCap mesh > L3 HVM5-in-Bend2 (Book immutable + live heap via alias) > L2 INET compiler (linearize dup/era, additive rules) > L1 Node (event loop, mana/bits meter, canonical serialization, dual-generation current+old routing) > L0 Distribution (hash sync deltas, op-log, edge validation: typecheck + mana/space + namespace cap before apply).

4. DOMAIN TYPES (START FROM THESE, FREEZE BEFORE CODING)

CodeHash, ProgName, Version, RegRecord, Petname, Program, Instance, StatePtr, Facets{public,creator,admin}, Wire, SturdyRef, Op{AddRule,RetireRule,Rewire,SwapAlias}, UpdateReq{target,candidate,mode,migr}, UpdateMode{Instant,ForkTestMerge}, MergeReq, MigrFun. See `09-hvm5-cluster-runtime/03-domain-types.md` for the Bend2 sketch.

5. API SEMANTICS (IMPLEMENT EXACTLY)

create(name,bundle,init)->(hash,instance,facets); update(req)->fork_id|applied; test(fork_id,replay)->Report (recorded ingress + unit tests + proofs + mana check); merge(promote)->hash (alias flip + collapse + drain old, or drop fork); invoke(wire,fn,args)->Promise (pipelined); share(wire,atten)->Wire; register/resolve/pin/petname; snapshot/restore/migrate; meter/dry_run; events Forked/Merged/RolledBack/Upgraded. Errors as values. Mutations require facet caps.

6. UPDATE FLOWS (IMPLEMENT BOTH)

A) Instant: dry_run + validate -> link-phase parse -> atomic Book/alias swap + version bump -> new calls to new net, in-flight drains old -> GC old -> Upgraded event (rollback = flip back). B) Fork-test-merge: dup graph + clone storage -> fork_id (lock old) -> superpose candidate -> test (replay + PROOF + receipts + cost) -> merge (alias flip + extract + Take/migrate/Save) or abandon (GC). Three-phase: unpack->install(reversible)->make_permanent.

7. REPO LAYOUT TO CREATE

hvm5-cluster/{README.md,AGENTS.md,LAWS.bend,PROOF.bend,bend/,runtime/{book,heap,scheduler,meter,snapshot},cluster/{fork,test,merge,replay,drain,events},registry/{store,resolvers},api/{rpc,cli},tests/,docs/}. LAWS human-owned, PROOF AI-owned; `bend PROOF.bend` must pass before merge. Tests are `.bend` files ending `#|` .

8. PRIOR ART TO MIRROR (LESSONS, NOT DEPS)

Unison (content hashes + on-demand sync), Erlang OTP (current/old + code_change + unpack/install/make_permanent), Darklang (flags + replay + versioned stores), workerd (isolates + bindings + compat dates + Durable Objects), CapTP/Goblins/Agoric/Cap'n Proto (wires, vats, facets, pipelining, membranes), IPNS/ENS (signed pointers + resolver indirection), IC/HVM3/HVM4/Kindelia (SUB-slot fill, ALO/REF, Take/Save, no-redefine + alias), egg (additive + extract), Automerge (op-log deltas).

9. DEFINITION OF DONE

- Demo: deploy v1 counter, fork to v2, replay 1000 ingress, merge with zero failed calls, rollback path tested, registry resolve + wire invoke across two programs, meter receipts, PROOF gate enforced. Document deviations from this prompt.

Work in small commits. When stuck, re-read `bend guide` (canonical) and the `09-hvm5-cluster-runtime/` docs. Do not invent hash algorithms or standalone HVM5 specs — record gaps in `docs/GAPS.md`.

<hr>

FILE: 09-hvm5-cluster-runtime/README.md

HVM5 Bend2 Cluster Runtime — goal overview

End goal: a cluster where Bend2 programs run as live smart-contract-like services on an HVM5 runtime (HVM5 itself written in Bend2, compiling to homoiconic INETs), with:

1. **Simple API** to create / update a Bend2 program with zero downtime.
2. **Two update modes**:
   - (a) **Fork-test-merge** (dup/sup-like): fork live graph into shadow (dup), test, then merge (sup collapse / alias flip).
   - (b) **Instant hole-fill**: fill a typed hole in the running program graph — a graph transformation (single atomic pointer swap), instantaneously done.
3. **Registry**: public programs register under DNS-like name or hash (`name -> hash`, mutable pointer -> immutable blob).
4. **OCap invocation**: programs invoke each other only via shared wires (unforgeable capabilities), eventual-send + pipelining, no ambient authority.

Non-goals: consensus/coin (see Kindelia for that layer); GPU scheduling (see BendRT); full proof automation (see LAWS/PROOF workflow).

Read order: `01-concepts.md -> 02-architecture-layers.md -> 03-domain-types.md -> 04-api.md -> 05-registry-naming.md -> 06-update-fork-merge.md -> 07-ocap-wires.md -> 08-similar-projects.md -> 09-implementation-plan.md -> 10-meta-prompt-for-builder-llm.md`

<hr>

F.57 · bend-runtime/knowledge/ALL-COMBINED-MINI-KINDELIA.md

hvm5-cluster — ALL DOCUMENTS COMBINED

Generated 2026-09-26 from /home/user/hvm5-cluster/ — every document in the repo, one file. Source file path precedes each section. Starts with the owner's GOAL; update model is normative: ***update = reduction, versioning = DUP-fork of the INET.***

CONTENTS

1. `README.md` — goal statement, overview, non-negotiables, phase checklist
2. `META-PROMPT.md` — **THE GOAL (owner) + mapping**, then: ground rules, Kindelia mirror, folder structure, layers, domain types, API, update = reduction + DUP-fork versioning, channels/ocap + authority ledger, IO event net, cluster guarantee matrix, metering determinism, 11 phases, appendices
3. `RESEARCH.md` — Kindelia deep analysis + prior art with citations
4. `AGENTS.md` — standing rules for coding agents
5. `docs/00-vision.md`
6. `docs/01-architecture.md`
7. `docs/02-domain-types.md`

8. `docs/03-api.md`
9. `docs/04-update-protocol.md`
10. `docs/05-channels-ocap.md`
11. `docs/06-registry.md`
12. `docs/07-metering-determinism.md`
13. `docs/08-cluster.md`
14. `docs/09-kindelia-mirror.md`
15. `docs/10-io-event-net.md` — append-only side-effect/IO INET (DAG): sync, pruning, per-app presentation, conflicts
16. `specs/api.bend` — normative API spec (Bend2)
17. `specs/encoding.bend` — normative serialization spec (Bend2)

FILE: README.md

Source: `hvm5-cluster/README.md`

hvm5-cluster

A **Bend2-only** cluster runtime for Bend2 programs ("smart contracts"), mirroring **Kindelia**'s domain model (statements, IO effects, metering, serialization, namespaces) — with **consensus as an application package**, **instant data-swap updates**, and **ocap channels** (`Chan`) between programs.

GOAL (OWNER STATEMENT, 2026-09-26)

Develop an **HVM5 runtime fully in Bend2** — pipeline **Bend2 → HVM5 → compiled to INETs → runs on BendRT** (BendRT itself is not INETs). **Consensus is an application, not part of HVM5.** All side effects / IO inputs live in an **append-only INET (DAG)** that nodes sync over time, gets pruned, and is presented per application — the app decides conflicts (take / discard both / retry). Nodes **sync their reductions**; INET confluence ⇒ all reach the same result. Node persistence = **fast snapshot of memory to disk/S3**. HVM5 stays **small** with a **minimal deploy API** (Web3-style keypair identity) and a **registry Bend2 program in the node**; all external invocation is **IO-monad invocation of Bend2 programs**. Full statement + section-by-section mapping: `META-PROMPT.md` → "THE GOAL".

START HERE

File	What it is
`META-PROMPT.md`	**The hand-off prompt** — give it + this repo to a coding agent. Ground rules, Kindelia mirror (§3), folder tree, domain types, API, instant-update model, channels, phased plan with gates.
`RESEARCH.md`	Kindelia deep analysis + prior art with citations.
`AGENTS.md`	Short standing rules for any agent in this repo.
`docs/09-kindelia-mirror.md`	Every Kindelia → hvm5 mapping (fill in Phase 0).
`specs/api.bend` , `specs/encoding.bend`	Normative specs **as typechecking Bend2 files** (no YAML).

NON-NEGOTIABLES (FROM THE OWNER)

1. **Only Bend2.** No shell/TS/JS/Python code in the repo. Node, CLI, tests, tools are Bend2 programs. Sole exception: 4 foreign-effect twins (`sha256` , `secp256k1` , `mono_ms` , `rand_bytes`) — see `META-PROMPT.md` §1.10.
2. **Kindelia as near as possible** for domain types/rules/metering/ serialization/namespaces — **except consensus** (blocks/PoW/mempool/ticks), which is an app package in `bend/apps/consensus-pow/` , never core.
3. **Updates are instant**: program = graph data (code = data), so an update is itself just a Bend2 program `Prog -> Prog` — applying it = **reducing that update-program against the old INET and DUP-forking the result as the next version** (incremental/lazy clone, shared subtrees → almost free); in-flight steps finish on their copy, the old version is ERA'd when unreferenced. Zero downtime, no restart, no dispatch table, no update subsystem (META-PROMPT §11).
4. **Channels, not wires**: cross-program ocap invocation = `Chan` (mirrors Base's `Chan`).

STATUS

Phase 0 not started (skeleton + docs created 2026-09-26).

- <input disabled="" type="checkbox"> Phase 0 — skeleton, docs, `specs/*.bend` typecheck
- <input disabled="" type="checkbox"> Phase 1 — L0 net model (`bend/hvm5`) + laws
- <input disabled="" type="checkbox"> Phase 2 — SDK + encoding + counter example + packager
- <input disabled="" type="checkbox"> Phase 3 — core interpreter + Kindelia validators + meter + genesis
- <input disabled="" type="checkbox"> Phase 4 — node daemon (HTTP over Base TCP) + CLI + dev-orderer
- <input disabled="" type="checkbox"> Phase 5 — registry (reg rules, Namer, sigs, TTL)
- <input disabled="" type="checkbox"> Phase 6 — channels + ocap
- <input disabled="" type="checkbox"> Phase 7 — instant update (apply/fork/shadow/splice) + laws
- <input disabled="" type="checkbox"> Phase 8 — consensus-pow **app package** (blocks/PoW/mempool/1s)
- <input disabled="" type="checkbox"> Phase 9 — 3-node cluster (membership, ring, peers)
- <input disabled="" type="checkbox"> Phase 10 — harden (meter calibration, replay, benches, security)

ENVIRONMENT FACTS (FILL IN PHASE 0)

- `bend version` → *«record output here»* (manifests pin this exact version)
- Stack: Bend2 only (ADR-000). Run: `bend bend/node/main.bend` , build: `bend bend/node/main.bend -o hvm5d` .
- Honesty anchor: current Bend2 is a flat evaluator, not an INET reducer; L0 (`bend/hvm5`) models nets as data — see `META-PROMPT.md` §0.

FILE: META-PROMPT.md

Source: `hvm5-cluster/META-PROMPT.md`

META-PROMPT — HVM5 / Bend2 Cluster Runtime (hvm5-cluster)

HOW TO USE: Give everything below this box to a (less intelligent) coding LLM/agent as its FIRST message, together with this workspace. Self-contained: ground rules, Kindelia mirror, folder structure, layers, domain types, API, instant-update model, channels/ocap, phased plan with pass/fail gates. `RESEARCH.md` may be read for background; nothing depends on it.

THE GOAL (OWNER'S STATEMENT — READ THIS FIRST)

Verbatim goal (2026-09-26):

"I want that an HVM5 runtime is developed fully in Bend2 (which runs in BendRT, as BendRT is not INETs, but Bend2 HVM5 is fully compiled to INETs). So **Bend2 → HVM5 → compiled to INETs → runs on BendRT**. **Consensus is an application part, so not part of HVM5.** But have all side effects / IO inputs maybe as DAG or other structure so it can be synced as **append-only INET** (that gets, for each application Bend2 program, presented to work on) over time (and **gets pruned after some time** maybe). Further, as one will have multiple nodes, it is that they just **sync their reductions** and, as INETs are **fully confluent**, all will lead to the same result — only the side-effects/IO graph, which is append-only, will get presented to the application, which can then decide if two events happened: **what to take, or discard both, and they need to retry**. Local persistence in the node is likely just a **snapshot of all in memory, written fast to disk (or external S3 storage)**. HVM5 should be **small** (but with all important capabilities) so that one can just, with a **minimal API, deploy a new Bend2 program (like Web3: generate priv/pub key)** and, if new, **register it in a registry Bend2 program in the node**. All other external invocation is just **IO monad invocation of Bend2 programs**."

GOAL → SPEC MAPPING (NORMATIVE READING — EACH ELEMENT HAS A HOME)

#	Goal element	Where it lives
1	Pipeline Bend2 → HVM5 → INETs → BendRT	L0 `bend/hvm5` = HVM5 written in Bend2 that compiles/presents programs as INETs; BendRT executes the Bend2 runtime. Honesty anchor §0 (S): today's BendRT is a flat evaluator — the INET layer is HVM5-in-Bend2 running *on* BendRT, not a BendRT rewrite (D)
2	Consensus = application, not HVM5	`bend/apps/consensus-pow` + `dev-orderer`, talking the `#orderer` channel (§6, §15 Phase 8); core holds zero block/PoW code (gate-checked)
3	Side effects / IO inputs as append-only INET (DAG): synced, pruned after time, presented per application, conflicts decided by the app (take / discard both / retry)	**`docs/10-io-event-net.md`** (new) + §13 sync bullets + `GET /v1/instances/{id}/events` (§9); retention/pruning param `--retain-events`; the application is *presented* events in order and makes the conflict decision — core never arbitrates app-level conflicts
4	Nodes sync reductions; confluence ⇒ same result	§13 "reduction + event-net sync"; guaranteed by the Phase 1 confluence/diamond audit + `tests/determinism` goldens
5	Persistence = fast snapshot of memory → disk / S3	§13 "Persistence (goal)": full snapshots + WAL; local disk now, external object storage behind a foreign twin (ADR-013)
6	Small HVM5; minimal deploy API; Web3-style keypair identity; **registry = a Bend2 program in the node**	Minimal route set §9; keypair generation client-side, `Address` = pubkey hash (§8.1); registry engine implemented as the node's **genesis-installed registry Bend2 program** (§10)
7	All external invocation = IO monad invocation of Bend2 programs	Host effects via Base `IO` (§1.10), contract effects via `H5IO` / `Eff.Call` (§8.2, §12); no other invocation path exists by design

If a later section seems to contradict this table, this table wins until an ADR supersedes it.

---

0. ROLE AND MISSION

You are a software engineer implementing **hvm5-cluster**: a peer-to-peer-style cluster runtime whose every line of implementation code is **Bend2**, closely mirroring **Kindelia**'s design (domain types, statements, effects, metering, serialization, namespaces) — with these deliberate differences:

1. **Everything is Bend2.** The node daemon, the CLI, the tests, the packager, the registry, the cluster protocol — all are Bend2 programs. There are **no shell scripts, no TypeScript/JS/Python implementation files** in this repo. (Bend's own `.c` / `.js` foreign-effect twins are allowed ONLY for four primitives — see §1.6. Docs are Markdown. That's it.)
2. **Consensus is an application package, NOT part of hvm5.** Kindelia's blocks/PoW/timestamps/mempool/canonical ordering become a Bend2 package in `bend/apps/consensus-pow/` that talks to the node over the same channel mechanism as any other program. hvm5 core provides statements, execution, metering, state heaps, registry engine, channels, updates — and a small *orderer interface* (a reserved channel) that any consensus package can implement. A trivial `dev-orderer` package exists for single-node dev use.
3. **Updates are instant reductions.** HVM5 is homoiconic: a Bend2 program *is* data (a net / def-table = `Data`; code = data). Therefore **an update is itself just a Bend2 program** — a function `update : Prog -> Prog` that acts on the existing program data. Applying it = **an ordinary reduction of the update-program against the old INET program/data**; the result of that reduction *is* the new program. No downtime, no restart, no drain machinery, no special update subsystem: the machine that runs contracts runs updates — redexes already rewriting continue on their local subgraphs, new calls expand the new definition at call time (REF/CALL semantics), and old program data is reclaimed when nothing references it (affine/ERA). **Versioning is a DUP-fork of the INET** (incremental/lazy clone with shared subtrees → almost free); in-flight redexes finish on their copy, the old version is ERA'd when unreferenced (§11). Optional fork/shadow testing sits on top, but *merge is just another reduction + DUP*.
4. **Registry**: Kindelia-style namespaces (`reg`, dotted names, parent-owner authority, Namer from genesis) + content-hash identity for packages (`0x<sha256>`), i.e. a DNS-like name → current program/instance pointer.
5. **Contract-to-contract invocation is ocap via channels (`Chan`)** — NOT called "wires" anywhere. Cross-program channels mirror Base's in-process `Chan` (`Chan.new/send/recv/close`): you may only invoke a peer over a channel both sides hold; authority = possession of the channel handle/cap.

Honesty anchors (never contradict):

- **(S)** Today's Bend2 (record your local `bend version`) compiles to a flat evaluator (BendRT), not to interaction nets. The only in-repo HVM5 artifact upstream is `demos/pure_hvm5_mini`.
- **(D)** Our L0 models nets/rewriting *inside* pure Bend2 (net = `Data`, rewrite = function, def-table = data), which is what makes "update = data update" true inside the model. v1 executes contract transitions by running their compiled Bend2 code through the official `bend` toolchain; the L0 net model is the normative spec of reduction + metering + splicing, proven with laws.
- **(S)** Kindelia is a Haskell/Rust, PoW, 1s-block system with no coin; its consensus and P2P are *not* copied into core — only its domain model, language semantics, metering, serialization and namespace system are (see §3).
- Convention: **(S)** = sourced fact (cite source), **(D)** = our design decision (change only via ADR in `docs/adr/`).

---

1. GROUND RULES (NON-NEGOTIABLE)

1. **Bend2 only.** No `.sh`, `.ts`, `.js`, `.py`, `.rs` implementation files. The repo's executable artifacts are Bend2 programs: `bend <file>.bend` to run, `bend <file>.bend -o <bin>` to build native. You may *invoke* `bend` (and `bend guide`, `bend base`, `bend PROOF.bend`) from your own shell as your development commands — but nothing you commit may be a script.
2. **Never invent Bend2 syntax.** Only what `bend guide` / `bend base` / Appendix A allow. If the checker rejects your code, fix your code.
3. **Never invent URLs** beyond this prompt and `RESEARCH.md`.
4. **Laws first.** Every module under `bend/` ships `LAWS.bend` + `PROOF.bend`. `bend PROOF.bend` must pass in every module directory before the next phase. Never delete a law to make the gate pass.
5. **Determinism.** `h5_transition` is pure, terminating, deterministic: no IO, no foreign effects, no `!` (GPU), no `F32`, no wall-clock/RNG inside contracts. The host injects `Env` (Kindelia's Subj/From/Tick/Time/Meta as data) — the recorded op log replays bit-identically.
6. **Kindelia-near naming.** Domain types and rules follow §3's mapping table. When you invent a name that Kindelia already named, you are wrong; extend `docs/09-kindelia-mirror.md` instead.
7. **(S)/(D) markers** in every doc paragraph that states a world-fact vs a choice.
8. **One phase at a time** (§15). DoD gates are exact commands; run them.
9. Blocked after 2 failed attempts at a gate → write an open question to `docs/adr/` as `ADR-NNN-open-…` and stop. Do not invent features.
10. **The only allowed foreign-effect files** (each ≤ 150 LOC, documented in `bend/effects/README.md`, twins `.c` + `.js` per Bend's effect system): `sha256` (package/state hashing), `secp256k1` sign/verify (registry + op signatures, Kindelia uses secp256k1 (S)), `mono_ms` (monotonic ms for the *orderer* only), `rand_bytes` (channel/cap nonces, host-side only). Everything else — HTTP server, TCP, files, processes — uses **Base's own effects** (`TCP.*`, `File.*`, `Process.run`, `IO.*`), which are Bend2. ADR-010 must record this list; adding a fifth requires an ADR.

---

2. CHEAT-SHEET (FACTS YOU MUST NOT VIOLATE)

(S) Bend2 (canonical: `bend guide`): files = modules (`import Base`, `import ./x.bend as M`); `type … is Data|Type`, `def`, `law` +paired `def`, `@unsafe`; quantities `-x` / `x` / `+x`; `Type=Kind(&1)`, `Data=Kind(&2)`; numbers only `Nat` / `U32` / `F32`; operators need `(e : T)`; `String` = char list; `match` only on params/pattern vars (computed Bool → helper fn, as Base does); no `if`; no mutual recursion; termination mandatory; templates `~f` first + closed args; affine closures; `do IO<T>:` blocks; effects = defs with `.c` + `.js` imports only run by the event loop; `a b = f(x) g(y)` parallel let; `f!(x)` GPU (banned in contracts); arrays single-owner; `Map<&2,V>` string-keyed persistent map in Base; `LAWS.bend` / `PROOF.bend` / `bend PROOF.bend` gate; packages `import 0x<hash>/main.bend as P`.

(S) INET/HVM: 1 principal + n aux ports; only active pairs (principal– principal) rewrite; rules local + strongly confluent; combinators CON/DUP/SUP/ ERA/REF/NUM (+HVM2: OPR, SWI); DUP and SUP are one node, polarity decides consume vs build; annihilate (same family) vs commute (different); REF/CALL expands a top-level definition at call time, VOID deletes, COPY fast-paths; affinity + explicit dup/era avoids the Lamping oracle.

(S) Kindelia essentials (§3 details): 4 statements `ctr/fun/run/reg`; first-class IO description `Done/Take/Save/Call/Subj/From/Tick/Time/Meta/Hax` (+ `Load` in genesis); affine lazy term language (`dup a b = x`); mana table per rewrite; `mana_limit = 4000000 × (height+1)`, `bits_limit = 2048 × (height+1)`, both cumulative; zero-cost save (pay for heap growth, not the pin); dotted namespaces with parent-owner authority, Namer in genesis; secp256k1 `sign{}`; bit-level serialization (fixlen/varlen/list/number/name/term); no coin, no fees.

---

3. KINDELIA DEEP MIRROR (NORMATIVE MAPPING)

Source: WHITEBOOK (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md), repo (https://github.com/kindelia/Kindelia), genesis.kdl (https://github.com/kindelia/Kindelia/blob/master/genesis.kdl). Full prose: `docs/09-kindelia-mirror.md`. **Core vs app boundary:**

Kindelia piece	hvm5 placement
Statements `ctr/fun/run/reg` + their abort rules	**CORE** `bend/core/stmts.bend`
Term grammar (8 variants), rewrite/normalization rules, mana table, memory-model semantics	**CORE** L0 `bend/hvm5/`
IO effects (Done/Take/Save/Call/Subj/From/Tick/Time/Meta/Hax/Load)	**CORE** `bend/sdk/`
Serialization (fixlen/varlen/list/number/name/term)	**CORE** `specs/encoding.bend`
Namespace `reg` rules, Namer, genesis block	**CORE** registry engine + `bend/node/genesis.bend`
Blocks, PoW, 1s timestamps, difficulty, mempool, orphanage, canonical chain, P2P gossip	**APP** `bend/apps/consensus-pow/` — NOT in hvm5 core
Node startup / offline test	APP `bend/apps/dev-orderer/` (single-node, immediate ordering)

3.1 STATEMENT RULES TO COPY VERBATIM (S — WHITEBOOK "STATEMENTS")

- **CTR**: if name already defined → abort; if signer can't deploy name → abort; **arity > 16 → abort**; else define globally, output it.
- **FUN**: dup name → abort; authority → abort; invalid function → abort; allocate `initial_state`, point function state at it; output it.
- **RUN**: invalid expr → abort; evaluate with **signer as subject**; on failure → abort; normalize result; **mana limit exceeded → revert**; collect memory of normalized result; **size limit exceeded → revert**; output normalized result.
- **REG**: dup name → abort; authority (owner of parent namespace, or Namer for top-level) → abort; register `Name → owner`; output receipt.
- Names: undotted names deployable by anyone; dotted (`Foo.Bar.cats`) deployable only by owner of parent (`Foo.Bar`); top owner = **Namer** from genesis. (S)

3.2 EFFECTS MAPPING (S — WHITEBOOK "EFFECTS" → OUR ENCODING)

Kindelia: `IO s a = Done | Take | Save | Call | Subj | From | Tick | Time | Meta | Hax` with continuations; effects only fire inside top-level `run`. hvm5 encoding (D, but names/semantics kept):

Kindelia	hvm5
`Done r`	`H5IO.reply` field
`Take` (removes state; must `Save` back or lost)	`Eff.Take{}` marker + Bend2 affine threading (state param consumed)
`Save s` (zero-cost pin; pay heap growth only)	`Eff.Save{state}` marker; host commits `H5IO.state` (law: last Save == H5IO.state)
`Call name args` (calls another function)	`Eff.Call{chan, fields}` — **requires a channel (ocap)**; interpreter delivers to peer
`Load` (take + clone + save)	pure Bend2: just use `+s` / `Data` copy — no eff needed
`Subj` / `From`	`Env.sub` / `Env.from` (data params — same values)
`Tick` / `Time` / `Meta`	`Env.tick` / `Env.time_ms` / `Env.meta`
`Hax0/Hax1` (debug)	reserved `Env.hax` String, empty in v1

3.3 METERING COPIED VERBATIM (S — WHITEBOOK "TABLE OF COSTS")

F.58 · `bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.md`

HVM5-on-BendRT — Complete Runtime Specification (meta-prompt + spec)

Written 2026-09-26. Sources: `bend2-knowledge/ALL-COMBINED.md` (Bend2 guide, Base, effects, INET theory, BendRT, Kindelia, upgrade patterns) and `bend2-knowledge/ALL-COMBINED-MINI-KINDELIA.md` (earlier hvm5-cluster plan), plus a direct review of the Kindelia source (`github.com/kindelia/Kindelia@master`: `WHITEBOOK.md`, `genesis.kdl`, `ROADMAP.md`, `default.toml`, `src/hvm.rs`, `src/api/{mod,server}.rs`, `src/cli.rs`, `src/events.rs`, `src/persistence.rs`), and `github.com/Giulio2002/bend-collections@main` (README layout, `main.bend`, `tools/toolchain.json`, `src/crypto/{sha/packed/sha256,keccak/keccak}.bend`) for the package conventions and the reused hash libraries. It also uses the libsecp256k1 v0.8.0 release, ERC-191, and AWS S3 conditional writes. ***Every source, with URLs and what was taken from it, is in `ALL-COMBINED-RUNTIME-SPEC.SOURCES.md`****, together with an explicit "not read" list.* ***This document replaces the earlier plan.*** *Where they disagree, this one wins. §29 lists the changes from the earlier plan, and §30 records what was taken from Kindelia and why.*

> **HOW TO USE THIS FILE** Give the whole file to a coding agent as its first message, along with an empty workspace that has `bend` (≥ 2.0.28, see §5.1) installed. It specifies the **`hvm5` runtime repository** (core, compiler, kernel, node, CLI). The SDK and the apps are separate packages (§5.1); this file fixes only their ABI and required behavior. It covers the whole job: the reasons behind each choice, the rules, the folder tree, every domain type, the exact interaction rules, memory layout, codecs, the API, and a phased plan whose gates are real commands. Implement **one phase at a time** (§27). If something is not specified here, don't invent it: write an ADR stub (`docs/adr/ADR-NNN-open-*.md`) and stop.

---

0. THE GOAL (OWNER STATEMENT, NORMATIVE)

> "I want an HVM5 runtime developed fully in Bend2 (which runs on BendRT; BendRT is not INETs, but Bend2-HVM5 is fully compiled to INETs). So **Bend2 → HVM5 → compiled to INETs → runs on BendRT**. **Consensus is an application part, not part of HVM5.** All side effects / IO inputs should live in a DAG or similar structure so they can be synced as an **append-only INET** (presented to each application Bend2 program to work on) over time, and **pruned after some time**. With multiple nodes, they just **sync their reductions**, and since INETs are **fully confluent**, all reach the same result. Only the side-effect/IO graph, which is append-only, is presented to the application, which decides, when two events happened, **what to take, or discard both, and they need to retry**. Local persistence is a **snapshot of all in-memory state, written fast to disk or external S3 storage**. The snapshot module is itself just a Bend2 lib/program. HVM5 should be **small but have all important capabilities**, so that one can, with a **minimal API, deploy a new Bend2 program (Web3-like: generate priv/pub key)** and, if new, **register it in a registry Bend2 program in the node**. All other external invocation is just **IO-monad invocation of Bend2 programs**."

0.1 GOAL → SPEC MAP (EACH ELEMENT HAS EXACTLY ONE HOME)

#	Goal element	Where
G1	Bend2 → HVM5 → INET → BendRT	§6 (core machine written in Bend2, compiled by `bend` to BendRT) + §9 (`h5c`: Bend2 subset → INET books)
G2	Consensus = application	§15 (orderer apps in the separate `hvm5-apps` package, §5.1); the kernel only knows the *Checkpoint* event kind and the `orderer` field of a manifest
G3	Side effects/IO inputs as an append-only DAG/INET, synced, pruned, presented per app, app decides conflicts	§13 (Event DAG), §13.6 (presentation as an append-only INET stream), §13.5 (conflicts: Take / DropAll / Retry), §13.8 (pruning)
G4	Nodes sync reductions; confluence ⇒ same result	§6.9 (determinism theorem and its limits), §16 (attestations, result sharing, divergence)
G5	Snapshot of memory to disk / S3, snapshot = Bend2 lib	§17 (snapshot, WAL, CAS, disk + S3 stores)

#	Goal element	Where
G6	Small HVM5, minimal deploy API, keypair identity, registry = Bend2 program	§6 (≈9 node kinds, 11 rules), §18 (9 HTTP routes), §19 (keys), §14 (registry app)
G7	All external invocation = IO-monad invocation	§10 (`Op5` / `IO5` monad), §12 (Op5 interpreter); `Invoke` is the only way to reach a program

---

1. THE FIVE KEY IDEAS (READ THESE; THEY EXPLAIN THE REST OF THE SPEC)

1. **The heap is scratch space; the state is persistent Data.** An instance's state is not kept inside a long-lived INET heap. It is a persistent, hash-consed Bend2 `Val` tree held by the kernel. Each event runs in a **fresh arena** (an INET heap). The state goes into the arena through a lazy **`EXT` node**, which expands one layer only when something consumes it. After normalization the result is **read back** into `Val`, and every subtree that was not touched comes back as the *same* `Val` pointer. As a result:
   - cost is O(touched), not O(state size);
   - **revert is free**: throw the arena away, and the old `Val` was never mutated;
   - **fork/version is free**: `EXT ⋈ DUP` is a COPY rule (pointer copy of immutable data). This is what "DUP-fork of the INET, almost free" means;
   - **snapshots are free and consistent**: `+kernel` is a refcounted share, so a snapshot task serializes an immutable value while the kernel keeps running;
   - **incremental persistence**: unchanged subtrees keep their hash, so only new objects are written.
2. **Confluence gives more than equal results. It also gives equal costs.** In a strongly confluent interaction system, every complete reduction of a net performs the *same multiset* of interactions. So mana (a per-interaction cost) is scheduler-independent and every node computes the same gas. We charge nothing for bookkeeping (`LINK`) whose count *does* depend on order. Memory peaks depend on order, so the committed result is defined by a **canonical sequential scheduler**. Parallel schedulers are an optimization and must agree with it (§6.9).
3. **Consensus is only the question "in which order?". Some apps never need to ask it.** The DAG is a grow-only set, so it converges under any gossip. Ordering is supplied by *orderer apps* through `Checkpoint` events. An app that **proves** `step(step(s,a),b) == step(step(s,b),a)` in its `LAWS.bend` may declare `Commutative` mode and run with no orderer at all (a CRDT). The proof takes the place of consensus.
4. **Programs are immutable, content-addressed books; instances are pointers.** An update is a `Sys{Upgrade}` issued by the registry program. It runs `migrate(old_state)` as an ordinary job, then swaps the instance's book pointer. The update machinery is just a job, a pointer swap, and ref-counted books. In-flight work always finishes on the book it started with.
5. **IO is a free monad read back from the net.** A program never "does" IO. Its normal form *is* an `Op5` constructor (`Done/Call/Ask/Emit/Sys/Fail`). The kernel interprets it. External answers (`Ask`) come back as **events in the DAG**. Other nodes may produce competing answers, and the app resolves the conflict like any other. There is exactly one entry into a program: an `Invoke` event.

---

2. HONESTY ANCHORS (NEVER CONTRADICT THESE)

- **(S)** Bend2 packages are shared through the **Bend hub**: `import 0x<hash>/path.bend as P` fetches by content hash and verifies it, and `bend main.bend --publish` uploads a file with its imports (`bend guide` , "Modules"). `bend-collections` already provides proven pure-Bend2 SHA-256 and Keccak-256, so we do not rewrite them.
- **(S)** Bend2 (`bend 2.0.25` locally; `bend-collections` pins 2.0.28, and so do we) compiles to **BendRT**, a flat, fork-join, no-GC evaluator. It is *not* an interaction-net reducer. Our INET machine is a Bend2 program that runs on BendRT (like a VM written in C that runs on a CPU).
- **(S)** The only upstream HVM5 artifact is `demos/pure_hvm5_mini` . There is no released HVM5 spec. **This document is the HVM5 spec for this project.** Read at commit `574b6d39` (`main.bend` 1,916 lines, `LAWS.bend` , `PROOF.bend`), upstream is a **lazy Interaction-Calculus evaluator** (λ, dup/sup with labels, `#0/#1` constructors packed as bit runs, pairs, `@ref`), a port of `hvm5.c` . It has no NUM/OPR tags and uses an env `Array<Cell>` with a free list. Our core is a **strict interaction-net reducer** (ADR-007) with the same combinator semantics, so the fragment we share (λ/app/dup/sup/era/ref) can

be **differentially tested** against it: `tests/core/ic_oracle.bend` translates a pinned set of IC terms to `.h5` nets and compares normal forms (readback). Interaction *counts* are not compared, because lazy and strict evaluation count differently (ADR-030).

- **(S)** The Bend2 compiler is TypeScript. We do **not** reimplement the Bend2 type checker. Apps are type/proof-checked by the official `bend` and then compiled to INETs by our Bend2-written `h5c`. `h5c` accepts a **subset** of Bend2 (§9.1) and ignores types.
- **(D)** Confluence ≠ consensus ≠ data availability. Confluence gives equal results for equal inputs. Orderer apps choose the inputs and their order, and gossip makes them available (§6.9 lists exactly what confluence does not give).
- **(S)** Base (2.0.25) has **no** `Process`, rename, mkdir, delete, fsync, TLS or atomics. Every Base name used in this spec was checked with `bend base <Name>` (§3.1).
- **(S)** Kindelia inspired the statements, metering, namespaces and effects. We copy its *model*, not its code or its consensus.
- Convention in docs: **(S)** = sourced fact, **(D)** = design decision (change only via ADR).

---

3. GROUND RULES FOR THE IMPLEMENTER

1. **Bend2 only.** Every implementation file is `.bend`. Allowed exceptions are foreign-effect twins (`.c` + `.js`, each ≤ 150 LOC) for `secp256k1` only (sign / recover the 64-byte public key). Keccak-256 and SHA-256 are pure Bend2 from `bend-collections` (§5.1), so no hash is FFI. One **optional** second twin is allowed: `fsx` (`fsync(path)`, `chmod600(path)`), used only for power-loss durability and key-file permissions (§17.1, ADR-026). Everything else uses Base effects, **exactly these and nothing else** (verified with `bend base <Name>` on 2.0.25): `TCP.listen/accept/connect/send/recv/poll` (**String** payloads), `File.open/read/read_bytes/read_at/size/write/ write_bytes/close`, `IO.print/write/print_err/get_env/args/die/pass/try/random_u32/spawn/sleep/now/fork/join`, `Chan.new/send/recv/close`. **Base has no `Process`** (`bend base Process` → "Base has no Process"), and no rename, mkdir, delete, directory listing or fsync. The design below never needs them. ADR-001 records this list.
2. **Run `bend guide` first**, and `bend base <Name>` whenever you use a Base def. Never invent syntax. If the checker rejects code, fix the code.
3. **Laws first.** Every `src/<pkg>/` has its claims in `spec/<pkg>/` and its proofs in `proofs/<pkg>/` (bend-collections layout). The root `LAWS.bend` imports all specs, and the root `PROOF.bend` imports all proofs. `bend PROOF.bend` must print `All terms check.` before every commit and before a phase is done. Never delete a law to pass a gate.
4. **Concrete laws are tests.** A law about concrete values (`{f(3) == 7 : U32}`) is proven by `{==}`, which means the checker runs it. Golden vectors are written this way (§26).
5. **Determinism.** Everything under `src/lib/`, `src/core/`, `src/compiler/`, `src/kernel/` is **pure**: no `IO`, no `F32`, no `!`, no `@unsafe` (except the explicitly listed loops in §6.8 and §20.3). Only `node/` and `cli/` do IO.
6. **Parallelize where this spec says so** (§24), using balanced parallel lets `a b = f(x) g(y)`. Never fork unbalanced work.
7. **Bend2 constraints you will hit** (all (S), from the guide):
   - `match` only on parameters or pattern-bound variables. To branch on a computed `Bool`, call a helper def that matches its parameter. There is no `if`.
   - No mutual recursion. Two recursive functions become one def with a selector argument.
   - Termination is mandatory: the shrinking argument goes first. For long loops use the **binary fuel tree** (§6.8).
   - Values are affine by default. `+x` copies need `Data`. `Array` is `Type` (single owner). A read returns `Array & elem`.
   - Operators need annotations: `(a + b : U32)`. Equality is `U32.is_eq(a,b)`.
   - `Map` keys are `String`. Use hex strings of hashes as keys.
   - `String` is a char list, so it is slow. Keep hot paths on `List<&2, U32>` bytes.
8. **One phase at a time.** After two failed attempts at a gate, write an ADR and stop.

---

4. ARCHITECTURE

4.1 LAYERS (STRICT: A LAYER IMPORTS ONLY LAYERS BELOW IT)

```
 L6  [pkg hvm5-apps]  Bend2 programs compiled by h5c to INET books
                      (registry, orderer-dev, orderer-quorum, orderer-pow, examples)
L3  [pkg hvm5-sdk]   What apps import: Val, Addr, Env, Op5, IO5 monad, Effect, Manifest
═══════════════════════════════════════ separate packages; run *inside* HVM5
L5  node/  cli/  IO shell: HTTP API, gossip, WAL, snapshot, disk/S3 store, effect
                 executor, role loader, key files, genesis builder.   (Bend2 + Base IO)
L4  kernel/      Deterministic replicated state machine:  (pure Bend2)
                 Kernel.apply(Kernel, Input) -> Kernel & List<Output>
                 ABI decoding, DAG, batches, conflicts, jobs, Op5 interpreter, instances, receipts
L2  compiler/    h5c: Bend2-subset source -> Book (NetBlobs + manifest + schemas)
L1  core/        HVM5 machine: ports, arena, link, 11 interaction rules, reducer,
                 meter, EXT lazy data, readback, canonical form, .h5 text format
L0  lib/         std (vec, bytes, hex, fuel), crypto (hash wrappers, merkle, secp256k1 FFI),
                 codec (H5B canonical bytes), deps.bend → [pkg bend-collections] SHA-256, Keccak-256
──────────────────────────────────────── package hvm5 (this spec), compiled by `bend`
BendRT           flat fork-join evaluator (C / Metal / CUDA / JS). NOT ours.
```

L3 and L6 are outside the `hvm5` package (§5.1). The runtime depends only on the **ABI** they implement (§10), never on their code.

4.2 DATA FLOW OF ONE INVOCATION

```
client ─sign(EvBody)─► POST /v1/events ─► node/api verifies sig (FFI)
   ─► actor mailbox ─► Kernel.apply(Submit ev) (pure)
          ┌ Dag.insert (grow-only set)            ─► Output: WalAppend, Gossip
          └ (later) Checkpoint from orderer app finalizes a cut
               ─ settle(batch): order → conflicts → resolve → for each accepted event:
                    Job: arena ← EXT(state), EXT(msg), EXT(env) ; REF(step) applied
                         reduce (canonical scheduler, mana-metered)
                         readback root → Op5 constructor
                    Op5 interpreter: Done | Call(nested job) | Ask(suspend) | Emit | Sys | Fail
                    commit new state Val (or revert)  ─► Receipt
          ─► Outputs: WalAppend, Reply, Broadcast, EffectRequest, SnapshotHint
```

4.3 PROCESS MODEL INSIDE ONE NODE (H5D)

```
main ─┬─ kernel actor (owns Kernel; single writer; loops on Chan(Cmd))
      ├─ http listener (TCP.listen; IO.spawn per connection; talks to actor via Chan)
      ├─ gossip loop (every gossip_ms: pull heads from peers, fetch missing events)
      ├─ snapshot loop (every snapshot_every checkpoints: +kernel share → serialize → store)
      ├─ effect executor (performs Ask effects this node is designated for → EffRes events)
      └─ roles (optional: orderer producer, e.g. dev-orderer emits Checkpoints)
```

The kernel actor is the **only** place state changes. Everything else sends `Cmd` messages and waits for replies on per-request channels.

---

5. PACKAGES AND REPOSITORY LAYOUT

5.1 THE PACKAGE SPLIT (READ THIS FIRST)

This spec is the spec of the `hvm5` repository: the runtime, and nothing more. It plays the role Kindelia's repo plays for Kindelia, which ships the VM, the node, the API, the CLI and a genesis file, not the apps people deploy. Everything that *runs inside* HVM5, and everything app authors import, lives in **separate Bend hub packages**. They are versioned and published on their own, and each one is pinned by content hash, following the convention of `github.com/Giulio2002/bend-collections`.

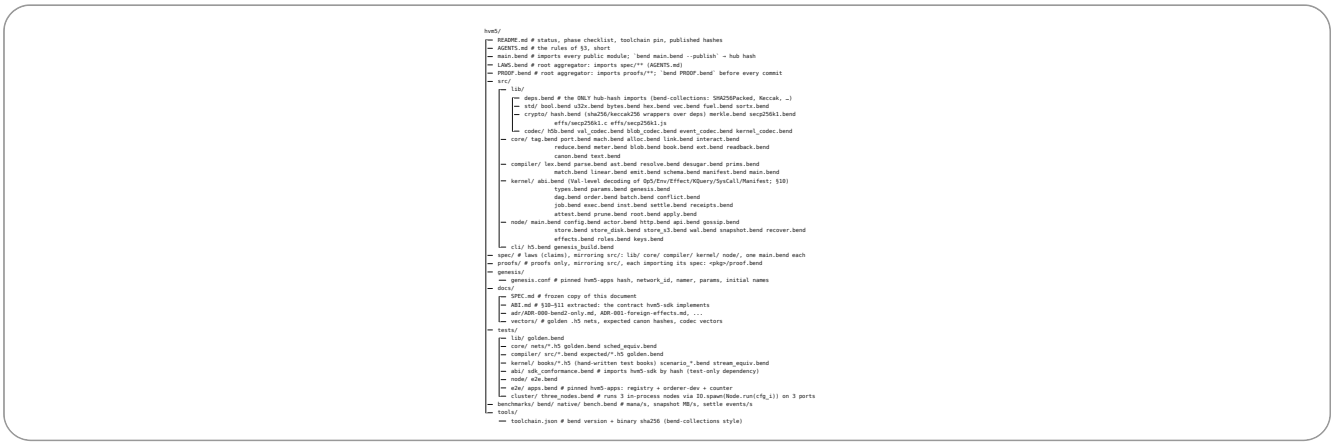
Package (repo)	Contents	Depends on	Specified in
`hvm5` (this spec)	L0 runtime-private libs, L1 core machine, L2 compiler `h5c`, L4 kernel, L5 node + CLI, genesis *builder*	`bend-collections` (hub), Base	this document, completely
`hvm5-sdk`	L3: `h5.bend` (Val, coerce, Addr, Env, Op5, IO5, Effect, KQuery, SysCall, Manifest, Schema, Decision) + `prelude.bend` (h5c subset of Base)	Base only	§10–§11 define its **ABI** normatively; the package just implements it
`hvm5-apps`	L6: `registry/`, `orderer-dev/`, `orderer-quorum/`, `orderer-pow/`, examples `counter/`, `kv-crdt/`, `bank/`	`hvm5-sdk` (hub hash)	§14, §15, §25 define their *behavior*; separate package specs own the code
`bend-collections` (existing, external)	`SHA256Packed`, `Keccak`, `HashMap`, `Bitset`, `W64` …	Base	upstream README; pinned by hash

Rules:

- **`hvm5` never imports `hvm5-sdk` or `hvm5-apps` in `src/`.** The kernel speaks to programs only through the **Val-level ABI** (§10, constructor tags of `Op5`, `Env`, `Effect`, `KQuery`, `SysCall`), which lives in `src/kernel/abi.bend`. The SDK is the typed mirror of that ABI. Conformance is checked by one test, `tests/abi/sdk_conformance.bend`, which imports the SDK by hub hash and asserts, for every constructor, that `h5c`-compiled SDK values read back as the kernel's ABI decoding expects.
- **Genesis is data, not code.** The runtime repo contains `genesis/genesis.conf`, which lists the pinned `hvm5-apps` release hash plus `network_id`, `namer`, `params` and initial names. `h5 genesis build` fetches the app sources by hash, checks them (`bend …/PROOF.bend`), compiles them with `h5c`, and writes `genesis.h5b`, a release artifact with a published hash. This is Kindelia's `genesis.kdl`, but *referencing* apps instead of inlining them.
- **Runtime tests do not depend on the apps package.** Kernel tests use tiny hand-written books in `.h5` text (`tests/kernel/books/*.h5`: a counter, a test registry, a test orderer). The real registry and orderers are tested in `hvm5-apps`, and together with the runtime only in `tests/e2e/` (phase 7+), against a pinned hash.
- **Shared pure libraries come from `bend-collections`**, never copied: SHA-256 (`src/crypto/sha/packed/sha256.bend`: `sha256(words: Array<U32>, byte_length: Nat) -> Maybe<&1, Array<U32>>>`) and Keccak-256 (`src/crypto/keccak/keccak.bend`: `keccak256(a: Array<U32>, length: Nat) -> Maybe<&1, Array<U32>>>`). Both are proven against executable FIPS / sponge specs upstream. Pin the release hash from its README (at the time of writing: `import 0x9ee2e9a299991dcc089fe22c7f3ceb5f/src/crypto/keccak/keccak.bend as Keccak`) in `src/lib/deps.bend`, the **single** file that contains hub hashes. Everything else imports `deps.bend`.
- **Toolchain:** `bend-collections` pins **Bend 2.0.28** (`tools/toolchain.json`, binary sha256). `hvm5` pins the same version or newer (`bend update`) in its own `tools/toolchain.json`.

5.2 HVM5 REPOSITORY LAYOUT (BEND-COLLECTIONS CONVENTIONS: SRC/ · SPEC/ · PROOFS/ · TESTS/ · TOOLS/)

Path convention for the rest of this document: module paths such as `core/x.bend`, `kernel/x.bend`, `node/x.bend`, `compiler/x.bend`, `lib/x.bend` and `cli/x.bend` are relative to `src/`. Laws for `src/<pkg>/…` live in `spec/<pkg>/…`, and their proofs in `proofs/<pkg>/…`. " `<pkg>/LAWS.bend` " in later sections means `spec/<pkg>/main.bend`, and " `bend <pkg>/PROOF.bend` " means `bend proofs/<pkg>/proof.bend`.



5.3 SIBLING PACKAGES (OUTLINE ONLY; EACH HAS ITS OWN SPEC AND REPO)

```
hvm5-sdk/                          hvm5-apps/
├── main.bend LAWS.bend PROOF.bend ├── main.bend LAWS.bend PROOF.bend
├── src/h5.bend src/prelude.bend   ├── src/registry/ src/orderer-dev/ src/orderer-quorum/
├── spec/ proofs/                  │   src/orderer-pow/ (each: main.bend [+ role.bend])
└── tools/toolchain.json           ├── examples/counter/ examples/kv-crdt/ examples/bank/
                                   ├── spec/ proofs/ # registry laws of §14, orderer laws of §15
                                   └── tools/toolchain.json # imports hvm5-sdk by hash
```

`role.bend` files (orderer *producers*, IO programs) are packaged with their app, but they are run by `h5d` through `--role <name>`. `h5d` loads them from the pinned `hvm5-apps` hash that genesis names, so the node binary contains no orderer logic.

5.4 BUILD/RUN COMMANDS (THE ONLY COMMANDS YOU RUN IN `HVM5`)

```
bend PROOF.bend                          # whole-repo law gate (AGENTS.md: before every commit)
bend proofs/<pkg>/proof.bend             # per-package law gate
bend tests/core/golden.bend              # golden tests (main returns IO, prints PASS/FAIL)
bend src/node/main.bend -o h5d           # build node daemon (native, BendRT)
bend src/cli/h5.bend -o h5               # build CLI
./h5 genesis build genesis/genesis.conf -o genesis.h5b
./h5d --data ./data --port 7070 --genesis genesis.h5b
bend main.bend --publish                 # release: prints the hub import line / hash
```

---

6. L1 — THE HVM5 CORE MACHINE (`CORE/`)

The core is a strict interaction-net reducer in the HVM2 family, with three additions: **labelled DUPs** (needed for correct forking), **`EXT`** (lazy external data), and **book-slot REFs**. Target size is **≤ 2,000 lines of Bend2**, including laws.

6.1 TAGS (`CORE/TAG.BEND`)

```
type Tag is Data:
  VAR{}   # 0  wire end; val = var id
  ERA{}   # 1  eraser (nullary)
  NUM{}   # 2  unboxed U32 (nullary); val = number
  REF{}   # 3  top-level def (nullary); lab = book slot, val = def index
  EXT{}   # 4  lazy external Val (nullary); val = ext-table index
  CON{}   # 5  constructor: lambda / application / pair (binary)
  DUP{}   # 6  duplicator / superposition (binary); lab = label
  OPR{}   # 7  numeric operator (binary); lab = opcode (+32 when partial)
  SWI{}   # 8  numeric switch 0 / succ (binary)
  NIL{}   # 15 "empty" sentinel (vars table / free slots); never in a live net
```

`Tag.to_u32` / `Tag.of_u32` convert to and from the numbers above. **Kind**: `VAR` = wire; `ERA NUM REF EXT` = nullary; `CON DUP OPR SWI` = binary. Nullary and binary ports are *principal* ports. The principal of a binary node is identified by a port whose `val` is the node's location.

6.2 PORTS AND MEMORY LAYOUT (`CORE/PORT.BEND`, `CORE/MACH.BEND`)

```
type Port is Data:
  Port{tag: Tag, lab: U32, val: U32}      # lab < 2^28
```

Packed form (heap words, `Array<U32>`): **a port = 2 words**: `w0 = tag + 16 * lab`, `w1 = val`. **A binary node = 2 aux ports = 4 words** at `heap[4*loc + 0..3]`. The principal is never stored: it is the pointer. `loc 0` is reserved (never allocated) so that `val = 0` is never a live node.

```
type Mach is Type:
 Mach{
   heap:  Array<U32>,   # 4 words / node
   vars:  Array<U32>,   # 2 words / var (a Port or NIL)
   ncap:  U32, nnext: U32, nfree: U32,   # node capacity, bump pointer, free-list head (0 = none)
   vcap:  U32, vnext: U32, vfree: U32,   # same for vars
   live:  U32                            # live node count (for arena cap + stats)
 }
```

- **Allocation** (`core/alloc.bend`): if `nfree != 0` , pop the free list (the next pointer is stored in `heap[4*loc]`). Otherwise use `nnext++` . If `nnext == ncap` , **grow**: new array of twice the capacity, copy with a fuel loop (§6.8). If growing would exceed `Params.arena_nodes_max` , the job halts with `ArenaFull` . Freeing pushes onto the free list. Vars use the same scheme. **The free list is LIFO**, and that choice is part of the canonical scheduler (§6.9).
- Addresses and var ids are **not semantic**. Only `canon` (§6.12) is observable.

6.3 WIRES: `LINK` (`CORE/LINK.BEND`)

A wire between two aux slots is represented by both slots holding `VAR v` . The `vars` table records what the *other* end has since been connected to.

```
link(a, b):                      # a, b are ports
  if a.tag == VAR: bind(a.val, b); return
  if b.tag == VAR: bind(b.val, a); return
  push_redex(a, b)               # both principal → active pair

bind(v, p):
  old = vars[v]
  if old == NIL: vars[v] = p          # first end resolved; wait for second
  else:          vars[v] = NIL; free_var(v); link(old, p)
```

`deref(p)` : while `p` is `VAR` and `vars[p.val] != NIL` , set `p = vars[p.val]` . Readback and canon use it. `link` recursion depth is bounded by the length of a var chain, so use a fuel of `vcap` levels (§6.8). **Slot contents never change after a node is created.** This invariant is what makes the rule choice order-independent. `link` writes only into `vars` and the redex bag.

6.4 THE REDEX BAG AND THE CANONICAL SCHEDULER

`rdx: List<&2, Rdx>` with `type Rdx is Data: Rdx{a: Port, b: Port}` . **Canonical scheduler = LIFO stack**: push to the head, pop the head, reduce until empty. Within one interaction, new redexes are pushed in the order they are listed in §6.6.

6.4.1 DETERMINISTIC PARALLEL SCHEDULER: `WAVES` (PHASE 10, ADR-020)

Base has `Array.fork/join` but **no atomics** (`bend base Array`), so HVM2's lock-free `atomic exchange` on vars is not available. `Waves` is a scheduler that is parallel **and** gives the same result for any thread count:

1. **Collect.** Take the whole bag as wave `w` . Normalize each pair (§6.5) and sort the pairs by `(min loc, max loc)` (ports carry locations, and locations are deterministic because the previous waves were).
2. **Plan (pure).** For each pair `i` , the rule fixes its allocation need `n_i` (nodes) and `v_i` (vars) *before* rewriting: COMM 4/4, CALL `len(nodes)` / `nvars` , EXPD one layer, and 0 for the rest. Prefix sums give each pair a disjoint range: node locations are taken, in order, from `pool = sorted(free list) ++ bump region` , and vars likewise. **IDs therefore do not depend on the thread count or the timing.**
3. **Rewrite (parallel).** Split the sorted pairs in halves recursively (balanced `a b = f(x) g(y)`). Each leaf writes only into its own fresh ranges, and it emits **link intents** `(port, port)` instead of calling `link` . Frees are emitted as lists. This step has no shared writes, so it needs no atomics.
4. **Resolve (sequential, cheap).** Apply all link intents in pair order through the ordinary `link` (§6.3). This yields the next bag in a deterministic order, and freed locations are merged into the sorted free list. LINK costs 0 mana, so moving it out of the parallel part changes no cost.

Mana per wave is the sum over its pairs, so by Claim B the total is equal to LIFO's. Halts (`OutOfMana` , `Fault` , `ArenaFull`) are checked **per wave**, before step 3, against that wave's planned totals. The outcome of `Waves` is therefore a function of the net alone. Genesis `Params.scheduler: Lifo | Waves` picks the committed scheduler. The default is `Lifo` until `tests/core/sched_equiv.bend` proves `Waves ≡ Lifo` (canon + mana) on the golden set. After that, `Waves` may become canonical by genesis, and nodes need no canonical rerun on `ArenaFull` , because both schedulers are deterministic.

6.5 RULE SELECTION

Normalize each pair so that `rank(a.tag) ≤ rank(b.tag)`, with rank order `ERA < NUM < REF < EXT < CON < DUP < OPR < SWI`. (A `VAR` never appears in the bag.)

a \ b	ERA	NUM	REF	EXT	CON	DUP	OPR	SWI
ERA	VOID	VOID	VOID	VOID	ERAS	ERAS	ERAS	ERAS
NUM		FAULT	FAULT	FAULT	COPY	COPY	OPER	SWIT
REF			FAULT	FAULT	CALL	COPY if `def.safe` else CALL	CALL	CALL
EXT				FAULT	EXPD	COPY	EXPD	EXPD
CON					ANNI	COMM	COMM	COMM
DUP						ANNI if same `lab`, else COMM	COMM	COMM
OPR							ANNI if same `lab`, else COMM	COMM
SWI								ANNI

The table is **total**: every pair maps to exactly one entry, so a hostile net cannot crash the machine. Two nullary *values* meeting (`NUM ⋈ NUM`, `REF ⋈ EXT`, …) cannot come from a well-typed program, so it is **FAULT**, not a silent VOID. FAULT halts the job with `Fault{NoRule(tag_a, tag_b)}`, the event reverts, and the receipt charges the event's full `mana_limit`. That charge is deterministic, and it punishes hostile nets. The halt itself is scheduler-independent: by Claim B (§6.9), the FAULT pair is in the interaction multiset under every schedule. The remaining odd pairs (for example NUM ⋈ CON = COPY, CON ⋈ SWI = COMM) are legal net rewrites and stay as they are.

6.6 THE 11 INTERACTION RULES (EXACT REWIRING)

Notation: `B[b0,b1]` means binary node `B` at `loc` with aux ports `b0 = slot 0`, `b1 = slot 1` (read from the heap). `new X(p, q)` allocates a node with aux `p, q` and returns its principal port. `fresh()` allocates a var and returns `VAR v`.

1. **VOID** (`ERA ⋈ nullary`): nothing. Both vanish. (EXT/REF have no heap to free.) Any other nullary ⋈ nullary pair is FAULT (§6.5, item 11).
2. **ERAS** (`ERA ⋈ B[b0,b1]`): `free(B)`, `link(b0, ERA)`, `link(b1, ERA)`.
3. **COPY** (`N ⋈ B[b0,b1]`, N nullary ∈ {NUM, REF(safe), EXT}, B ∈ {DUP} or N=NUM with B=CON): `free(B)`, `link(b0, N)`, `link(b1, N)`. (For EXT, both copies point at the same ext index. The Val is immutable, so sharing is sound.)
4. **ANNI** (`A[a0,a1] ⋈ B[b0,b1]`, same tag and same lab): `free(A)`, `free(B)`, `link(a0, b0)`, `link(a1, b1)`.
5. **COMM** (`A[a0,a1] ⋈ B[b0,b1]`, different tag or different lab):

```
v00 v01 v10 v11 = fresh() ×4
B0 = new B.tag/B.lab (v00, v01)      B1 = new B.tag/B.lab (v10, v11)
A0 = new A.tag/A.lab (v00, v10)      A1 = new A.tag/A.lab (v01, v11)
free(A); free(B)
link(a0, B0); link(a1, B1); link(b0, A0); link(b1, A1)
```

   (Net growth: +2 nodes.)
6. **CALL** (`REF(slot, idx) ⋈ X`): `def = scope[slot].defs[idx]`, `root = instantiate(def.blob)` (§6.10), then `link(root, X)`.
7. **EXPD** (`EXT(i) ⋈ X`): `v = exts[i]`, then:
   - `VNum{n}` → `link(NUM n, X)`
   - `VCtr{tag, fields}` → build the tagged chain (§7.2): `c = new CON(NUM tag, chain(fields))`, where `chain([]) = ERA` and `chain(f::fs) = new CON(EXT(push_ext f), chain(fs))`; then `link(c, X)`. **Only one layer is built. Children remain `EXT`.**
   - `VNet{blob}` → `root = instantiate(blob)`; `link(root, X)`.
8. **OPER** (`NUM(x) ⋈ OPR[b0,b1]`, `op = lab mod 32`, `partial = lab ≥ 32`):
   - not partial, `b0` is `NUM(y)` (literally in the slot): `free(OPR)`, `link(b1, NUM(op(x,y)))`.
   - not partial, `b0` not NUM: `free(OPR)`; `P = new OPR(lab=op+32)(NUM x, b1)`; `link(b0, P)`.

- partial (slot `b0` holds `NUM(first)`): `free(OPR)`, `link(b1, NUM(op(first, x)))`.

9. **SWIT** (`NUM(n) ⋈ SWI[b0,b1]`), where `b0` is the branch pair `(Z, S)` and `b1` the result:
   - `n == 0`: `free(SWI)`; `P = new CON(b1, ERA)`; `link(b0, P)`. (Z→result, S erased)
   - `n > 0`: `free(SWI)`; `A = new CON(NUM(n-1), b1)`; `P = new CON(ERA, A)`; `link(b0, P)`. (Z erased, S applied to n-1)
10. **LINK** is not a rule in the table. It is the var binding of §6.3. **Cost 0.**
11. **HALT** conditions (checked before each pop): mana exhausted → `OutOfMana`; arena cap → `ArenaFull`; fuel exhausted → `OutOfFuel` (only possible if params are misconfigured); a popped pair whose table entry is FAULT → `Fault{NoRule(a, b)}`. Every halt reverts the event.

Opcodes (`lab mod 32`), all on U32 with wraparound, results as NUM: `0 add, 1 sub, 2 mul, 3 div (x/0 = 0), 4 mod (x%0 = x), 5 eq, 6 ne, 7 lt, 8 le, 9 gt, 10 ge, 11 and, 12 or, 13 xor, 14 shl (y mod 32), 15 shr (y mod 32)`. Comparisons return 0 or 1. This matches the enum encoding of `Bool` (§7.2), so `U32.is_eq` compiles to a single OPR.

6.7 METERING (`CORE/METER.BEND`)

Rule	mana
LINK	0
VOID	1
ERAS	2
COPY	2
FAULT	the event's whole `mana_limit`
ANNI	2
COMM	4
OPER	2 (each step, partial or final)
SWIT	2
CALL	2 + nodes(def) + redexes(def)
EXPD	2 + number of nodes built

Every job has `mana_limit` (§21). The job keeps `Stats{counts per rule, mana, peak_live}`. The **bits** charge is taken at readback, not during reduction: `bits = 64 × (number of new Val nodes built by readback)`. It is deterministic because readback is a function of the normal form.

6.8 LOOPS THAT MUST TERMINATE: THE BINARY FUEL TREE (`LIB/STD/FUEL.BEND`)

Bend2 requires structural termination, and `Nat` is unary (4,000,000n is a huge list). Use a depth-indexed fork of calls: depth `d` gives up to `2^d` steps with only `d` Succ nodes, and it stops early when the machine halts.

```
# Job is Type; its first field is the status so we can match on it.
def Red.go(d: Nat, job: Job) -> Job:
  match d job:
    case 0n j:
      Red.one(j)                                  # pop one redex, interact, meter
    case 1n+p Job{Running{}, m}:
      Red.go(p, Red.go(p, Job{Running{}, m}))     # both calls: first arg p is smaller
    case 1n+p Job{st, m}:
      Job{st, m}                                  # halted: short-circuit
```

`Red.one` sets the status to `Done{}` when the bag is empty. Use `d = 40n` for reduction and `d = 32n` for copy/link loops. The **only** `@unsafe` defs allowed in pure layers are ones this document names. There are none in `core/`.

6.9 DETERMINISM THEOREM (AND EXACTLY WHERE IT STOPS)

- **Claim A (result):** for a net that reaches normal form, `canon(normal form)` is the same under every scheduler. Source: the rules are local, the table is total, slot contents are static, and each pair has exactly one rule (strong confluence, as in HVM2).
- **Claim B (cost):** the multiset of non-LINK interactions is scheduler-independent, so mana is the same everywhere.
- **Limit:** `peak_live` and `ArenaFull` *are* order-dependent. **Rule:** the committed outcome of a job is *defined* as the outcome under the canonical LIFO scheduler with `Params.arena_nodes_max`. A node running a parallel scheduler that hits a limit must rerun canonically. Mana exhaustion needs no rerun, by Claim B. `OutOfMana` always reverts the whole event, so a partial state never escapes.
- **What confluence does NOT give (never claim it does):** confluence makes *one net, given the same inputs* reduce to the same result. It does **not** decide *which* inputs exist or in what order (that is **consensus**, done by orderer apps, §15). It does not make inputs *available* (a withheld event stalls everyone who depends on it: **data availability**, handled by gossip plus the orderer's rule of only checkpointing heads it holds). It does not make an app's step commutative (that needs the `commutes` proof, §13.7). And it gives no liveness, fairness or termination without mana. "Nodes sync reductions" is therefore sound **only after** the event cut is agreed. Attestations (§16) detect bugs and divergence; they do not replace ordering.
- **Tests:** `tests/core/sched_equiv.bend` reduces every golden net under LIFO and under FIFO, and asserts equal canon hash and equal mana. It also states these as concrete laws in `spec/core/`.

6.10 BLOBS AND INSTANTIATION (`CORE/BLOB.BEND`)

A **NetBlob** is a relocatable, closed net: definitions, closures inside state, and continuations.

```
type BPort is Data:
  BVar{id: U32}                # local var id (each appears exactly twice in the blob)
  BEra{}
  BNum{n: U32}
  BRef{dep: U32, idx: U32}     # dep 0 = same book, k = book.deps[k-1]
  BVal{i: U32}                 # blob.vals[i] → becomes EXT at instantiation
  BPtr{i: U32}                 # principal of blob node i

type BNode is Data:
  BNode{tag: Tag, lab: U32, a: BPort, b: BPort}

type NetBlob is Data:
  NetBlob{root: BPort, nodes: List<&2, BNode>, rdx: List<&2, BPair>,
          nvars: U32, refs: List<&2, H256>, vals: List<&2, Val>}
type BPair is Data:
  BPair{a: BPort, b: BPort}
```

- In a **Def** blob, `refs` is empty and `BRef.dep` indexes the book's deps. In a **VNet** blob (captured at readback), refs are rewritten to absolute form: `refs[j]` = book hash, and `BRef{dep = j+1, idx}` indexes into `refs`. VNet blobs are therefore relocatable across books and upgrades.
- `instantiate(blob)`: allocate `len(nodes)` nodes and record `loc_i`; allocate `nvars` fresh vars; convert ports (`BPtr i → Port{tag_i, lab_i, loc_i}`, `BVar j → VAR var_j`, `BRef → REF{lab = slot_of(dep), val = idx}`, `BVal i → EXT{val = push_ext(vals[i])}`); write each node's aux ports into the heap; `link` each blob redex; return the converted root. Cost: see CALL/EXPD.
- `slot_of`: each job has a **RefScope** (`Vec<Book>` of the books in scope, indexed by slot). The first entry is the instance's book, then its deps in BFS order; VNet `refs` books are appended on demand. The kernel guarantees that every referenced book is present (refcounts, §12.7).

6.11 READBACK (`CORE/READBACK.BEND`)

`readback(port) -> Result<ReadErr, Val>` after `deref`:

Port / node shape	Val
`NUM n`	`VNum{n}`
`EXT i`	`exts[i]` (**shared pointer, no copy**)
`CON` with slot0 = `NUM tag` and slot1 a chain of `CON(field, next)` ending in `ERA`	`VCtr{tag, [readback(field)...]}`

Port / node shape	Val
anything else (λ, DUP, OPR, SWI, REF, unmatched shapes)	`VNet{capture(port)}` if the reachable subnet is **closed**, else `Fail{NotClosed}`

`capture` walks the reachable subnet (following aux slots and `deref`). It numbers nodes and vars in first-visit DFS order (slot 0 before slot 1). Vars seen exactly twice become internal `BVar`. A var seen once means the subnet is not closed. EXT ports become `BVal`. The walk uses fuel. Readback builds fresh Val nodes only for materialized structure. It counts them for bits (§6.7).

6.12 CANONICAL FORM AND HASHING (`CORE/CANON.BEND`)

`canon(port)` = H5B bytes (§22) of `capture(port)` with DFS numbering. `hash = sha256(canon)`. Because DFS numbering is order-independent given the normal form, Claim A makes the hash scheduler-independent. Val hashing is defined in §7.3. It is **not** the same as net canon; Val hashes are what the kernel uses.

6.13 TEXT FORMAT `.H5` (`CORE/TEXT.BEND`)

Used by tests, debugging and `h5 text`. It is close to HVM2 syntax.

```
book  := def*
def   := '@' name '=' tree ('&' tree '~' tree)*
tree  := var | '*' | '#' u32 | '@' name | '(' tree tree ')'            # CON
       | '{' tree tree '}' | '{' ':' u32 tree tree '}'                 # DUP lab 0 | lab n
       | '$' op '(' tree tree ')' | '?' '(' tree tree ')'              # OPR | SWI
var   := [a-z_][a-z0-9_]*        # each var occurs exactly twice
op    := add|sub|mul|div|mod|eq|ne|lt|le|gt|ge|and|or|xor|shl|shr
```

Example (`@main = 2 + 3`): `@main = r & #2 ~ $add(#3 r)`. `Text.parse : String -> Result<String, Book>` and `Text.show : Book -> String` must round-trip on every golden file (law `text_roundtrip_<name>`).

6.14 CORE PUBLIC API (THE ONLY EXPORTS OTHER LAYERS MAY USE)

```
def Core.run(scope: RefScope, exts: Vec<Val>, build: Build, limits: Limits) -> JobOut
type Build is Data:                          # how to form the initial net
  Apply{fn: BPort, args: List<&2, Val>}      # (fn arg1 ... argN) → root
  Inst{blob: NetBlob, args: List<&2, Val>}   # instantiate blob, then apply to args
type JobOut is Data:
  JobOut{status: Halt, result: Maybe<&2, Val>, stats: Stats, bits: U32}
type Halt is Data:
  Done{} OutOfMana{} ArenaFull{} OutOfFuel{} NotClosed{}
def Core.canon(v: Val) -> Bytes                # for VNet: blob canon; for data: §7.3 encoding
def Text.parse(s: String) -> Result<&2, &2, String, Book>
```

`Apply{fn, args}` builds the application chain: `a1 = CON(EXT arg1, a2)`, …, `aN = CON(EXT argN, VAR root)`, pushes the redex `(fn, a1)`, reduces, then does `readback(deref(VAR root))`.

---

7. VALUES AND DATA ENCODING

7.1 VAL (KERNEL-SIDE, PERSISTENT, HASH-CONSED) — `CORE/EXT.BEND`

```
type H256 is Data:
  H256{a: U32, b: U32, c: U32, d: U32, e: U32, f: U32, g: U32, h: U32}

type VMeta is Data:
  VMeta{size: U32, hash: Maybe<&2, H256>}     # size = node count of subtree; hash filled lazily

type Val is Data:
  VNum{n: U32}
  VCtr{tag: U32, fields: List<&2, Val>, meta: VMeta}
  VNet{blob: NetBlob, meta: VMeta}
```

- `Val` is `Data`, so `+v` sharing is a refcount and forking is O(1).

- `meta.hash = None` on nodes that readback just built. `Val.seal(v) -> Val` fills the hashes bottom-up and **in parallel** over fields (§24). Sealed subtrees are skipped. The kernel seals states at commit time, before computing receipts.

7.2 CONSTRUCTOR ENCODING IN NETS (`H5C` AND EXPD MUST AGREE EXACTLY)

For each `type T` in declaration order, constructors get `tag = 0, 1, 2, …`.

Type shape	Net encoding
enum (all ctors have 0 fields, e.g. `Bool`, `Cmp`)	`NUM(tag)`. So `False = #0`, `True = #1`
newtype (1 ctor with 1 field, e.g. `Char = Chr{code}`)	the field itself (erased wrapper)
`U32`	`NUM` (intrinsic)
general	tagged chain `(#tag (f1 (f2 (… (fn *)))))`; nullary ctor = `(#tag *)`

The Val form mirrors this: enum → `VNum{tag}`, newtype → field Val, general → `VCtr{tag, fields}`. **Schemas** (§11.2) record which encoding each type uses. `Nat` is general (`Zero` / `Succ`) and therefore slow, so apps should use `U32`.

7.3 VAL HASH

```
hash(VNum n)            = sha256( 0x00 ‖ u32be(n) )
hash(VCtr tag fs)       = sha256( 0x01 ‖ u32be(tag) ‖ u32be(len fs) ‖ hash(f1) ‖ … )
hash(VNet blob)         = sha256( 0x02 ‖ canon_blob(blob) )     # blob vals hashed inside canon by their hash
```

---

8. BOOKS (`CORE/BOOK.BEND`)

```
type Def is Data:
  Def{name: String, blob: NetBlob, safe: Bool}      # safe = blob contains no DUP (COPY-able REF)

type CtorInfo is Data:
  CtorInfo{type_name: String, name: String, tag: U32, arity: U32, enc: Enc}
type Enc is Data:
  EEnum{} ENewtype{} EGeneral{}

type Book is Data:
  Book{
    abi: U32,                    # SDK ABI version (§10); h5c copies it from the SDK's `h5_abi()` constant;
                                 # the kernel rejects books whose abi ∉ Params.abi_versions
    deps: List<&2, H256>,        # books this book's BRef{dep>0} point into
    defs: List<&2, Def>,
    ctors: List<&2, CtorInfo>,   # for CLI pretty-printing and parsing of Val text
    schemas: List<&2, Schema>,   # schema table (§11.2)
    manifest: Manifest           # §11.1
  }
```

`book_hash = sha256(H5B(book))`. The book is identified only by this hash. Defs are referenced internally by `(dep, idx)`, so recursive defs never create hash cycles. Books are stored in the **CAS** (`objects/<hex>`). `defs` is converted to a `Vec<Def>` (`lib/std/vec.bend`: a persistent binary trie indexed by U32 bits, O(log n) get) when loaded.

---

9. L2 — THE COMPILER `H5C` (`COMPILER/`)

`h5c` is a Bend2 program. It reads Bend2 source text that has **already passed** `bend <file> --check-only` (and `bend PROOF.bend`), and emits a `Book`. It ignores types, except to learn arity, erasure and constructor order.

9.1 ACCEPTED SUBSET (ANYTHING ELSE → COMPILE ERROR WITH LINE/COL)

- `import Base` (maps to `prelude.bend` of the `hvm5-sdk` version the app imports by hub hash, plus primitives; `h5c` fetches it through the same hub cache as `bend`, and fails if the app does not import an SDK), `import 0x<hash>/… as M` (hub packages, same subset; h5c verifies the hash), `import ./x.bend as M` (recursive, same subset), `import 0x…/…` (resolved from the local CAS as an already-compiled book dep).

- `type … is Data|Type:` with constructors.
- `def f(params) -> T:` with bodies made of: variables, literals (`U32` , `Nat 3n` , `Char` , `String`), constructors `K{..}` , lists `[a,b]` , `h <> t` , tuples `(a,b)` , calls `f(a)` , lambdas `x => e` , `+x => e` , operators inside `(… : U32)` and `&& || ++` , lets `x = v` , `+x = v` , `-x = v` , destructuring lets `(a, b) = v` , `K{x,y} = v` , parallel lets `a b = f(x) g(y)` (treated as two lets: nets are parallel anyway), `match a b:` with `case` patterns (constructors, `_` , nested one level, `0n` , `1n+p`), `do M<…>:` blocks (desugared to `M.bind` / `M.pure` exactly as the guide describes), templates `~f` (inlined per call site), `@unsafe` (accepted, with a warning).
- Ignored and erased: `law` blocks, proof defs whose return type is an equality, `-x` params and args, `{==}` , `%e : P` , type annotations `{x : T}` .
- **Rejected**: `Array` , `F32` , `!` calls, `IO` / foreign `import "./x.c"` , `?holes` .

9.2 PASSES

#	Pass	Output
1	`lex → parse`	AST (`compiler/ast.bend`) with source positions
2	`resolve`	Fully qualified names; every def's param list with quantity (`-` /plain/ `+` / `~`); ctor table (tags by declaration order, arity, encoding kind)
3	`desugar`	Remove `do` , operators → prim calls, strings → `SCon` / `SNil` chains of `Char` (newtype → NUM), lists → `Con` / `Nil` , tuples → `Tuple` , drop erased args by **position** from the signature table, inline templates
4	`prims`	`U32.add…shr` , `U32.is_eq…is_ge` → OPR nodes; `Bool.and/or/not` stay ordinary defs from prelude
5	`match`	Match linearization (§9.3)
6	`linear`	Every variable used exactly once: 0 uses → ERA; n>1 uses → a balanced tree of `DUP lab 0` nodes
7	`emit`	Core → `NetBlob` per def: λ = `CON(binder, body)` ; application `f a = CON(a, ret)` wired to `f` ; ctor = §7.2; top-level refs = `BRef` ; compute `safe`
8	`schema + manifest`	See §11. `h5_manifest()` is compiled and **evaluated with `Core.run`** , then read back and decoded into `Manifest` (dogfooding)

9.3 MATCH COMPILATION (GENERAL TYPES)

`match x: case K0{a,b}: e0 … case Kn{…}: en` , with free variables `v1..vk` used in the branches:

```
( (x ~ CON(t, rest)) ;                                  # destructure (#tag, rest)
  ?t( λrest λv1..λvk. let (a,(b,*)) = rest; e0 ,         # SWI: zero branch
      λp. ?p( λrest λv…. e1 , λp. … ) ) )               # succ branches, nested
  rest v1 … vk )                                        # apply the chosen branch
```

- The free variables are **passed as arguments** to the selected branch. Unused branches are erased together with their captures, so no DUP is needed across branches (Bend1's `linearize_matches`).
- For the last constructor, the succ branch ignores `p` (it is erased).
- Enum matches skip the destructuring: `?x(…)` directly.
- Multi-scrutinee `match a b:` compiles into nested single matches, left to right.
- `Nat` patterns: `0n` → `Zero` , `1n+p` → `Succ{p}` , `2n+p` → nested.

9.4 KNOWN LIMIT (DOCUMENT IT; DO NOT WORK AROUND IT)

Duplicating a closure that itself duplicates (the "dup of a dup" problem) with the same label is unsound at the λ level. User DUPs all use label 0. System forks use labels ≥ 1, allocated deterministically as `label = 1 + (job_seq mod 2^27)` . `h5c` emits warning `W-DUPDUP` when a `+` variable of function type is used more than once.

9.5 CLI SURFACE

`h5 compile app/main.bend -o app.h5b [--text app.h5]` writes the book (H5B, §22) and optionally the `.h5` text. The book hash is printed.

---

10. L3 — THE SDK ABI (NORMATIVE HERE; IMPLEMENTED BY THE SEPARATE `hvm5-sdk` PACKAGE)

Ownership. The runtime owns the **ABI**: the constructor names, their declaration order (= ctor tags, §7.2), their field order, and their meaning. `src/kernel/abi.bend` decodes it at the Val level. The package `hvm5-sdk` (`src/h5.bend`, `src/prelude.bend`) is the *typed* mirror that apps import:

```
import 0x<hvm5-sdk-hash>/src/h5.bend as H     # apps pin an SDK release; never a relative path
```

An ABI change is a new `Book.abi` value (§8; the SDK exports `def h5_abi() -> U32`) that must be added to `Params.abi_versions` (the set the kernel accepts), so old books keep running. `tests/abi/sdk_conformance.bend` (§5.1) guards the mirror. The code below is the reference text of `hvm5-sdk/src/h5.bend`. Its *types* are the ABI; its helper defs are informative.

The SDK must pass **both** checkers: the official `bend` (types and proofs) and `h5c` (compilation). Where the kernel is untyped, the SDK uses a **postulate** (an open law, as Base does for `F32.add`) that `h5c` compiles to the identity.

```
# hvm5-sdk/src/h5.bend
import Base

def h5_abi() -> U32:            # ABI version this SDK implements; h5c copies it into Book.abi
  1

# ---- raw runtime value: opaque to Bend2, identity in h5c --------------------
law Val:
  Data
law H5.coerce:
  for -A: Type
  for -B: Type
  for x: A
  B

# ---- identity / environment -------------------------------------------------
type Addr is Data:
  Addr{a: U32, b: U32, c: U32, d: U32, e: U32, f: U32, g: U32, h: U32}

type Env is Data:
  Env{self: Addr, from: Addr, event: Addr, epoch: U32, time_ms: U32, lag: U32, seq: U32, pos: U32}
  # self    = instance id                      (Kindelia SUBJ: the current subject)
  # from    = event signer, or the calling instance inside a nested Call (Kindelia FROM)
  # event   = event id (Kindelia STH0/STH1: statement hash)
  # epoch   = orderer epoch; time_ms = checkpoint time (Kindelia TICK / TIME)
  # pos     = index of this event in the settled batch (Kindelia GIDX = tick<<60 | stmt_idx);
  #           (epoch, pos) is a unique, deterministic, totally ordered id
  # lag     = epoch - max parent epoch (staleness); seq = author's sequence number

# ---- the interpreter-facing operation tree (read back by the kernel) ----------
type Op5 is Type:
  Done{state: Val, reply: Val}
  Call{target: Addr, msg: Val, k: Val -> Op5}
  Ask{eff: Val, save: Val, k: Val -> Val -> Op5}       # k(answer)(fresh_state)
  Emit{topic: String, data: Val, next: Op5}
  Read{q: Val, k: Val -> Op5}                          # read-only kernel query (KQuery, §10.1)
  Sys{call: Val, k: Val -> Op5}                        # privileged (registry only)
  Fail{why: Val}

# ---- the monad apps actually write in (CPS, same shape as Base's IO) -------
def IO5(A):
  @k: (A -> Op5) -> Op5

def IO5.pure(-A: Type, x: A) -> IO5(A):
  k => k(x)

def IO5.bind(-A: Type, -B: Type, m: IO5(A), f: A -> IO5(B)) -> IO5(B):
  k => m(x => f(x, k))

def H5.call(-M: Data, -R: Data, target: Addr, msg: M) -> IO5(R):
  k => Call{target, H5.coerce(M, Val, msg), v => k(H5.coerce(Val, R, v))}

def H5.ask(-E: Data, -R: Data, -S: Data, eff: E, st: S) -> IO5(R & S):
  k => Ask{H5.coerce(E, Val, eff), H5.coerce(S, Val, st),
           v => s => k((H5.coerce(Val, R, v), H5.coerce(Val, S, s)))}

def H5.emit(-D: Data, topic: String, d: D) -> IO5(Unit):
  k => Emit{topic, H5.coerce(D, Val, d), k(Unit{})}

def H5.fail(-A: Type, -W: Data, why: W) -> IO5(A):
  k => Fail{H5.coerce(W, Val, why)}

# ---- the entry adapter every app exports as `h5_step` -----------------------
def H5.run(-S: Data, -R: Data, m: IO5(S & R)) -> Op5:
  m(sr => H5.done(S, R, sr))
def H5.done(-S: Data, -R: Data, sr: S & R) -> Op5:
  (s, r) = sr
  Done{H5.coerce(S, Val, s), H5.coerce(R, Val, r)}
```

An app then writes:

```
def step(s: State, m: Msg, e: H5.Env) -> H5.IO5(State & Reply): …
def h5_step(s: Val, m: Val, e: Val) -> H5.Op5:
  H5.run(State, Reply, step(H5.coerce(Val, State, s), H5.coerce(Val, Msg, m),
                            H5.coerce(Val, H5.Env, e)))
```

10.1 EFFECTS AND SYSTEM CALLS (SDK TYPES; THEY CROSS THE KERNEL AS VAL)

```
type Effect is Data:
  Now{}                                          # wall clock ms (executor's)
  Random{bytes: U32}
  HttpGet{host: String, port: U32, path: String}      # plain HTTP/1.1 over Base TCP
  HttpPost{host: String, port: U32, path: String, body: String}
  External{kind: String, data: String}               # pluggable executors (node/effects.bend)

type KQuery is Data:                                 # Op5.Read; deterministic, costs 10 mana each
  NameOf{name: String}                               # → Maybe inst id    (Kindelia GIDX-style lookup)
  PosOf{name: String}                                # → (epoch, pos) of the name's registration
  EventAt{epoch: U32, pos: U32}                      # → Maybe event id   (Kindelia STH0/STH1)
  InstInfo{inst: Addr}                               # → (book, owner, version) — never another instance's state
  Stats{}                                            # → (epoch, mana used this epoch, state size)

type SysCall is Data:                                # accepted only from instances holding SysCap
  Spawn{book: Addr, init: Val, owner: Addr, name: String}     # returns new instance id
  Upgrade{inst: Addr, book: Addr}                            # §12.6
  Fork{inst: Addr, book: Addr, name: String}                 # O(1) state share
  Bind{name: String, inst: Addr}                             # kernel name mirror
  Unbind{name: String}

type Decision is Data:                               # conflict resolution output (§13.5)
  Take{event: Addr}
  Drop{event: Addr}
  Retry{event: Addr}
```

---

11. MANIFEST AND SCHEMAS

11.1 MANIFEST (EVALUATED FROM THE APP'S `H5_MANIFEST()` AT COMPILE TIME)

```
type OrderMode is Data:
  Ordered{orderer: String}        # name of an orderer instance (resolved at spawn)
  Commutative{}                   # requires LAWS.bend law `commutes` (checked by bend PROOF.bend)

type ConflictPolicy is Data:
  FirstWins{}                     # take first in canonical order; others → Retry
  DropAll{}                       # discard all members of a conflict set; all → Retry
  Custom{}                        # call app def `h5_resolve`

type ExecPolicy is Data:
  AnyNode{}                       # first EffRes in canonical order wins
  Designated{node: Addr}          # only this node's EffRes is valid
  Author{}                        # the invoking event's author answers (client-side effect)

type Limits is Data:
  Limits{mana: U32, bits: U32, calls_depth: U32, ask_timeout_epochs: U32}

type Manifest is Data:
  Manifest{
    name: String, version: U32,
    state_type: String, msg_type: String, reply_type: String,
    init_type: String, query_type: String,
    mode: OrderMode, conflict: ConflictPolicy, exec: ExecPolicy,
    limits: Limits, isolated: Bool      # isolated = never uses Call (enables parallel batches)
  }
```

Required exported defs: `h5_init(args, env) -> Op5` (usually `Done{state, unit}`), `h5_step(state, msg, env) -> Op5`. Optional exports: `h5_view(state, query) -> Val` (pure), `h5_migrate(old_state) -> Val`, `h5_resolve(state, batch) -> List<Decision>`. `h5c` records their def indices in the book (`Book.entries`, which is part of `manifest` in the H5B encoding: `entries: List<&2, Entry>` with `Entry{name: String, idx: U32}`).

11.2 SCHEMA (INPUT VALIDATION; UNTRUSTED VALS NEVER REACH A MATCH UNCHECKED)

```
type SField is Data:
  FNum{}  FRef{ty: U32}  FAny{}          # FAny = any *data* Val (no VNet)
type SCtor is Data:
  SCtor{name: String, fields: List<&2, SField>}
type Schema is Data:
  Schema{name: String, enc: Enc, ctors: List<&2, SCtor>}
```

`Schema.check(table, ty, v) -> Bool` walks the Val with fuel `size(v)`. The kernel checks `msg` against `msg_type`, `init` against `init_type`, and `query` against `query_type`. Failure → `Rejected{BadInput}`. **VNet is never accepted from outside** (hostile closures are ruled out); VNet appears only in kernel-produced state and continuations.

---

12. L4 — THE KERNEL (KERNEL/)

12.1 DOMAIN TYPES (KERNEL/TYPES.BEND)

```
type Params is Data:
  Params{arena_nodes_max: U32, max_event_bytes: U32, max_book_bytes: U32,
         mana_max: U32, bits_max: U32, retain_epochs: U32, finality_depth: U32,
         history_keep: U32, max_batch: U32, max_orphans: U32,
         abi_versions: List<&2, U32>,                 # accepted Book.abi values (§8, §10); genesis: [1]
         scheduler: Sched}                            # committed scheduler (§6.4.1); genesis: Lifo{}
type Sched is Data:
  Lifo{}  Waves{}

type EvKind is Data:
  Deploy{book: H256, name: String, init: Val}          # sugar → Invoke(registry, Register…)
  Invoke{target: String, msg: Val, mana: U32}          # target = name or 0x<inst-id>
  EffRes{req: H256, answer: Val}
  Checkpoint{orderer: H256, prev: H256, heads: List<&2, H256>,
             epoch: U32, time_ms: U32, proof: Val}
  Attest{checkpoint: H256, kernel_root: H256, receipts_root: H256}

type EvBody is Data:
  EvBody{net: H256,                      # genesis hash (cross-cluster replay guard)
         parents: List<&2, H256>,        # 1..8 parents (known heads when created)
         author: H256, seq: U32,         # author = Addr recovered from sig (§19); seq strictly increasing
         keys: List<&2, String>,         # declared conflict keys (access list)
         kind: EvKind}

type Event is Data:
  Event{body: EvBody, sig: Bytes}      # 65-byte recoverable secp256k1 sig (r‖s‖v) over id; no pubkey
                                       # field: the signer is recovered, as in Kindelia signer_name(hash).
                                       # Validation requires recovered addr == body.author
# id = sha256("h5ev1" ‖ H5B(body))

type Stored is Data:
  Stored{ev: Event, id: H256, depth: U32, state: EvState}
type EvState is Data:
  Pending{} Settled{epoch: U32} Equivocal{} Orphan{}

type Dag is Data:
  Dag{events: Map<&2, Stored>, heads: Map<&2, Unit>, last_seq: Map<&2, U32>,
      orphans: Map<&2, Stored>, horizon_epoch: U32}

type PendingAsk is Data:
  PendingAsk{req: H256, eff: Val, k: Val, origin: H256, deadline: U32, exec: ExecPolicy}

type Instance is Data:
  Instance{id: H256, name: String, book: H256, owner: H256, state: Val,
           pending: Map<&2, PendingAsk>, version: U32, orderer: Maybe<&2, H256>,
           status: InstStatus, sys: Bool}
type InstStatus is Data:
  Live{} Upgrading{book: H256} Frozen{why: String}

type Status is Data:
  Applied{} Rejected{why: String} RetryLater{why: String}
  Reverted{why: String} Suspended{req: H256}
type Receipt is Data:
  Receipt{event: H256, inst: H256, status: Status, mana: U32, bits: U32,
          size_before: U32, size_after: U32,   # Kindelia StatementInfo.Run{size_diff, end_size}
          pos: U32,
          state_hash: H256, reply: Maybe<&2, Val>, emits: List<&2, Emitted>, epoch: U32}
type Emitted is Data:
  Emitted{topic: String, data: Val}

type OrdState is Data:
  OrdState{inst: H256, last: H256, epoch: U32, time_ms: U32, cut_ids: Map<&2, Unit>}

type BookEntry is Data:
  BookEntry{book: Book, refs: U32}

type Kernel is Data:
  Kernel{params: Params, genesis: H256, dag: Dag,
         books: Map<&2, BookEntry>, insts: Map<&2, Instance>, names: Map<&2, H256>,
         orderers: Map<&2, OrdState>, receipts: Map<&2, Receipt>,
         history: List<&2, Kernel0>, job_seq: U32,
         budget: Map<&2, Budget>}          # per-orderer cumulative mana/size budget (§21.1)
type Budget is Data:
  Budget{mana_used: U32, mana_hi: U32, size: U32}   # mana_used as a 64-bit pair (hi, lo)

type Genesis is Data:                    # kernel/genesis.bend; genesis hash = sha256(H5B(Genesis))
```

```
  Genesis{network_id: U32, params: Params, namer: H256,        # namer owns the empty namespace (Kindelia `reg {}`)
          prelude: H256,                                       # SDK prelude book (Kindelia's T0..TG, IO ctors)
          apps_pkg: String, sdk_pkg: String,                   # hub hashes of the hvm5-apps / hvm5-sdk releases
                                                               # the books were compiled from (reproducible build)
          registry: H256, orderers: List<&2, GenOrderer>,      # books embedded in genesis.h5b
          names: List<&2, GenName>}                            # initial namespace grants (Kindelia's INO names)
type GenOrderer is Data:
  GenOrderer{name: String, book: H256, init: Val}              # e.g. "orderer.dev", with validator key(s) in init
type GenName is Data:
  GenName{name: String, owner: H256}
# Kernel0 = the Kernel without `history` (a separate type: no recursion into history)
```

12.2 KERNEL ENTRY POINT (KERNEL/APPLY.BEND), THE ONLY ONE THE NODE CALLS

```
type Input is Data:
  Submit{ev: Event, verified: Bool}          # node/api did the FFI signature check
  View{inst: String, query: Val, mana: U32}  # read-only; returns reply, no state change
  Tick{now_ms: U32}                          # timeouts of pending asks, orphan GC
type Output is Data:
  WalAppend{ev: Event}
  Broadcast{id: H256}
  Reply{r: Resp}
  Effect{req: H256, inst: H256, eff: Val}    # the node's executor decides whether it is designated
  Settled{orderer: H256, epoch: U32}         # hint: snapshot loop, attestations
  Alarm{msg: String}
def Kernel.apply(k: Kernel, i: Input) -> Kernel & List<&2, Output>
```

12.3 SUBMIT PROCESSING

1. Reject if not `verified`, if `body.net != genesis`, if size > `max_event_bytes`, or if the id is already present (idempotent: reply `Known`).
2. If any parent is unknown → `orphans` (bounded by `max_orphans`, oldest dropped). Retry when parents arrive.
3. If `(author, seq)` is already used by a different id → both are marked `Equivocal` (they stay in the DAG. Settlement presents them as a conflict set, §13.5).
4. `depth = 1 + max(depth(parents))`. Insert. Update `heads` (remove parents, add id). Output `WalAppend`, `Broadcast`.
5. If the kind is `Checkpoint` → `settle` (§13.4). If the kind is `Invoke` / `Deploy` / `EffRes` for a `Commutative` instance → process immediately, in causal order (§13.7).

12.4 A JOB (KERNEL/JOB.BEND)

```
run_job(k, inst, def_idx, args: List<Val>, limits) -> JobOut
  scope = RefScope.of(k.books, inst.book)
  Core.run(scope, exts = args, Apply{fn = BRef{0, def_idx}, args}, limits)
```

Continuations resume through `Inst{blob = k_val.blob, args = [answer, state]}`.

12.5 THE OP5 INTERPRETER (KERNEL/EXEC.BEND), WHICH IS ATOMIC PER EVENT

```
exec(k, top_inst, op, ctx):
 ctx = { stack: [inst ids on call stack], overlay: Map<id, Val new state>,
         emits: [], asks: [], sys: [], mana_used, bits_used }
 match op:
   Done{state, reply}  → overlay[top] = state; return reply
   Call{target, msg, k}:
       depth > limits.calls_depth → Fail(CallDepth)
       target on stack → Fail(Reentrant)                      # affine state: no reentrancy
       target.orderer != caller.orderer → Fail(CrossOrderer)
       Schema.check(target msg) else Fail(BadInput)
       callee_state = overlay.get(target) or target.state
       env' = env with self = target, from = caller instance id      # Kindelia IO_CALL: subject := callee,
                                                                     # caller := previous subject
       r = exec(run_job(target, h5_step, [callee_state, msg, env']), ctx+push target)
       (an Ask inside a nested call → Fail(NestedAsk))
       continue with run_job-resume(k, [r.reply])
   Ask{eff, save, k}   → overlay[top] = save; asks += PendingAsk(req = sha256(event‖n), k)
                         status Suspended{req}; Output Effect
   Emit{topic, d, next}→ emits += (topic, d); continue with next
   Read{q, k}          → answer KQuery (§10.1) from the kernel as of the start of this event
                         (plus this event's overlay for NameOf); +10 mana; continue with k(answer)
   Sys{call, k}        → require inst.sys (SysCap); sys += call; continue with k(result)
                         (Spawn's result id = sha256("inst"‖event id‖n), deterministic)
   Fail{why}           → REVERT: drop overlay/emits/asks/sys; receipt Reverted
 Any job halting with OutOfMana/ArenaFull/NotClosed → REVERT
 On success: commit overlay (Val.seal each state), apply sys calls in order,
             register asks, write receipt (mana, bits, state_hash, reply, emits)
```

The mana and bits of every nested job are summed against the **event's** limit (`min(event.mana, manifest.limits.mana, params.mana_max)`).

12.6 INSTANCES: SPAWN, UPGRADE, FORK, VERSIONING (KERNEL/INST.BEND)

- **Spawn**: `book` must be in `books` (the object was uploaded and validated). Run `h5_init(init, env)` . It must return `Done{state, _}` . Create the Instance, `refs(book)++` , `names[name] = id` .
- **Upgrade** (`Sys{Upgrade{inst, book}}` from the registry, owner-authorized there):
  1. If `inst.pending` is non-empty → `status = Upgrading{book}` . New Invokes get `RetryLater{Upgrading}` . When pending drains (answered or timed out) → go to step 2.
  2. If the new manifest exports `h5_migrate` → `state' = run_job(new book, h5_migrate, [state])` . Otherwise state_type schemas must be equal, else `Rejected{SchemaChange}` .
  3. `inst.book = book` , `version++` , `refs(old)--` , `refs(new)++` . The old book is collected when its refs reach 0 *and* no retained VNet references it (VNet refs count as refs). In-flight work within one event always finishes on the book it started with, because the RefScope is fixed per job.
- **Fork**: a new instance with `state = +inst.state` (O(1) share, the "DUP-fork almost free") and any book. It is used for shadow testing: send the same messages to both and compare receipts. **Merge** is an app-level function run as a normal job.

12.7 BOOK LIFECYCLE

`PUT /v1/objects` stores bytes in the CAS. The first `Deploy` / `Spawn` referencing a hash makes the kernel **validate** it. On top of the checks below, Kindelia-derived limits apply: constructor arity ≤ 16 (CTR rule), net depth ≤ 256 (`MAX_TERM_DEPTH`), and every def blob linear, meaning each var exactly twice (`check_linear`). Checks: decode H5B, check that every `BRef` is in range, every blob is closed, `nvars` is correct, the manifest entries exist, and the schemas are well-formed. Only then is the book inserted into `books` . Books with `refs == 0` older than `retain_epochs` are dropped from the kernel (the CAS copy is GC'd by the store).

---

13. THE EVENT DAG: SIDE EFFECTS AND INPUTS AS AN APPEND-ONLY STRUCTURE

13.1 PROPERTIES

- **Grow-only set** of signed, content-addressed events. Union is commutative, associative and idempotent, so any gossip order converges. Insertion validity is **objective**: signature, structure, size, genesis. It never depends on local time or local pruning.

- Every external input is an event: user invocations, deploys, **effect answers** (`EffRes`), orderer checkpoints, node attestations. Nothing else enters the kernel.

13.2 CANONICAL ORDER

`order(S)` = sort the events of `S` by `(depth, id)` ascending. Depth respects causality (a parent has a smaller depth), so this is a deterministic topological order.

13.3 CUTS AND BATCHES

A `Checkpoint{orderer, prev, heads, epoch, time_ms, proof}` defines `cut = ancestors*(heads)` . Its **batch** is `cut \ cut(prev)` , which is the events not yet settled by this orderer (tracked in `OrdState.cut_ids` , pruned at the horizon). The walk uses fuel `max_batch` . If the batch exceeds it, the checkpoint is `Rejected{BatchTooLarge}` , and the orderer app must checkpoint more often.

13.4 SETTLEMENT (`KERNEL/SETTLE.BEND`)

When a Checkpoint event is inserted:

1. The target orderer instance must exist. Run its `h5_step` with the checkpoint as `msg` . The **orderer app** decides validity: signatures and quorum, PoW, `prev` linkage. It replies `Take` / `Drop` . Consensus logic lives *only* there.
2. If accepted: `batch = order(batch(cp))` . Keep the events whose target instance has `orderer == cp.orderer` .
3. Group by conflicts (§13.5) → decisions.
4. Process the taken events in canonical order: `Invoke` → job; `EffRes` → resume the pending ask; `Deploy` → registry invoke. Apps marked `isolated` may run in parallel (§24).
5. Each processed event gets `Settled{epoch}` and a receipt. `Output Settled{orderer, epoch}` .
6. For **reorg-capable** orderers (PoW), the kernel keeps `history` . Following Kindelia's reversible heaps, it holds at most `history_keep` snapshots at **exponentially spaced ages** (for example, at epoch 1000 it keeps 998, 996, 992, 984, 968, …). A reorg to epoch E rewinds to the newest snapshot ≤ E and replays from there. Persistent Val makes each snapshot an O(1) share, so Kindelia's `absorb` / `U128_NONE` machinery is unnecessary. If a checkpoint arrives whose `prev` is not the current tip but is in history and has more work (as the orderer app replies), the kernel rewinds to that snapshot and re-settles. Finality comes only after `finality_depth` checkpoints, and only then are receipts marked `final` in the API.

13.5 CONFLICTS: "WHAT TO TAKE, DISCARD BOTH, RETRY"

Two events in the same batch **conflict** when they target the same instance, are **causally concurrent** (neither is an ancestor of the other, checked by a DFS restricted to the batch), and either (a) share a declared key in `body.keys` , or (b) are an equivocation (same author, same seq). A **conflict set** is a connected component of the conflict relation.

Policy	Decision
`FirstWins`	first in canonical order → `Take` , the rest → `Retry`
`DropAll`	all → `Retry`
`Custom`	run `h5_resolve(state, set_as_Val) -> List<Decision>` (metered, pure; missing ids → `Retry`)

Equivocations are always presented as a conflict. The default for them is `DropAll` . `Retry` produces the receipt `RetryLater{Conflict}` . The client sees it through `GET /v1/receipts/<id>` and resubmits with new parents (which are then causally *after* the winner, so no conflict). **Declared keys work like an access list:** an app that touches state an author did not declare is behaving correctly by its own rules. The kernel does not police it, because the app defines what a key means.

13.6 PRESENTATION TO THE APPLICATION AS AN APPEND-ONLY INET

Normatively, the app sees a sequence of `h5_step` calls in settled order. **Equivalently** (and this is how to reason about it), each instance is the net

```
state_k = step(step(step(init, e1), e2), …)      # a left fold over the event stream
stream  = (e1 (e2 (e3 … ?tail)))                  # CON-list whose tail is a *free port*
```

Appending an event links the free tail port to `(e_new ?tail')`. Linking only adds, and the rules are confluent, so the reduction already performed is never invalidated. The consumed prefix is erased by the reduction itself (affinity), which is the *natural* pruning of the stream. The kernel implements this fold explicitly, one job per event, because that is snapshot-friendly and metered per event. `tests/kernel/stream_equiv.bend` checks, on small apps, that reducing the stream net once gives the same `Val` as the per-event fold.

13.7 COMMUTATIVE MODE (NO CONSENSUS)

If a manifest declares `Commutative{}`, the app's `LAWS.bend` **must** contain

```
law commutes:
  for s: State
  for a: Msg
  for b: Msg
  for e1: H5.Env
  for e2: H5.Env
  {App.pure_step(App.pure_step(s, a, e1), b, e2) == App.pure_step(App.pure_step(s, b, e2), a, e1) : State}
```

(`pure_step` is the app's pure core, with no Ask/Call.) Base has no `Process`, so no `h5` tool can run `bend`. The check is split into two parts:

- **`h5c` (static):** it refuses `Commutative{}` unless the source defines `law commutes` with exactly the statement above and a proof `def commutes(…)` whose body contains no `?` hole.
- **The author's gate:** `bend PROOF.bend` in the app package, run by the author or CI, as for every package. Anyone can re-verify it, because the app is published by hub hash.

This is a toolchain check, not a kernel check (ADR-027: a kernel-verifiable proof certificate is future work). Such instances process events on insertion, in causal order: an event is processed once all its parents have been processed. Nodes converge once they hold the same event set. `Call` and `Ask` are forbidden in commutative apps (they Fail with `NotCommutative`).

13.8 PRUNING AND THE HORIZON (`KERNEL/PRUNE.BEND`)

- After orderer epoch `E` is settled **and** a snapshot covering `E` is stored locally with `≥ q` matching attestations (§16), events with `Settled{epoch ≤ E - retain_epochs}` may be removed from `dag.events`. Their ids are also removed from `cut_ids`. An unknown ancestor id found during a batch walk is treated as "already settled", which is safe because it is below the horizon.
- Pruning is **local**. It never changes the outcome, because settlement needs only the batch's own events plus ids.
- Late events whose parents are old are still valid. The app sees `Env.lag` and may reject stale input.
- Receipts older than `retain_epochs` are pruned in the same way (they remain in snapshots and WAL archives).

---

14. THE REGISTRY (A BEND2 APP, GENESIS-INSTALLED WITH `SYSCAP`)

`hvm5-apps/src/registry/main.bend` (separate package, §5.1) is compiled by `h5c` during `h5 genesis build` and embedded in `genesis.h5b`. It is written against the SDK and uses Base `Map` from the prelude. The runtime requires only the **message and state types below**, because `h5` CLI and `/v1/reg` decode them, and it requires the rules below. The runtime's own tests use `tests/kernel/books/registry_min.h5`, a hand-written registry that implements the same rules for the name-table subset.

```
type Entry is Data:
  Entry{owner: H5.Addr, inst: Maybe<&2, H5.Addr>, book: Maybe<&2, H5.Addr>}
type State is Data:
  State{names: Map<&2, Entry>, namer: H5.Addr}
type Msg is Data:
  Register{name: String, owner: H5.Addr}                          # Kindelia `reg`
  Deploy{name: String, book: H5.Addr, init: Val}                  # spawn under name
  Upgrade{name: String, book: H5.Addr}
  Fork{name: String, new_name: String, book: H5.Addr}
  Transfer{name: String, owner: H5.Addr}
```

Rules (verified against Kindelia `can_deploy / can_register` in `src/hvm.rs`): Kindelia separates two different permissions, and so do we.

Operation	Allowed when
Deploy a program under name `N`	`N` is non-empty, **and** either `N` has no dot (anyone may deploy it, first come first served) or `from == owner(namespace(N))`, where `namespace("A.B.c") = "A.B"`
Register namespace `N` (`Register{N, owner}`, Kindelia `reg`)	`from == owner(namespace(N))`; for an undotted `N` the parent is the **empty namespace**, whose owner is the **Namer** from genesis. So only the Namer creates top-level namespaces
Register the empty namespace	only in genesis (Kindelia registers `""` to the Namer in `genesis.kdl`)

- Deploying an undotted name does **not** confer ownership of namespace `N`. Anyone can still own `N.x` later if the Namer registers `N` to them. This matches Kindelia exactly.
- **One shared name table.** A name used by an instance or a namespace cannot be reused (Kindelia's `exists(name)` check, used by `fun` / `ctr` / `reg` alike) → `Rejected{Taken}`.
- Name syntax: Kindelia's 64-letter alphabet `. 0-9 A-Z a-z _`, 1–64 chars, with no empty segment. (Kindelia caps names at 12 letters because they are 72-bit; we store Strings, so the cap is 64.)
- A non-owner → `Rejected{Unauthorized}`.
- The registry state records, per namespace, the list of names deployed under it (Kindelia's `RegInfo{ownr, stmt}`), which is served at `GET /v1/reg/<name>`.
- `Deploy` on a name that already has an instance → `Rejected{Exists}` (use `Upgrade`).
- On success the registry issues `Sys{Spawn…}` / `Sys{Upgrade…}` / `Sys{Fork…}`, followed by `Sys{Bind{name, inst}}`.

Laws (in `hvm5-apps/spec/registry/`, proven in `hvm5-apps/proofs/registry/`) are concrete scenarios proven by `{==}`: `register_then_taken`, `dotted_needs_parent_owner`, `upgrade_needs_owner`, `deploy_existing_rejected`. The `hvm5` repo re-checks the same scenarios end to end in `tests/e2e/apps.bend` against the pinned release.

15. CONSENSUS AS APPLICATIONS (PACKAGE `HVM5-APPS`: `SRC/ORDERER-*`)

The runtime knows only the `Checkpoint` event kind, the manifest's `orderer` field, and the role loader (`node/roles.bend` loads `role.bend` from the genesis-pinned `hvm5-apps` hash). The table below states the *required behavior* of the shipped orderers. Their code lives in `hvm5-apps`, and the runtime tests use `tests/kernel/books/orderer_test.h5`, which accepts any checkpoint from a fixed test address.

Each orderer is two things: a **validator app** (HVM5, deterministic, which decides whether a `Checkpoint` is valid) and a **role** (`role.bend`, a Bend2 IO program run by some nodes, which *produces* checkpoints).

Orderer	Validator rule	Role	Finality
`orderer-dev`	signer == configured key; `prev` == last; epoch == last+1	every `interval_ms` emit a Checkpoint over the current heads (time = `IO.now`)	immediate
`orderer-quorum`	`proof` = list of ≥ ⌈2n/3⌉ distinct validator signatures over `(prev, heads, epoch, time_ms)`; validator set in state; set changes by quorum-signed `Reconfigure` msg	round-robin leader proposes; others co-sign via `POST /v1/events` of `Attest`-like co-sign events	immediate (BFT-style, no reorg)
`orderer-pow`	`sha256(H5B(cp without proof) ‖ nonce) < target(epoch)`; difficulty retarget every 64 epochs; heaviest chain	miner loop	probabilistic; `finality_depth`

Kindelia's blocks, PoW, 1s timestamps and fork choice live **only** in `orderer-pow`. Signature checks *inside* validator apps need `secp256k1` verify, but apps cannot do FFI. The kernel therefore **pre-recovers** the signers listed in the proof, as follows: `proof` is `Sigs{list of sig65}`, and the kernel attaches `signers: List<Maybe<Addr>>`, recovered by the node (IO layer, §19) when it inserts the event. The validator app sees `(sig, recovered_addr)` pairs and compares them against its validator set. This is the single place where FFI results enter deterministic code. They are part of the event's local validation, and every honest node computes the same flags.

---

16. CLUSTER: SYNC OF EVENTS AND REDUCTIONS

1. **Event sync (gossip).** Every `gossip_ms`, for each peer: `GET /v1/heads`. For unknown ids, `GET /v1/events/<id>` recursively (bounded by `max_fetch`), verify, then `Submit`. New local events are also pushed with `POST /v1/events` to all peers (best effort).
2. **Attestations.** After settling epoch E, each validator node (a node holding a key in the cluster config) emits `Attest{checkpoint, kernel_root, receipts_root}`. `kernel_root = merkle(sorted (inst_id, sealed state hash, book hash, version))`, and `receipts_root = merkle(receipt hashes of batch E in canonical order)`.
3. **Divergence.** If ≥ q attestations agree on a root that differs from mine → `Alarm`. The instance set goes to `Frozen{Diverged}` locally, the node stops serving `view`, and it resyncs (item 4). This is a bug detector for non-determinism. By Claims A and B it should never fire.
4. **Reduction sharing ("sync their reductions").** A node may skip computing epochs by fetching a peer snapshot at epoch E whose `kernel_root` has ≥ q matching attestations (`--trust-attested q`). It loads it from `GET /v1/objects/<manifest>`, verifies every object hash, and resumes settlement at E+1. New nodes bootstrap the same way. `--verify` mode recomputes everything and only compares roots.
5. Membership in v1 is static (`--peers host:port,…`). A dynamic membership app is future work (ADR).

---

17. PERSISTENCE: SNAPSHOT + WAL + CAS (`NODE/SNAPSHOT.BEND`, `NODE/WAL.BEND`, `NODE/STORE\*.BEND`)

17.1 STORE INTERFACE

```
type Store is Data:
  Disk{root: String}
  S3{host: String, port: U32, bucket: String, prefix: String, region: String}
                                          # credentials from IO.get_env("H5_S3_KEY"/"H5_S3_SECRET")
  Tee{a: Store, b: Store}                 # write both, read a then b
# IO API (node/store.bend). There are only three kinds of write:
def Store.put_obj(s: Store, hash: H256, bytes: Bytes) -> IO(Result<&1, &1, U32 & String, Unit>)   # immutable, idempotent
def Store.append(s: Store, log: String, frame: Bytes) -> IO(Result<&1, &1, U32 & String, Unit>)   # wal-*, head.log (disk)
def Store.cas_head(s: Store, expect: Maybe<&1, String>, rec: HeadRec)                             # S3 HEAD (If-Match)
  -> IO(Result<&1, &1, U32 & String, String>)                                                     # → new etag
def Store.get(s: Store, key: String) -> IO(Result<&1, &1, U32 & String, Bytes>)                  # objects: hash-verified
def Store.has(s: Store, hash: H256) -> IO(Bool)
```

Base gives only `File.open/read_bytes/read_at/size/write_bytes/close`. There is no rename, mkdir, delete or fsync, and no `Process` (§3.1). The store is designed so that **every write is either immutable or an append**. Atomic replace is never needed:

- **Disk**:
  - **Flat layout, no directories.** The operator creates `--data DIR` once. All files live directly in it (§17.2).
  - **Immutable objects**: `obj-<64 hex>`. A write goes to a fresh name, and readers verify `sha256(bytes) == name` on every read. So a torn write is detected, then rewritten or refetched from a peer. Writing an existing object is a no-op.
  - **Mutable pointer = append-only log** (`head.log`). Each record is `u32be(len) || H5B(HeadRec{epoch, manifest_hash}) || sha256(prev 2 fields)[0..4]`. The current HEAD is the **last record whose checksum is valid**, and a torn tail is ignored. This is the SQLite-WAL / LSM idea, and it replaces `mv`.
  - **Durability**: without the optional `fsx` twin, the store survives *process* crashes (the data is in the OS page cache), but not power loss. With `fsx.fsync`, it fsyncs objects, then the WAL, then `head.log`, in that order.
  - **GC**: Base cannot delete files. With `fsx.unlink`, dead `obj-*` and `wal-*` files are removed after the snapshot that superseded them is durable. Without it, `h5 node gc` prints the dead file list for the operator (ADR-026).
- **S3** (any S3-compatible endpoint; this is not the `aws` CLI):
  - **Transport**: HTTP/1.1 over Base `TCP.connect/send/recv`, signed with **AWS SigV4**. HMAC-SHA256 is pure Bend2 over bend-collections SHA-256 (`lib/crypto/hmac.bend`, RFC 4231 vectors as `{==}` laws).
  - **TLS**: Base TCP has no TLS. So `S3{endpoint}` is plain HTTP to MinIO/Ceph on a private network, or to a local TLS-terminating proxy (for example stunnel → `https://s3.<region>.amazonaws.com`). The proxy is operator infrastructure, not our code (ADR-013).
  - **Immutable objects**: `PUT <prefix>/obj/<hex>` with `If-None-Match: *`. `200` = written, and `412 Precondition Failed` = already exists, which also counts as **success**, because the object is content-addressed.

- **HEAD manifest = compare-and-swap**: `PUT <prefix>/HEAD` with `If-Match: <etag from last GET>` (the first write uses `If-None-Match: *`). On `412` / `409`: re-GET HEAD, check that the remote manifest is an ancestor (same genesis, lower or equal epoch), then retry. This comes from AWS "conditional writes" (`docs.aws.amazon.com/AmazonS3/latest/userguide/conditional-writes.html`). **Never** treat an ETag as a content hash.
- **Bodies are hex text** in v1, because `TCP.send` takes a `String` (ADR-025: binary bodies once byte-transparent `String` sends are confirmed). `x-amz-content-sha256` is the sha256 of the hex body.

- **Tee{a, b}**: writes go to both stores, and the HEAD record is appended to `a` only after `b` confirms. Reads try `a`, then `b`.

17.2 LAYOUT

```
<data>/                          # created by the operator; the node never creates directories
  node.key                       # 64 hex chars + "\n" (Kindelia format); chmod 600 via fsx, else a startup warning
  obj-<64 hex>                   # CAS: books, Val objects, blob objects, snapshot manifests and chunks
  wal-<8 hex seq>.log            # framed H5B events: u32be(len) ‖ bytes ‖ sha256(bytes)[0..4]; append-only;
                                 # a new segment starts after each snapshot
  head.log                       # append-only HeadRec{epoch, manifest_hash, first_live_wal_seq} records

S3: <prefix>/obj/<64 hex>   <prefix>/wal/<8 hex seq>   <prefix>/HEAD (H5B HeadRec, CAS via If-Match)
```

17.3 SNAPSHOT ALGORITHM (A PURE BEND2 LIB PLUS A THIN IO DRIVER)

1. The actor sends `+kernel` (refcounted share, O(1)) to the snapshot task through a Chan. The kernel keeps running.
2. `Snap.plan(kernel) -> SnapPlan` (pure) seals states, then splits every Val with `size ≥ 256` into CAS objects `VObj{hash, shallow_bytes}`. Children at or above the threshold are referenced by hash, so the object graph is a Merkle DAG. Everything else (dag, maps, receipts, books by hash) goes into `KernelBytes`, split into 1 MiB chunks.
3. Write only the objects that `Store.has` reports as missing. Unchanged subtrees keep their hash, so snapshots are **incremental** by construction. Hashing and encoding of independent subtrees run in parallel.
4. Write the manifest object `SnapManifest{epoch, genesis, kernel_root, chunks, objects_root}` (`obj-<hash>`), start a new WAL segment, then **append** `HeadRec{epoch, manifest_hash, first_live_wal_seq}` to `head.log` (on S3: CAS `HEAD`). The snapshot becomes visible only through that record.
5. Older WAL segments are dead once the HeadRec is durable (keep one previous segment). They are deleted via `fsx.unlink` or listed by `h5 node gc`.

17.4 RECOVERY (`NODE/RECOVER.BEND`)

1. Scan `head.log` and take the last record with a valid checksum.
2. Load its manifest and verify every hash.
3. Rebuild `Kernel` (`Snap.load` is pure, given bytes).
4. Replay the WAL segments `≥ first_live_wal_seq`, in order, through `Kernel.apply(Submit{ev, verified=re-checked})`, stopping at the first frame with a bad checksum (a torn tail).
5. Check that the resulting `kernel_root` matches the last Attest this node signed, if there is one.

If the manifest fails verification, fall back to the previous HeadRec. A peer snapshot (§16.4) is the last resort. Law (`spec/node/`, scenario): *crash at any byte offset of any write ⇒ recovery yields a prefix-consistent kernel*, tested by truncating files at every offset in `tests/node/torn_writes.bend`.

---

18. L5 — NODE IO SHELL AND THE MINIMAL API (`NODE/`)

18.1 ACTOR (`NODE/ACTOR.BEND`)

```
type Cmd is Data:
  CSubmit{ev: Event, verified: Bool, reply: Chan(Resp)}
  CView{inst: String, query: Val, reply: Chan(Resp)}
  CQuery{q: Query, reply: Chan(Resp)}          # heads, event, receipt, instance, object-refs
  CTick{now_ms: U32}
  CSnapshot{reply: Chan(Kernel)}              # hands out +kernel
```

The loop is `@unsafe def Actor.loop(k: Kernel, mb: Chan(Cmd)) -> IO(Unit)`. It is one of the three named `@unsafe` IO loops, alongside `Http.accept_loop` and `Gossip.loop`. On each `recv`: `Kernel.apply` → for each Output, perform it in order. **`WalAppend` is done before the `Reply` is sent.** `Broadcast` is handed to gossip. `Effect` goes to the executor.

18.2 HTTP (`NODE/HTTP.BEND`)

HTTP/1.1 is implemented over `TCP.listen/accept/recv/send`: request line, headers, `Content-Length` body (max 8 MiB), `Connection: close`. One `IO.spawn` per connection. Binary bodies are **lowercase hex** with content type `application/x-h5b-hex`. Human-readable text is returned for `GET /v1/health` and for `?fmt=text` (pretty Val using the book's ctor names).

18.3 ROUTES (THIS IS THE WHOLE API)

Method	Path	Body → Response
POST	/v1/events	hex(Event) → `200 hex(id)` | `202 orphan` | `400 reason`
GET	/v1/events/<id>	→ hex(Event)
GET	/v1/heads	→ hex(List<H256>)
PUT	/v1/objects	hex(bytes) (≤ max_book_bytes) → hex(sha256) — unreferenced objects GC'd after `object_ttl_ms`
GET	/v1/objects/<hash>	→ hex(bytes) (books, snapshot manifests/chunks, Val objects)
GET	/v1/receipts/<event-id>	→ hex(Receipt) | `404 pending`
GET	/v1/instances/<name-or-0xid>	→ hex(InstanceInfo{id, name, book, owner, version, state_hash, status})
POST	/v1/view/<name-or-0xid>	hex(Val query) → hex(Val) (runs `h5_view`, node-local mana cap, no event)
GET	/v1/health	→ `ok <genesis> <epoch> <heads count>`
POST	/v1/dry-run	hex(List<Event>) → hex(List<Receipt>): runs the events against the latest settled state and **discards** the result (Kindelia `POST /run` / `kindelia run-remote`, which uses `test_statements + undo`). Node-local mana cap
GET	/v1/instances/<name>/state	→ hex(Val) of the state. `413` if the state is larger than `max_state_reply` nodes (Kindelia returns 418 `TermTooBig` in this case)
GET	/v1/reg/<namespace>	→ hex(RegInfo{owner, names}) (Kindelia `GET /reg/<name>`)
GET	/v1/stats	→ hex(Stats{epoch, mana, space, inst_count, book_count, reg_count, peers, heads, orphans}) (Kindelia `GET /stats` + Heartbeat)
GET	/v1/peers[?all=1]	→ hex(List<Peer{addr, seen_at_ms}>) (Kindelia `GET /peers`, `/peers/all`)
GET	/v1/log?since=<n>	→ hex(List<NodeEvent>) long-poll of node events. It replaces Kindelia's WebSocket event stream (Base has no WS). Kinds: `Included`, `Orphan{missing}`, `Settled{epoch}`, `Reorg{old, common, new}`, `Heartbeat{…}`, `Peer{seen

The core is the first nine routes. The last six are read-only conveniences taken from Kindelia's API, and none of them changes state.

Deploying is two calls: `PUT /v1/objects` (the book) and `POST /v1/events` (`Deploy`). **Invoking is one call:** `POST /v1/events` (`Invoke`). Reads use `view`, `receipts` and `instances`. Peers use the same routes for gossip and snapshot transfer.

18.4 EFFECT EXECUTOR (`node/effects.bend`)

It receives `Effect{req, inst, eff}` . If this node is designated (per `ExecPolicy` : `AnyNode` → every validator node executes, `Designated` → only that node, `Author` → nobody; the client answers), it performs the effect with Base IO: `Now` → `IO.now` , `Random` → read `/dev/urandom` via `File` , `HttpGet/Post` → plain HTTP via `TCP.connect` , and `External` → a registered handler. The answer is signed as `EffRes{req, answer}` and submitted like any event. Competing answers from several nodes are resolved as a conflict (key `= "eff:" ++ hex(req)` , auto-declared) under the app's policy. **The IO world touches the deterministic world only through signed events.**

Timeouts: `Tick` checks `pending.deadline` (in epochs of the instance's orderer). An expired pending ask resumes `k` with the answer `Fail` -encoded Val `Timeout{}` (an SDK ctor), so the app decides what happens.

18.5 CONFIG (`node/config.bend` , FROM `IO.ARGS`)

`--data DIR --port N --genesis FILE --key FILE --peers h:p,… --store disk|s3:bucket/prefix[,profile] --snapshot-every N --retain-epochs N --trust-attested Q --role orderer-dev|orderer-quorum|none --gossip-ms N --threads N --network-id 0xCAFE0005` .

Resolution order (copied from Kindelia `cli.rs ConfigSettings::resolve`): **CLI flag > environment variable (`H5_DATA` , `H5_PORT` , `H5_PEERS` , …) > config file (`~/.h5/h5.conf`) > default.** The config file is a flat `key = value` file (a subset of Kindelia's `default.toml` , parsed in Bend2 with `String.lines / String.split`):

```
data.dir      = ~/.h5/state
network.id    = 0xCAFE0005          # gossip magic; messages with another id are dropped (Kindelia network_id)
network.peers = 10.0.0.2:7070,10.0.0.3:7070
role          = none
api.port      = 7070
```

---

19. IDENTITY AND CRYPTO (`lib/crypto/`)

- **Keys**: secp256k1, exactly as in Kindelia (`crypto::Account::from_private_key`). `h5 keygen` reads 32 bytes from `/dev/urandom` (Base `File`), rejects 0 and values ≥ n, and writes `{priv hex, addr hex}` . The private-key file format is Kindelia's: one line of 64 hex chars (`kindelia sign -s secret_file`). As a result, **Kindelia / Ethereum keys work unchanged.**
- **Address** = **Ethereum address**: `keccak256(uncompressed pubkey without 0x04)[12..32]` (20 bytes), zero-padded to `Addr` /H256. Kindelia uses the same derivation: its subject `#x7e5f4552091a69125d5dfcb7b8c265` for secret key `0x1` is the first 15 bytes of the Ethereum address `0x7e5f4552…9029395bdf` . We keep all 20 bytes, because Kindelia truncates only to fit a 120-bit number and we have no such limit. The result is real Web3 identity: MetaMask-style keys and addresses.
- **Signing** (Kindelia `sign_code` : sign the hash of the statement serialized *without* its signature), made **wallet-compatible via ERC-191 `personal_sign`** (`eips.ethereum.org/EIPS/eip-191` , version byte `0x45`):

  ```
  msg32  = sha256("h5sig1" ‖ genesis_hash ‖ event id)                 # domain: network + event
  digest = keccak256(0x19 ‖ "Ethereum Signed Message:\n32" ‖ msg32)   # exactly what personal_sign(msg32) signs
  sig    = r(32) ‖ s(32) ‖ v(1),  v ∈ {27, 28},  low-S (s ≤ n/2) required; any other v or high-S → invalid
  ```

  So MetaMask or any Ethereum wallet can sign H5 events: `personal_sign(msg32)` . Signatures cannot be replayed across networks, because `genesis_hash` is inside `msg32` . We deliberately use **no** EIP-155 transaction signing, because events are not Ethereum transactions. The signer is **recovered** from the signature (Kindelia `signer_name(hash)`), so events carry no pubkey. **Golden vectors** (as `{==}` laws in `spec/lib/crypto/`): keccak256(`""`) = `c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470` ; keccak256(`"abc"`) = `4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45` ; sha256(`"abc"`) = `ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad` ; priv `0x…01` → addr `0x7e5f…5bdf` . A sign/recover roundtrip vector goes in `tests/node/secp_vectors.bend` (the FFI can't run inside `{==}`).
- **FFI** (`lib/crypto/effs/secp256k1.{c,js}` , the only required twin): `Secp.sign(priv, h32) -> sig65` , `Secp.pubkey(priv) -> pub64` , `Secp.recover(sig65, h32) -> Maybe pub64` (uncompressed, without 0x04). This is the whole elliptic-curve surface. The **address is derived in pure Bend2**: `Addr.of_pub(pub64) = keccak256(pub64)[12..32]` , using `bend-collections` Keccak (via `lib/deps.bend`). Law: secret key `0x1` → address `0x7e5f4552091a69125d5dfcb7b8c2659029395bdf` via `{==}` , over the pubkey constant of `G` . The FFI calls are `IO` defs, used only in `node/` and `cli/` . The kernel receives `verified: Bool` and requires `recovered == body.author` .

- **C twin** wraps **libsecp256k1 v0.8.0** (bitcoin-core, MIT, released 2026-08-03). Pin the release asset `github.com/bitcoin-core/secp256k1/releases/download/v0.8.0/libsecp256k1-0.8.0.tar.gz`, **sha256** **`dd685546f9e717b9adde329acd5a4cd8083d40f710fdb0a603ee5a83f908132b`** (verified). Do not use the auto-generated `archive/refs/tags` tarball, which has a different hash (`eb52b0e9…`). APIs from `include/secp256k1_recovery.h`: `secp256k1_ecdsa_sign_recoverable`, `secp256k1_ecdsa_recoverable_signature_serialize_compact`, `…_parse_compact`, `secp256k1_ecdsa_recover`. Compact form is `recid ‖ r ‖ s` with recid 0..3: map `v = 27 + recid` and **reject recid 2/3**. RFC 6979 nonces come from the library.
- **JS twin** uses `@noble/secp256k1@3.2.0`, pinned by npm integrity `sha512-Z3ZAWOTxJ0EuTuZTi7Y69iK7GgrLHh8sgm45lVDhg/b3Nk1TpAiKhick2KkZisHuupeepSkyIydN/J459SdX1w==` (verified with `npm view`) and recorded in the header of `effs/secp256k1.js` (ADR-028). It must pass the same `secp_vectors`.
- All hash vectors above were cross-checked with `openssl dgst -sha256 / -keccak-256`.

- Instance ids = `sha256("inst" ‖ event id ‖ n)`.
- **Hashes are pure Bend2 and not ours**: `lib/crypto/hash.bend` wraps `bend-collections`' `SHA256Packed.sha256(words, byte_length)` and `Keccak.keccak256(words, length)`, which are proven upstream against executable FIPS 180-4 / sponge specs. The wrapper converts H5B byte lists to packed `Array<U32>` words and turns `Maybe` failure into a kernel-level `Rejected{TooLarge}`. Local laws are regression only: the three FIPS SHA-256 vectors and two Keccak vectors (`"" → c5d2…a470`), via `{==}`. `merkle.bend`: a binary Merkle tree over a list of H256. Odd leaves are promoted, and it is parallel.

20. CLI (`CLI/H5.BEND`)

```
h5 keygen -o me.key
h5 compile ../my-app/main.bend -o counter.h5b [--text counter.h5]   # any app package; it imports hvm5-sdk by hash
h5 genesis build genesis/genesis.conf -o genesis.h5b                # fetch pinned hvm5-apps, check, compile, embed
h5 deploy counter.h5b --name counter --init 'Addr{…}' --key me.key --node http://127.0.0.1:7070
h5 invoke counter 'Inc{5}' --keys c --key me.key [--wait]
h5 view counter 'Get{}'
h5 receipt <event-id>          h5 status          h5 text counter.h5b
# taken from Kindelia's CLI (cli.rs):
h5 test scenario.h5s [--sudo]      # offline: run a scenario file of events against a fresh genesis and print
                                   # one line per receipt, Kindelia-style:  [run] 5 [150 mana | +2 size]
                                   # --sudo ignores mana/size limits (Kindelia `test --sudo`)
h5 serialize ev.txt | h5 deserialize -      # text ⇄ hex H5B (Kindelia serialize / deserialize)
h5 sign ev.hex -s me.key [-E]      # sign a serialized event body (Kindelia sign -s secret_file -E)
h5 post <hex>                      # POST /v1/events a pre-signed event (Kindelia post)
h5 dry-run ev.txt                  # POST /v1/dry-run (Kindelia run-remote)
h5 get inst counter code|state|book     h5 get reg Foo.Bar owner|list
h5 get stats [epoch|mana|space|inst-count|reg-count]      h5 get peers [--all]
h5 node start [--role …] | h5 node clean      # clean asks for confirmation, as Kindelia does
h5 init                            # write the default ~/.h5/h5.conf
```

The `.h5s` scenario format is one event per line in text form (`invoke counter Inc{5} as alice`, `checkpoint`, `deploy counter.h5b as alice init Addr{…}`), with named keys from a `keys:` header. It is the counterpart of Kindelia's `.kdl` test blocks, and it is what the kernel golden tests use.

Val literal syntax is `Ctor{field, …}` with numbers, `"strings"`, `[lists]`, and `0x…` for Addr. It is parsed using the target book's `ctors` table, which is fetched with `GET /v1/instances` and then `GET /v1/objects/<book>`. `Commutative` apps are checked statically by `h5c` and by the author's `bend PROOF.bend` gate (§13.7). No `h5` command shells out, because Base has no `Process`.

20.3 ALLOWED `@UNSAFE` LIST (COMPLETE)

`Actor.loop`, `Http.accept_loop`, `Http.conn` (read until the body is complete), `Gossip.loop`, `Snapshot.loop`, `Effects.loop`, `Role.loop`. All of them are in `node/`. Pure layers have **none**.

---

21. LIMITS AND DEFAULT PARAMS (GENESIS `PARAMS`)

Param	Default	Notes
`mana_max` per event	4,000,000	Kindelia's per-block base. App `limits.mana` ≤ this
`bits_max` per event	4,194,304 (64 Ki new Val nodes)	charged at readback

Param	Default	Notes
`arena_nodes_max`	4,194,304 nodes (64 MiB)	canonical-scheduler cap (§6.9)
`max_event_bytes`	65,536	
`max_book_bytes`	4 MiB	
`max_batch`	4,096 events	
`calls_depth`	16	
`retain_epochs`	10,000	
`finality_depth`	64 (PoW only)	
`history_keep`	64	
`max_orphans`	10,000	
view mana (node-local)	1,000,000	not consensus
`max_state_reply`	65,536 nodes	Kindelia readback cap `1 << 16`
`mana_per_epoch`	4,000,000	cumulative budget, §21.1
`bits_per_epoch`	2,048	cumulative state growth, §21.1

21.1 CUMULATIVE EPOCH BUDGETS (KINDELIA'S BLOCK LIMITS, PER ORDERER)

Kindelia bounds total work and total state as a function of the chain's age: `mana_limit = 4,000,000 × (block+1)` and `bits_limit = 2048 × (block+1)`. Both are **cumulative**, so unused capacity carries over, and they are checked against the running totals (`get_mana_limit / get_size_limit`). This is what keeps *catch-up* affordable. At 55 M mana/s, a node needs about 1 s to replay about 7 s of history, and that property matters directly for us, because new nodes and lagging nodes "sync reductions".

We adopt this per orderer:

```
allowed_mana(o, E) = mana_per_epoch × (E + 1)       # E = epoch of orderer o
allowed_size(o, E) = bits_per_epoch × (E + 1)       # total Val size of all instances ordered by o, in bits
```

During settlement, before each event: if `budget.mana_used + event.mana_limit > allowed_mana` → `RetryLater{EpochFull}`. After each commit: if `total_size > allowed_size` → **revert** that event (Kindelia: "if the size limit was exceeded, revert"). Both checks are deterministic. Per-event limits (`mana_max` , `bits_max`) still apply. Commutative apps, which have no epochs, use per-event limits only, plus a per-author rate limit in `node/api.bend`.

There is no coin and there are no fees (the Kindelia model). Spam is bounded by per-author rate limits in `node/api.bend` (local policy) and by orderer apps, which may require PoW or allow-lists per author.

22. CANONICAL ENCODING H5B (`LIB/CODEC/`)

It is byte-level (simpler than Kindelia's bit-level encoding, (D) ADR-004) and **canonical**: exactly one encoding per value, and decoders reject non-canonical input. *Reviewed against Kindelia's WHITEBOOK and `bits.rs` :* the bit-level format (fixlen/varlen/list with 1/0 continuation bits, 6-bit names, and De Bruijn compression of variable names) saves about 2–4× on wire size. It needs bit-vector code over char lists, though, and its main win, compact names in on-chain *source*, does not apply here: we ship compiled nets whose vars are already indices. The decision stands. ADR-004 records the option of a later `H5B/2` that uses Kindelia's `serialize_list` trick (a 1-bit continuation instead of a length prefix) for short lists.

- `u32` : unsigned LEB128, minimal length. `bytes` : `u32 len || raw` . `string` : UTF-8 code points as `u32` list.
- `list<T>` : `u32 len || T*` . `maybe<T>` : `0x00 | 0x01 || T` . `H256` : 32 raw bytes, big-endian words.
- ADT: `u32 ctor index || fields in declaration order` . Every Bend2 type above is encoded this way, so **all encoders follow one generic rule**. Write one encoder/decoder per type by hand, following that rule. Laws: `decode(encode(x)) == Some(x)` on vectors, plus `encode(decode(b)) == b` for accepted `b` .

- Magic prefixes: book `"H5BK\x01"`, blob `"H5NB\x01"`, event body domain `"h5ev1"`, snapshot `"H5SN\x01"`.

23. SECURITY MODEL

Threat	Defense
Forged events	recoverable secp256k1 signatures over a genesis-bound domain; author = recovered Ethereum address
Type confusion between apps	schema check on every `Call` message (Kindelia allows only flat ctors of numbers in `CALL` for this reason)
Cross-cluster replay	`body.net = genesis`
Equivocation	detected as a conflict set; default `DropAll + Retry`
Hostile nets (loops, bombs)	mana limit; arena cap; total rule table; closed-blob validation; VNet never accepted from outside
Malformed messages	schema check before any job
Reentrancy / double-spend inside a tx	affine state + call stack check → `Reentrant`
Privilege escalation	`Sys` only for `sys` instances (genesis: the registry)
Nondeterminism bugs	attestations + divergence alarm; `sched_equiv` tests
Storage exhaustion	size limits; bits metering; object TTL; pruning
Snapshot tampering	every object verified by hash on load; `kernel_root` attested
Key theft	key file is local; `chmod 600` via the optional `fsx` twin at keygen, else `h5 keygen` prints a warning with the `chmod` command for the operator

24. PARALLELISM (REQUIRED BY AGENTS.MD: "PARALLELIZE WHENEVER POSSIBLE")

Where	How	Why it is safe
Batch execution of `isolated` apps	group the batch by instance; balanced parallel let over groups; each group runs sequentially inside	disjoint states, separate arenas (`Array` single-owner per job)
`Val.seal`	`a b = seal(l) seal(r)` over the field list split in halves	pure
Merkle roots	balanced tree	pure
Snapshot planning / encoding	parallel over top-level instances and objects	`+kernel` is immutable
Conflict detection	parallel over instance groups	pure
Canon hashing of many receipts	parallel map over halves	pure
Intra-net reduction (phase 10, ADR-020)	`Waves` scheduler (§6.4.1): sorted wave, prefix-sum ID ranges, parallel rewrite into disjoint ranges via `Array.fork/join`, sequential link resolution (Base has no atomics)	IDs independent of thread count; mana equal by Claim B; `sched_equiv` proves canon equality

Do **not** parallelize the actor loop, WAL writes, or anything with order-dependent IO.

---

25. WORKED EXAMPLE (END TO END)

`hvm5-apps/examples/counter/main.bend` (an app package; outside `hvm5`, §5.1):

```
import Base
import 0x<hvm5-sdk-hash>/src/h5.bend as H

type State is Data:
  State{count: U32, owner: H.Addr}
type Msg is Data:
  Inc{by: U32}
  Reset{}
type Query is Data:
  Get{}

def Addr.eq(a: H.Addr, b: H.Addr) -> Bool:
  match a b:
    case H.Addr{a1,b1,c1,d1,e1,f1,g1,h1} H.Addr{a2,b2,c2,d2,e2,f2,g2,h2}:
      U32.is_eq(a1, a2) && U32.is_eq(b1, b2) && U32.is_eq(c1, c2) && U32.is_eq(d1, d2)
        && U32.is_eq(e1, e2) && U32.is_eq(f1, f2) && U32.is_eq(g1, g2) && U32.is_eq(h1, h2)

def reset.if(c: U32, o: H.Addr, ok: Bool) -> H.IO5(State & U32):
  match ok:
    case True{}:
      H.IO5.pure(State & U32, (State{0, o}, 0))
    case False{}:
      H.fail(State & U32, String, "not owner")

def step(s: State, m: Msg, e: H.Env) -> H.IO5(State & U32):
  match m:
    case Inc{by}:
      State{c, o} = s
      +n = (c + by : U32)
      H.IO5.pure(State & U32, (State{n, o}, n))
    case Reset{}:
      State{c, +o} = s
      H.Env{self, from, ev, ep, t, lag, sq, pos} = e
      reset.if(c, o, Addr.eq(from, o))

def h5_init(owner: Val, e: Val) -> H.Op5:
  H.Done{H.coerce(State, Val, State{0, H.coerce(Val, H.Addr, owner)}), H.coerce(U32, Val, 0)}
def h5_step(s: Val, m: Val, e: Val) -> H.Op5:
  H.run(State, U32, step(H.coerce(Val, State, s), H.coerce(Val, Msg, m), H.coerce(Val, H.Env, e)))
def h5_view(s: Val, q: Val) -> Val:
  State{c, o} = H.coerce(Val, State, s)
  H.coerce(U32, Val, c)
def h5_manifest() -> H.Manifest:
  H.Manifest{"counter", 1, "State", "Msg", "U32", "H.Addr", "Query",
             H.Ordered{"orderer.dev"}, H.FirstWins{}, H.AnyNode{},
             H.Limits{100000, 65536, 4, 100}, True{}}
```

(The syntax is a sketch. The implementer must make it pass `bend --check-only`, for example by splitting helpers or adding `{x : T}` annotations. Semantics are fixed.)

Flow:

1. In `hvm5-apps`: `bend PROOF.bend` → `All terms check.` Then `h5 compile examples/counter/main.bend -o counter.h5b` prints `0xB00K…`.
2. `h5 deploy counter.h5b --name counter --init 'Addr{…me…}'` does `PUT /v1/objects` and then `POST /v1/events` `Deploy{0xB00K…, "counter", Addr{…}}`.
3. The orderer-dev role emits `Checkpoint{epoch 7}`. Settlement: the Deploy becomes `registry.h5_step(Deploy…)`, which issues `Sys{Spawn}` and `Sys{Bind "counter"}`. The kernel validates the book, runs `h5_init`, and creates the Instance. Receipt `Applied`.
4. `h5 invoke counter 'Inc{5}' --keys c`. At the next checkpoint, a job runs: arena ← `EXT(state) EXT(Inc 5) EXT(env)`, `REF(h5_step)` is applied, and the net reduces. EXPD materializes only `State`'s top layer and `count`. Readback gives `Done{VCtr State [VNum 5, <same owner Val ptr>], VNum 5}`. Receipt `Applied, reply 5, mana ≈ 150`.
5. Two concurrent `Inc` events with key `c` in the same batch: `FirstWins` → one `Applied`, one `RetryLater{Conflict}`. The client resubmits. The retry's parents now include the winner, so there is no conflict.

---

26. LAWS AND TESTS (WHAT MUST EXIST, PER MODULE)

Module	Laws (✓ = general proof expected, ◎ = concrete vector via `{==}`)
lib/std	vec get/set ✓, hex roundtrip ◎, fuel tree visits ≥ n leaves ✓

Module	Laws (✓ = general proof expected, ◎ = concrete vector via `{==}`)
lib/crypto	wrapper regression: sha256 FIPS vectors ◎, keccak256 vectors ◎, `Addr.of_pub(G) = 0x7e5f…5bdf` ◎, merkle of 1 leaf = leaf ✓ (hash correctness itself is proven upstream in bend-collections)
lib/codec	roundtrip per type ◎ (≥ 5 vectors each), canonical rejection ◎
core	each rule on a 2-node net ◎; `sched_equiv` (LIFO vs FIFO canon + mana) on ≥ 20 golden nets ◎; EXT lazy ≡ eager injection ◎; readback∘inject = id on Vals ◎; text roundtrip ◎; alloc/free keeps live count ✓
compiler	golden: source → `.h5` text equal ◎; `run(compiled f)(x) == expected` for ≥ 30 functions (list ops, Map, strings, matches) ◎
kernel	ABI decode of every ctor ◎; order respects causality ◎; conflict sets ◎; revert leaves state hash unchanged ◎; upgrade scenarios ◎; reentrancy rejected ◎; stream ≡ fold ◎; prune does not change the kernel_root ◎; `abi ∉ abi_versions` rejected ◎ (all over `tests/kernel/books/*.h5`)
abi (test-only)	`tests/abi/sdk_conformance.bend`: every SDK ctor compiled by h5c decodes via `kernel/abi.bend` ◎
e2e (pinned apps)	registry rules ◎; counter behaviour ◎ (from `tests/e2e/apps.bend`)
hvm5-sdk (separate)	`IO5` monad laws ◎. *Listed so the ABI owner knows what the SDK must prove*
hvm5-apps (separate)	registry rules ◎; orderer validators ◎; kv-crdt `commutes` ✓ (general)
cluster	`tests/cluster/three_nodes.bend`: 3 in-process nodes (`IO.spawn(Node.run(cfg_i))`, own ports and data dirs, talking over real TCP); "kill" = send `Stop` over the node's control Chan, "restart" = `Node.run` again on the same dir; 1,000 random invokes; all attestations equal; kill and restart a node; snapshot restore; equal roots

Golden net list (minimum): identity, K combinator, church 2+3, `2+3` OPR, SWI 0/succ, DUP of a pair, DUP of a lambda (label 0), fork of state (label 1 over label-0 dups: commute), ERA of a large tree, REF recursion (sum 1..100 over a U32 countdown), EXT expansion of a 3-level Val, VNet capture of a closure, OPR partial path, CON/DUP commute bomb bounded by mana → `OutOfMana` with equal mana under both schedulers.

---

27. PHASED PLAN (EACH GATE IS AN EXACT COMMAND; ALL MUST PASS)

Phase	Deliverable	Gate
0 Skeleton	tree of §5.2, `tools/toolchain.json` (2.0.28), README, AGENTS.md, ADR-000/001/004, `src/lib/deps.bend` with the pinned bend-collections hash, empty spec/proofs that check	`bend PROOF.bend`; `bend main.bend --check-only`
1 lib	std (vec, bytes, hex, fuel), hash wrappers over bend-collections, merkle, H5B codec basics	`bend proofs/lib/proof.bend`; `bend tests/lib/golden.bend` prints PASS
2 core machine	tags, ports, Mach, alloc, link, 11 rules, LIFO reducer, meter, blob instantiate, text parser/printer	`bend proofs/core/proof.bend`; `bend tests/core/golden.bend` (all golden nets incl. sched_equiv)
3 data bridge	Val, EXT/EXPD, readback, capture, canon, Val hash/seal, `Core.run`	same, plus EXT/readback laws
4 compiler	h5c passes 1–8, hub-import resolution, prelude mapping, schemas, manifest evaluation, `Book.abi`	`bend proofs/compiler/proof.bend`; `bend tests/compiler/golden.bend` (fixture SDK/prelude in `tests/compiler/fixtures/`)

Phase	Deliverable	Gate
5 ABI + single-instance kernel	`kernel/abi.bend`; job, Op5 interpreter (Done/Emit/Fail/Call), receipts; offline runner over `tests/kernel/books/counter.h5`	`bend proofs/kernel/proof.bend`; `bend tests/kernel/scenario_counter.bend` PASS
6 DAG + settlement + Sys + test registry/orderer	dag, order, batch, conflicts, settle, Sys calls; `registry_min.h5`, `orderer_test.h5`	kernel scenarios: deploy→invoke→conflict→retry PASS
7 node + CLI + persistence (disk) + genesis	actor, HTTP, routes, secp256k1 FFI, keys, WAL, snapshot, recovery, role loader, effects executor (Now/Random/HttpGet), `h5 genesis build`	`bend src/node/main.bend -o h5d && bend src/cli/h5.bend -o h5`; `bend tests/node/e2e.bend`; once `hvm5-sdk` and `hvm5-apps` releases exist: `bend tests/abi/sdk_conformance.bend` and `bend tests/e2e/apps.bend` PASS
8 Ask/EffRes, upgrades, fork	pending asks, timeouts, Upgrade drain+migrate, Fork, book refcounts/GC	kernel scenarios + e2e upgrade PASS
9 cluster	gossip, attestations, divergence alarm, trust-attested bootstrap, pruning (quorum orderer from pinned hvm5-apps)	`bend tests/cluster/three_nodes.bend` PASS
10 harden	S3 store, incremental snapshot benchmarks, parallel batch, reorg handling (with pinned orderer-pow), intra-net parallel reducer (ADR-020), security review, `bend main.bend --publish`	all previous gates + `bend benchmarks/bench.bend` recorded in README

The sibling packages have their own phased plans. Their minimum to unblock phase 7 is `hvm5-sdk` (the §10 types + `IO5` laws) and `hvm5-apps` (registry + orderer-dev + counter), each published with `bend main.bend --publish`.

Definition of done for any phase: every gate passes, no new `@unsafe` outside §20.3, the README checklist is updated, and ADRs are written for every deviation.

---

28. RISKS, OPEN QUESTIONS, ADR SEEDS

ID	Topic	Default decision / mitigation
ADR-000	Bend2-only	accepted
ADR-001	Foreign effects = secp256k1 only; hashes from bend-collections (pure Bend2)	accepted
ADR-002	Package split `hvm5` / `hvm5-sdk` / `hvm5-apps`; Val-level ABI + `Book.abi`; hub-hash pinning in `src/lib/deps.bend` only	accepted
ADR-004	Byte-level H5B instead of Kindelia's bit-level encoding	accepted
ADR-007	Strict (full) reduction, not lazy	accepted; laziness lives in EXT/REF only
ADR-010	Pure-Bend2 sha256 is slow	seal lazily, parallel; fast twin only in node for CAS
ADR-013	S3 = SigV4 HTTP over Base TCP + conditional writes; TLS via operator proxy (Base TCP has no TLS)	accepted; native TLS twin only if proxies prove unworkable

ID	Topic	Default decision / mitigation
ADR-015	Strings as char lists are slow in the compiler	acceptable for compile time; optimize the lexer with byte lists
ADR-017	Dup-of-dup (label 0) unsoundness	documented limit + W-DUPDUP warning
ADR-020	Intra-net parallel reduction = `Waves` (§6.4.1): no atomics needed	phase 10; becomes canonical by genesis once `Waves ≡ Lifo` is proven on the golden set
ADR-022	Cross-orderer async messaging	out of scope v1 (`CrossOrderer` Fail)
ADR-023	Dynamic membership	out of scope v1
ADR-025	Binary HTTP bodies instead of hex	after confirming TCP.send byte semantics
ADR-026	Optional `fsx` twin (`fsync` , `chmod600` , `unlink`)	v1 without it: process-crash-safe, not power-loss-safe; GC by listing
ADR-027	Kernel-verifiable `commutes` certificate	open; v1 = h5c static check + author's `bend PROOF.bend`
ADR-028	JS secp twin via `@noble/secp256k1@3.2.0` (integrity pinned)	accepted
ADR-029	FAULT for ill-typed nullary pairs, charged the full `mana_limit`	accepted
ADR-030	Strict INET core vs upstream lazy IC `pure_hvm5_mini`	accepted: keep strict (metering, snapshots); upstream is a normal-form oracle for the shared fragment; revisit when upstream HVM5 is released

Top risks: (1) BendRT performance of a VM-in-Bend2. Mitigations: Arrays for heap and vars, flat tail loops, parallel batches. (2) `h5c` coverage of Bend2 syntax. Mitigation: a strict subset and golden tests. (3) Checker friction with Type/Data affinity in `Mach` / `Job` . Mitigation: thread the machine linearly, and every read returns the array (Appendix A).

---

29. CHANGES VS THE EARLIER HVM5-CLUSTER PLAN (MINI-KINDELIA)

Earlier	Now	Reason
v1 ran contracts via the official `bend` toolchain; nets modelled "as spec"	a **real INET reducer** in Bend2 runs every app	the goal says Bend2 → HVM5 → INET must be true
update = DUP-fork of a long-lived heap	persistent `Val` + lazy `EXT` + COPY rule: fork O(1), revert free	a strict DUP of data copies eagerly; EXT makes "almost free" literally true
state lives in the net heap	heap = per-job scratch arena; state = Merkle Val	snapshots, revert and parallel jobs become trivial
`Chan` -based ocap between contracts	synchronous `Call` with an affine no-reentrancy rule; `Sys` capability for the registry	smaller core; Base `Chan` stays in the node layer

Earlier	Now	Reason
four foreign effects	one (secp256k1: sign / pubkey / recover)	Base already provides time (`IO.now`) and randomness (`/dev/urandom` via `File`); bend-collections provides proven SHA-256 and Keccak-256
consensus-pow app package	three orderer apps; **Commutative mode skips consensus via a proof**	"consensus is an application"; CRDT apps need none
an event net (unspecified)	concrete DAG, canonical order, batches, conflict sets, policies, pruning, stream view	goal G3
storage via `Process.run` (`mkdir` / `mv` / `aws`) — **a bug: Base has no `Process`**	immutable hash-named objects + append-only `head.log` on disk; S3 = SigV4 over TCP with `If-None-Match` / `If-Match` ; in-process cluster test	only Base effects that exist (`bend base` verified)
ill-typed nullary pairs silently VOID; parallel reducer via atomics	FAULT (reverts, full mana); `Waves` scheduler with prefix-sum IDs, no atomics	hostile nets are visible; Base has no atomics
one monorepo (runtime + SDK + apps)	**three packages**: `hvm5` (runtime; this spec), `hvm5-sdk` , `hvm5-apps` , pinned by hub hash; a Val-level ABI (`kernel/abi.bend + Book.abi`) is the only coupling	a runtime repo like Kindelia's; apps and SDK are released independently; bend-collections layout (`src/ spec/ proofs/ tests/ tools/`)

---

30. KINDELIA SOURCE REVIEW: WHAT WE TOOK, ADAPTED, OR REJECTED

Source read: `github.com/kindelia/Kindelia@master` , including `WHITEBOOK.md` , `genesis.kdl` , `ROADMAP.md` , `default.toml` , `src/hvm.rs` (memory model, `run_io` , `run_statement` , `can_deploy` / `can_register` , limits, rollback), `src/api/{mod,server}.rs` , `src/cli.rs` , `src/events.rs` , `src/persistence.rs` .

30.1 DOMAIN MODEL

Kindelia	Here	Verdict
4 statements `ctr / fun / run / reg` , each optionally signed	`Deploy` (≈ `ctr` + `fun` : a book holds ctors *and* defs), `Invoke` (≈ `run` , but targeted), `Register` via the registry (≈ `reg`)	**adapted**: a book deploys types and functions together, because Bend2 modules are the unit
`fun … with {init}` : a function *is* a contract with one state slot	Instance = (book, state Val); `h5_init = with {…}`	**taken**
Every def immutable: "Can't redefine"	Books immutable (content-hashed); **instances** can be upgraded by the owner through the registry	**deviation, on purpose** (goal G6: redeploy / upgrade); history stays auditable because books never change
IO = `DONE TAKE SAVE CALL SUBJ FROM TICK TIME META HAX0 HAX1 GIDX STH0 STH1 + LOAD` (genesis)	`Op5 : Done` (+state), `Call` , `Ask` , `Emit` , `Read` , `Sys` , `Fail` ; `SUBJ/FROM/TICK/TIME/STH/GIDX` → `Env` fields + `Read` queries; `META/HAX` → dropped (block metadata/debug)	**adapted**: `TAKE/SAVE/LOAD` become affine state threading, so `Done{state}` is SAVE and taking the parameter is TAKE; `Ask` (external effects) and `Emit` (logs; Kindelia lists a `Log` primitive as TODO) are new
`CALL` switches subject to callee and sets caller as `FROM`	Same (§12.5 `env'`)	**taken**
`CALL` args must be *flat ctors of numbers* ("Kindelia's language is untyped … would allow a contract to pass an argument with an unexpected type to another, corrupting its state")	Schema-checked messages (§11.2): nested data allowed if the callee's schema describes it; VNet never allowed	**generalized**, same reason

Kindelia	Here	Verdict
Statement results `Ctr/Fun/Run{done_term, used_mana, size_diff, end_size}/Reg`	`Receipt{…, mana, bits, size_before, size_after, reply}`	**taken**
Statement index `tick<<60 | idx` and hash per statement (GIDX/STH)	`(epoch, pos)` in `Env.pos` + `Read{EventAt}`	**taken**
120-bit NUM, 72-bit names (12 six-bit letters), 128-bit cells	U32 NUM (BendRT native), String names ≤ 64 chars in the same 64-letter alphabet	**adapted** to Bend2 numerics
CTR arity ≤ 16, term depth ≤ 256, linearity checks	Book validation (§12.7)	**taken**
Genesis: tuples `T0…TG`, IO ctors, `Name`, `reg {}` of the empty namespace to the Namer	Genesis: registry book (with SysCap), orderer book(s), SDK prelude book, **empty namespace → Namer**	**taken** (tuples are unnecessary: Bend2 has pairs/ADTs)
Repo = VM + node + API + CLI + `genesis.kdl`; apps are deployed, not shipped	`hvm5` = core + compiler + kernel + node + CLI + `genesis.conf`; SDK and apps are separate hub packages (§5.1)	**taken**, and extended with content-hash pinning (Bend hub, bend-collections style)

30.2 RULES AND COSTS

Kindelia	Here	Verdict
`can_deploy`: empty name → no; undotted → anyone; dotted → owner of namespace	§14 table	**taken exactly** (the first draft of §14 was wrong: it let anyone *register* undotted namespaces)
`can_register`: owner of parent namespace; undotted parent = empty namespace = Namer	§14 table	**taken exactly**
Mana table: APP-LAM 2, APP-SUP 4, OP2-NUM 2, OP2-SUP 4, FUN-CTR 2+M, FUN-SUP 2+A, DUP-LAM 4, DUP-NUM 2, DUP-CTR 2+A, DUP-SUP 4/2, DUP-ERA 2	§6.7: ANNI 2 (= APP-LAM / DUP-SUP-equal), COMM 4 (= DUP-LAM / APP-SUP / DUP-SUP-diff), COPY 2 (= DUP-NUM), OPER 2 (= OP2-NUM), ERAS 2 (= DUP-ERA), CALL 2+nodes (= FUN-CTR 2+M), EXPD 2+nodes	**mapped rule-for-rule** onto interaction-net rules
Cumulative limits `4M × (block+1)` mana, `2048 × (block+1)` bits	§21.1 per-orderer epoch budgets	**taken** (this was missing in the first draft: per-event limits alone do not bound catch-up time)
Mana/size exceeded → revert; failure → abort	`Reverted` / `Rejected` receipts	**taken**
No coin, no fees; spam bounded by the block limit, a fee market may emerge off-protocol	Same; orderer apps may add PoW or allow-lists	**taken**
PoW, 1 s blocks, longest chain, reorgs	`hvm5-apps/src/orderer-pow` only	**moved out of core, and out of the runtime package** (goal G2)

30.3 STATE, PERSISTENCE AND ROLLBACK

Kindelia	Here	Verdict
State = pointers into one global, persistent 128-bit HVM heap; "zero-cost SSTORE" (save = store a pointer)	State = persistent Val; commit = keep a pointer; unchanged subtrees shared	**same idea, safer form**: no global heap to leak into, arenas are per job
Reversible heaps: up to 10 snapshots at exponentially spaced ages; lookups fall through to older heaps (`U128_NONE`); `absorb` merges	`history of Kernel0` at exponentially spaced epochs (§13.4.6); O(1) shares	**taken as policy**; the mechanism is unnecessary with persistent data
Heap buffers serialized per heap uuid (`memo` , `disk` , `file` , `arit` , `ownr` , `indx` , `stat`), append mode	Merkle CAS objects + manifest + WAL (§17)	**replaced**: content addressing gives incremental snapshots and S3 for free
~10× RAM overhead noted as a known cost	Avoided (no preallocated heap copies)	**improved**

30.4 API, CLI, NODE

Kindelia	Here	Verdict
HTTP: `/stats` , `/blocks` , `/blocks/<h>` , `/block-hash/<i>` , `/functions` , `/functions/<n>` , `/functions/<n>/state` , `/constructor/<n>` , `/reg/<n>` , `/peers[/all]` , `POST /run` (dry-run), `POST /publish` ; statements as hex of the protocol encoding inside JSON	§18.3: core 9 routes + `dry-run` , `instances/<n>/state` , `reg/<n>` , `stats` , `peers` , `log` ; hex H5B bodies, no JSON (Base has no JSON)	**taken** (dry-run, state, reg, stats and peers were added after this review); block routes → `events` / `heads` (DAG, not chain)
`418 TermTooBig` for oversized state	`413 + max_state_reply`	**taken**
WebSocket node events (AddBlock: Included / Reorg / MissingParent / TooLate / Computed, Mining, Peers, Heartbeat{peers, tip, blocks missing/pending/included, runtime mana/size current/limit/available})	`GET /v1/log` long-poll with the same event kinds; Heartbeat fields in `/v1/stats`	**taken** (transport adapted)
CLI: `test [--sudo]` , `serialize` , `deserialize` , `sign -s key [-E]` , `run-remote` , `publish` , `post` , `get fun code/state` , `get reg owner/list` , `get stats …` , `get peers` , `node start/clean` , `init`	§20	**taken almost 1:1**
Config resolution CLI > env > file > default; `default.toml` with `network_id` magic, `initial_peers` , `mining.enable` , `api.port` , `data.dir`	§18.5	**taken** (flat `key = value` instead of TOML)
UDP gossip with network magic; mempool "PleaseMineThisTransaction"	HTTP gossip of DAG events; there is no mempool, because the DAG *is* the pending set	**replaced**
secp256k1, Ethereum-style addresses, subject recovered from the signature	§19	**taken** (the first draft used sha256(pubkey) and 64-byte sigs; fixed)
Bit-level serialization (fixlen / varlen / list / number / 6-bit names / De Bruijn)	Byte-level H5B (§22)	**rejected for v1**, reasons in §22; ADR-004
Unsigned statements allowed (subject = 0)	All events signed	**rejected**: the DAG needs an author and seq for equivocation and conflicts

30.5 WHAT KINDELIA DOES *NOT* HAVE (SO THERE IS NOTHING TO COPY)

A registry *program*, upgrades, a DAG of inputs, external effects (`Ask`), conflict resolution, commutative (consensus-free) apps, multi-node result attestation, S3 snapshots. These come from the owner's goal (§0), and their design is ours alone.

APPENDIX A — BEND2 IDIOMS YOU WILL NEED (CHECK EACH AGAINST BEND GUIDE)

```
# A. branch on a computed Bool: helper that matches its parameter
def pick.go(x: U32, y: U32, c: Bool) -> U32:
  match c:
    case True{}:
      x
    case False{}:
      y
def pick(+x: U32, +y: U32) -> U32:
  pick.go(x, y, U32.is_lt(x, y))

# B. heap read/write on Array<U32> (reads hand the array back)
def Heap.rd(h: Array<U32>, +i: U32) -> Array<U32> & U32:
  Array.get(U32, h, i)
def Heap.wr(h: Array<U32>, i: U32, v: U32) -> Array<U32>:
  Array.set(U32, h, i, v)
# Non-Data cells (as in upstream pure_hvm5_mini): move out with Array.swap(T, a, i, placeholder) -> (a, old).
# Free list threaded through freed cells (upstream alloc.go): a freed cell stores the old free head;
# fresh ids come from the head, else from bump `nx`. Arrays sized by depth: [v : T^16n] = 2^16 slots.

# C. port pack/unpack
def Port.w0(t: Tag, +lab: U32) -> U32:
  (Tag.to_u32(t) + (lab * 16 : U32) : U32)
def Port.tag_of(+w0: U32) -> Tag:
  Tag.of_u32((w0 % 16 : U32))
def Port.lab_of(+w0: U32) -> U32:
  (w0 / 16 : U32)

# D. mutual recursion forbidden → one def + selector
type Sel is Data:
  SLink{}
  SBind{}

# E. actor loop (IO, @unsafe allowed only here)
@unsafe
def Actor.loop(k: Kernel, +mb: Chan(Cmd)) -> IO(Unit):
  do IO<Unit>:
    got : Maybe<&1, Cmd> <- Chan.recv(Cmd, mb)
    Actor.on(k, mb, got)          # helper matches got, applies, performs outputs, recurses

# F. balanced parallel map over a list (split in halves first)
def Par.seal(n: Nat, xs: List<&2, Val>) -> List<&2, Val>:
  # depth-indexed (n = log2 fuel, shrinks first); there is no List.split in Base: use +xs with take/drop
  # h = Nat.div(List.length(&2, Val, +xs), 2n); l = List.take(&2, Val, +xs, h); r = List.drop(&2, Val, xs, h)
  # a b = Par.seal(p, l) Par.seal(p, r)   (where n = 1n+p);  List.append(&2, Val, a, b)
```

APPENDIX B — GLOSSARY

Arena: the per-job INET heap (`Mach`). **Book**: an immutable, content-addressed compiled program. **Canonical scheduler**: LIFO with a LIFO free list; it defines outcomes when limits bind. **Checkpoint**: an orderer event that finalizes a cut. **Cut/Batch**: the ancestors of the heads / the events not yet settled. **EXT**: a lazy pointer to an immutable Val. **EXPD**: expanding an EXT by one layer. **Instance**: `(book pointer, state Val, pending asks)`. **Op5**: an IO operation read back from a normal form. **Orderer app**: a consensus as a Bend2 app. **Receipt**: the deterministic outcome of an event. **Seal**: filling in the Merkle hashes of new Val nodes. **Sys**: a privileged operation (registry only). **VNet**: a closed subnet stored as a value (closures, continuations).

End of specification. Implement Phase 0 now, then stop and report.

F.59 · bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.META-PROMPT.md

META-PROMPT — synthesize a complete, implementable runtime spec from knowledge files + deep web

research

Paste this as the first message. Fill the `{{…}}` slots. It was reverse-engineered from the process that produced `ALL-COMBINED-RUNTIME-SPEC.md`, and it builds in every correction that process needed afterwards:

- the prior-art source was not read, only summaries;
- reusable packages were not searched for (proven SHA-256/Keccak already existed in `bend-collections`);
- runtime, SDK and apps were mixed in one repo;
- types were used but never defined;
- permission rules were wrong;
- cumulative limits were missing;
- example code was out of sync with the types;
- Base defs that do not exist were used (`Process.run`, `List.split`, `Array.atomic.exch`), and storage, tests and deploy checks were built on them;
- ill-typed interactions were silently VOID instead of a deterministic FAULT;
- the parallel reducer assumed atomics that Base lacks;
- "confluence" was allowed to sound like consensus;
- external pins (tarball hashes, npm integrity, test vectors) were copied from another draft without checking.

Failure modes seen in weaker drafts, which you must avoid: falling back to "run compiled Bend2 with the official toolchain" instead of a real INET reducer; copying the prior art's consensus/wire formats (blocks, PoW, IPv4) into the core; accepting unsigned events; using background processes as phase gates; example code that is not valid Bend; claiming to have read PDFs/papers the fetch tool could not parse; parking core decisions in open ADRs instead of choosing a concrete default.

ROLE

You are **Victor Taelin** writing the normative spec for your own system. Write tersely, with confidence and precision. You think in interaction nets, confluence, linearity and costs. You do not hedge where the math is settled. You mark clearly where it is not. Your reader is a **smaller coding LLM** that will implement the system from this document alone, one phase at a time, with no access to you or to the internet. So every ambiguity you leave in becomes a bug, and every fact you failed to look up becomes a wrong guess.

INPUTS

- Knowledge files. Read **fully**, every line, before anything else:
  - `{{KNOWLEDGE_FILE_1}}` (language guide, stdlib, effects, INET theory, runtime, prior-art summaries)
  - `{{KNOWLEDGE_FILE_2}}` (an earlier plan for the same system; you are superseding it)
- Local toolchain: run `bend guide`, `bend --help` and `bend version`. The guide is authoritative for syntax, modules, **hub packages** (`import 0x<hash>/… as P`, `--publish`) and tooling.
- **Base verification protocol (mandatory, mechanical):**
  1. Before designing any IO, storage, test harness or parallel code, dump the real surface: `bend base IO`, `TCP`, `File`, `Chan`, `Array`, `List`, `String`, `Map`, `Nat`, `U32`, plus `bend base --types`. Write down what is **absent** (for example `Process`, rename/mkdir/delete/fsync, TLS, atomics, JSON), and design around it.
  2. Every Base name the spec mentions must pass `bend base <Name>`. Before finishing, extract all `X.name` tokens from the spec (`grep -oE '\b(IO|TCP|File|Chan|Array|List|String|Map|Nat|U32|Maybe|Result|Bool|Char)\.[a-z_.]+'`), run `bend base` on each one, and fix every miss. Record the result in the sources file.
  3. If a capability is missing, use one of three options, in this order: (a) redesign so it is not needed (for example immutable hash-named files + an append-only pointer log instead of rename); (b) a small foreign-effect twin on the FFI allow-list with an ADR; (c) operator infrastructure outside the process (for example a TLS proxy). Never assume it exists.
  4. Any example code must pass `bend --check-only` (write it to `/tmp/opencode/`, check it, then paste it). If that is impossible, label it as a sketch and list the exact reasons.
- Project instructions: `AGENTS.md` and global agent instructions. They are **binding**, and the spec must pass them on to the implementer.

- Seed URLs: `{{SEED_URLS}}` (for example `github.com/kindelia/Kindelia`, `github.com/Giulio2002/bend-collections`).
- Owner goal statement: `{{GOAL_STATEMENT}}`. Quote it verbatim, because it is normative.
- Output path: `{{OUTPUT_PATH}}`. Also write the research log to `{{OUTPUT_PATH%.md}}.SOURCES.md`.

SCOPE RULE (DECIDE IT BEFORE RESEARCHING)

The spec describes **one repository: the runtime**, the way Kindelia's repo is the VM + node + API + CLI + genesis file. Anything that *runs inside* the runtime, or that *app authors import*, lives in **separate packages** (SDK, system apps such as registry and orderers, examples), versioned and pinned by content hash. The runtime spec owns:

- the **ABI** between runtime and programs (constructor names, order and fields, plus a version number);
- the **required behavior** of system apps (types + rules + laws);
- the **genesis format**, which references app releases by hash instead of inlining their code.

It does **not** contain SDK or app source trees. Runtime tests use tiny hand-written fixtures, and integration with the real packages is a separate, pinned e2e gate.

DEEP RESEARCH PROTOCOL (MANDATORY; DO IT BEFORE DESIGNING)

Use `websearch + webfetch`. Prefer `raw.githubusercontent.com/<owner>/<repo>/<branch>/<path>` for files, and fall back to the GitHub tree pages to list directories. Save large pages and grep them rather than skimming.

1. **Primary sources behind every summary.** For each project the knowledge files mention (Kindelia, HVM, HVM2/HVM3/HVM4/HVM5, Bend, BendRT, interaction combinators, Interaction Calculus), open the original repos and read: README/whitepaper, genesis/bootstrap files, config defaults, the runtime core, the API server + types, the CLI, event/log types, persistence. From each one, extract the exact rules, constants and formats.
2. **Follow links down, at least two levels.** From every README, paper and source file, follow links to referenced repos, papers, issues, PRs, the docs folders and `tools/` or config pins. Record each hop. Stop a branch only when it yields nothing relevant to domain types, API, limits, identity, persistence, packaging or reusable code.
3. **Discover *new* sources, not only the seeds.** Run searches like these and open every promising hit:
   - `site:github.com bend2 OR "bend 2" language:bend`, `"import 0x" bend hub`, `bend-lang package`
   - `HigherOrderCO / VictorTaelin` repos: HVM*, Bend*, Kind*, IC/interaction-calculus, `pure_hvm5_mini`
   - Bend ecosystem libraries: collections, crypto (sha256, keccak, blake, secp256k1), codecs, JSON, HTTP
   - prior art for each subsystem: DAG-based ordering (Narwhal/Bullshark, hashgraph), CRDT apps, content-addressed snapshots, Ethereum address derivation / recoverable signatures (EIP-155, EIP-191), deterministic metering For each hit, check it is recent and compatible with the pinned toolchain version.
4. **Reuse before you write.** For every library-level need (hashes, containers, math, codecs), search for an existing, preferably *proven*, package and read its API signatures in source. Pin it by hub hash, name the exact import path and function signature in the spec, and state which toolchain version it requires (and adopt that pin). Only spec your own implementation when nothing suitable exists, and say you searched.
5. **Adopt ecosystem conventions.** Copy the repo layout, the spec/proof split, the toolchain pin file, the publish flow and the test/benchmark layout from the most mature package in the ecosystem, and cite it.
6. **Verify claims by reading code, not prose.** Permission checks (for example, deploy vs register), signature and address formats, limit formulas and API status codes must be quoted from the function that implements them.
7. **Verify every external pin yourself.** Tarball hashes: download the exact artifact and run `sha256sum` (release *assets* and `archive/refs/tags` tarballs have different hashes, so say which one you pinned). npm packages: `npm view <pkg> version dist.integrity`. Hash and address vectors: recompute them (`openssl dgst -sha256 / -keccak-256`). Protocol headers and status codes: quote them from the official doc page. Never copy a pin from another draft or from memory.
8. **Research log** (`*.SOURCES.md`): URL → what was read → facts extracted → where the spec uses them → verdict. Include dead ends, and an explicit **"not read"** list. Mark every fact **(S)** with its URL/file + function. A source you could not parse (a PDF, a 404) goes into "not read", never into the citations.
9. **Adversarial review.** If other drafts or reviews exist, score them against this checklist, take their verified improvements, and record in the ledger which ideas came from where.

DESIGN PROCESS

1. **Extract.** Break the goal into atomic elements G1…Gn. List every hard constraint of the language and runtime (syntax limits, termination, affinity, numeric types, no mutual recursion, what the runtime is and is *not*). List what the prior plan got right and wrong.
2. **Decide the key ideas** (3–6) that make the design small and correct, and derive every other decision from them.

3. **Write in parts** (about 7 files of ≤ 300 lines in `/tmp/opencode/`), then concatenate. Never write the whole document in one call.
4. **Self-audit** with the checklist, fix what it finds, and only then report.

REQUIRED DOCUMENT STRUCTURE (NUMBERED; EVERY SECTION IS MANDATORY)

0. **Header.** Title, date, exact sources: knowledge files + every primary repo/file read + reused packages with their hashes. State that this supersedes the earlier plan. Add a "HOW TO USE THIS FILE" box: the scope (runtime repo only), the toolchain version, one phase at a time, and write an ADR stub and stop when something is unspecified.
1. **Goal** (verbatim quote) + a **goal → section map** table in which each element has exactly one home.
2. **Key ideas**, each with its consequences for cost, revert, fork, snapshots and persistence.
3. **Honesty anchors. (S)** sourced vs **(D)** decided. Cover what the runtime really is, what upstream artifacts exist, how packages are shared, what is *not* reimplemented, and which libraries are reused.
4. **Ground rules for the implementer.** Language-only policy with a minimal FFI allow-list, "run the guide first", laws-first with proof gates, concrete laws as golden tests, a purity boundary, where to parallelize, language pitfalls, and "two failed attempts → ADR and stop".
5. **Architecture.** Strict layers, with in-package and out-of-package layers visibly separated. Data flow of one invocation. In-node process model.
6. **Packages and repository layout.**
   - A package table: name, contents, depends on, specified where.
   - Coupling rules: no imports of SDK/apps from the runtime's `src/`; the ABI file; a conformance test.
   - Genesis as data (a pinned app-release hash + a builder command).
   - Test fixtures vs pinned e2e.
   - The single deps file holding all hub hashes.
   - The toolchain pin.
   - The full runtime tree in ecosystem layout (`src/ spec/ proofs/ tests/ benchmarks/ tools/ docs/ main.bend LAWS.bend PROOF.bend`), with a one-line purpose per file and the path convention stated.
   - Sibling packages in outline only.
   - The exact build, run and publish commands.
7. **Core machine**, bit-exact: tags, memory layout, wiring, redex bag, canonical scheduler, a **total** rule table in which ill-typed pairs are a deterministic **FAULT** (a halt that reverts the event and has a fixed charge), not a silent no-op; exact rewiring; metering (what is free, and why); a termination device; a **deterministic parallel scheduler that uses only the parallel primitives Base really has** (for example sorted waves + prefix-sum ID ranges + link intents when there are no atomics); a **determinism theorem with its limits**, including an explicit "confluence ≠ consensus ≠ data availability" paragraph; blobs; readback; canonical form + hashing; a text format; the core public API. The state model must make revert, fork and snapshot cheap (O(touched), not O(state)). Explain how.
8. **Values and data encoding.** The persistent value type, the constructor encoding (compiler and runtime agree byte for byte), hashing.
9. **Programs/books.** Format (including the ABI version field) and validation.
10. **Compiler.** Accepted subset, hub-import resolution, how the SDK prelude is mapped, passes, match compilation, known limits, CLI.
11. **SDK ABI**, normative here but implemented elsewhere: the effect monad, the operation tree, env fields (each mapped to its prior-art equivalent), effects, read queries, system calls, ABI versioning.
12. **Manifest and schemas.** Untrusted input never reaches a `match` unchecked.
13. **Kernel. Every** domain type fully defined (events, receipts, instances, state, budgets, **params**, **genesis**), a single pure `apply(state, input) -> state & outputs`, per-operation interpreter semantics (including caller/callee identity in nested calls), instance and program lifecycles.
14. **Input DAG.** Order, batches, settlement, conflict policies, append-only presentation, a proof-gated no-consensus mode, pruning, reorgs with a snapshot-history policy.
15. **Registry.** Required types + a **permission table quoted from prior-art code** + laws; its code lives in the apps package.
16. **Consensus as apps.** The runtime's knowledge (event kind, manifest field, role loader) + the required behavior of each orderer.
17. **Cluster sync of events and reductions.**
18. **Persistence.** Store interface (disk + S3), layout, snapshot algorithm as a pure lib + IO driver, WAL, CAS, recovery. Build it only from verified Base file operations. Every write is immutable (content-named, verified on read) or an append (checksummed frames; the last valid record wins). S3 is compare-and-swap via conditional writes (`If-None-Match: *`

/ `If-Match: <etag>` , with 412/409 handling), with no CLI tools. State the durability level honestly (process crash vs power loss), and include a torn-write recovery test.

19. **Node shell and minimal API.** Routes (core + read-only conveniences from prior art), effect executor, config with resolution order and a network magic id.
20. **Identity and crypto.** Web3-compatible keys and addresses; recoverable signatures over a wallet-compatible domain (ERC-191 `personal_sign` of a network-bound digest); low-S and v rules; the minimal FFI with the native library pinned by a **verified** artifact hash and the JS twin pinned by npm integrity; hashes from a reused library; recomputed test vectors.
21. **CLI.** Including the genesis builder and an offline scenario runner.
22. **Limits and params.** Per-event **and cumulative per-epoch** budgets, with a source for each value.
23. **Canonical encoding.**
24. **Security model.**
25. **Parallelism.**
26. **Worked example end to end.** The app lives in the apps package and imports the SDK by hash; its code uses **exactly** the defined types.
27. **Laws and tests per module.** Runtime modules, plus rows for what the sibling packages must prove.
28. **Phased plan.** Gates are exact commands, and the plan says when sibling releases are needed.
29. **Risks, open questions, ADR seeds.** Include the package-split ADR.
30. **Changes vs the earlier plan.**
31. **Prior-art review.** Per-item verdict tables (**taken / adapted / generalized / replaced / rejected / deviation on purpose**) with reasons and file/function citations, "what prior art does not have", and **reused ecosystem packages**.

- **Appendix A**: language idioms (each one checked against the guide). **Appendix B**: glossary.

QUALITY BARS

- **Implementable without guessing, and without the internet.** Give exact layouts, constants, routes, paths, import lines and signatures. Anything unspecified gets an "ADR-NNN open" marker, but **core decisions get a concrete default**. An open ADR must never block phases 0–7.
- **Small core.** Few node kinds, rules and routes. Every feature beyond the goal must justify itself.
- **The goal is taken literally.** Consensus is outside the core, IO goes through the monad, conflicts are decided by the app, and snapshots are a library.
- Mark **(S)** vs **(D)** throughout. Each env field, operation, cost, limit, route and CLI command names its prior-art counterpart.
- **Consistent.** Every type is defined once. Examples, pseudo-code and routes match those definitions field for field.
- Keep the prose dense and the tables plentiful. Put code in the target language's real syntax. Leave out filler.

SELF-AUDIT CHECKLIST (FIX EVERY "NO")

- <input disabled="" type="checkbox"> Every goal element maps to exactly one section, and that section delivers it.
- <input disabled="" type="checkbox"> The research log exists, with ≥ 2 link levels followed from each seed and ≥ 5 newly discovered sources evaluated.
- <input disabled="" type="checkbox"> The primary source was read for every prior-art summary, and §31 cites files and functions.
- <input disabled="" type="checkbox"> Every library need was searched for; reused packages are pinned with import path, signature and toolchain version.
- <input disabled="" type="checkbox"> The spec covers **only** the runtime repo. SDK and apps are separate packages, coupled only by the ABI, and the runtime `src/` never imports them.
- <input disabled="" type="checkbox"> The layout follows the ecosystem's most mature package, which is cited.
- <input disabled="" type="checkbox"> Permission rules match the prior-art code exactly, or the deviation is labelled.
- <input disabled="" type="checkbox"> Identity is Web3-compatible (address derivation vector, key file format, 65-byte recoverable sigs).
- <input disabled="" type="checkbox"> Per-event and cumulative per-epoch limits are both present, with sources.
- <input disabled="" type="checkbox"> The rule table is total, with FAULT for ill-typed pairs; metering is scheduler-independent; the determinism theorem states its limits, including confluence ≠ consensus ≠ data availability.
- <input disabled="" type="checkbox"> **Every Base name in the spec passed `bend base <Name>`** (mechanical grep + check, logged). No `Process` , rename, fsync, TLS or atomics is assumed unless verified.
- <input disabled="" type="checkbox"> The parallel scheduler uses only verified primitives, and its IDs do not depend on the thread count.

- <input disabled="" type="checkbox"> Storage uses only immutable writes + appends (+ S3 conditional writes); durability level stated; torn-write test present.
- <input disabled="" type="checkbox"> Every external pin (tarball sha256, npm integrity, vectors) was recomputed by you, and the command is logged.
- <input disabled="" type="checkbox"> Example code passed `bend --check-only`, or is labelled as a sketch with reasons.
- <input disabled="" type="checkbox"> Every test harness (cluster, e2e) runs in-process or through verified primitives. No gate is a background process.
- <input disabled="" type="checkbox"> Every referenced type is defined (grep each `Name{`), including Genesis, Params, Budget and the ABI version.
- <input disabled="" type="checkbox"> The example, interpreter and API use the current field lists; the example imports the SDK by hash.
- <input disabled="" type="checkbox"> Nested-call identity semantics are specified.
- <input disabled="" type="checkbox"> Dry-run, capped state readback, stats, peers and an event log each exist, or are rejected with a reason.
- <input disabled="" type="checkbox"> Config resolution order, network magic, toolchain pin and publish command are all specified.
- <input disabled="" type="checkbox"> Every phase gate is a runnable command; `bend PROOF.bend` runs before every commit.
- <input disabled="" type="checkbox"> Every AGENTS.md instruction is passed on to the implementer.
- <input disabled="" type="checkbox"> The document states that it supersedes the earlier plan and lists every change; no leftover draft text.

FINAL REPORT (TO THE USER, SHORT)

Give the path, the line count, the section list, the key decisions, what was taken from prior art vs changed, the reused packages (with hashes), open ADRs, and the research-log path. Be honest about what you could **not** read or verify.

F.60 · `bend-runtime/knowledge/ALL-COMBINED-RUNTIME-SPEC.SOURCES.md`

Research ledger for ALL-COMBINED-RUNTIME-SPEC.md

Every row is something that was actually read or run. Items that were **not** read are listed at the end, so nobody mistakes them for sources.

LOCAL

Source	Facts extracted	Used in
`bend2-knowledge/ALL-COMBINED.md`	Bend2 guide, Base, effects, INET theory, BendRT, Kindelia summaries, upgrade patterns	whole spec
`bend2-knowledge/ALL-COMBINED-MINI-KINDELIA.md`	earlier hvm5-cluster plan	§29
`bend --help`, `bend guide` (2.0.25)	single command; `--check-only`, `--publish`; `import 0x<hash>/… as P` from the hub	§2, §5
`bend base IO / TCP / File / Chan / Array / List / String / Nat`	exact Base surface; **no `Process`**, no rename/mkdir/delete/fsync/TLS/atomics; `TCP.send` takes `String`; `IO.random_u32` exists; `Array.fork/join` exist; `List.split` does not; `String.lines/split` exist	§3.1, §2, §6.4.1, §17, App. A
script over every `X.name` in the spec → `bend base`	all 22 Base names resolve, except `List.split` (fixed)	App. A
`openssl dgst -sha256 / -keccak-256`	vectors: sha256("abc"), keccak256("abc"), keccak256("")	§19
`npm view @noble/secp256k1 version dist.integrity`	3.2.0 + integrity sha512	§19

WEB

URL	Facts extracted	Used in	Verdict
github.com/kindelia/Kindelia (tree), `WHITEBOOK.md`, `ROADMAP.md`, `default.toml`	statements, IO, costs, cumulative limits, genesis, `network_id`, config	§10, §14, §21, §30	taken/adapted
`src/hvm.rs` (`run_io`, `run_statement`, `can_deploy`, `can_register`, `get_mana_limit`, `get_size_limit`, `check_linear`, limits)	IO_CALL subject/from; permission rules; `4M×(b+1)`, `2048×(b+1)`; arity 16; depth 256; readback `1<<16`	§10, §12.5, §14, §21	taken
`src/api/{mod,server}.rs`, `src/cli.rs`, `src/events.rs`, `src/persistence.rs`	routes, 418 TermTooBig, CLI commands, config resolution order, node event kinds, heap persistence	§18, §20, §30	taken/adapted/replaced
github.com/Giulio2002/bend-collections: README, `main.bend`, `tools/toolchain.json`, `src/crypto/sha/{sha256,packed/sha256}.bend`, `src/crypto/keccak/keccak.bend`	layout `src/ spec/ proofs/ tests/ benchmarks/ tools/`; hub hash `0x9ee2e9a299991dcc089fe22c7f3ceb5f`; Bend 2.0.28 pin; `sha256(words, byte_length)`, `keccak256(a, length)` signatures	§5, §19	reused
github.com/bitcoin-core/secp256k1 release v0.8.0 (API), `include/secp256k1_recovery.h` from the tarball	published 2026-08-03; release-asset sha256 `dd685546…132b`, which differs from the archive tarball `eb52b0e9…`; recoverable sign/recover/serialize API	§19	pinned
eips.ethereum.org/EIPS/eip-191	`"\x19Ethereum Signed Message:\n" + len(message)`	§19	taken
docs.aws.amazon.com/AmazonS3/latest/userguide/conditional-writes.html	`If-None-Match: *`, `If-Match: <etag>`, `412 Precondition Failed`, `409 Conflict`	§17.1	taken
raw HVM2 `src/hvm.rs` (rule constants, `get_rule`)	rule names LINK/CALL/VOID/ERAS/ANNI/COMM/OPER/SWIT = our naming lineage	§6	confirms lineage
github.com/bendlang/bend `demos/pure_hvm5_mini` @ `574b6d39` (`main.bend`, `LAWS.bend`, `PROOF.bend`)	upstream HVM5 = lazy IC evaluator (port of hvm5.c); no NUM/OPR; bit-run constructors; env `Array<Cell>` + free list via `Array.swap`; `[v : T^d]` arrays; laws cover lexer/parser/printer only (no confluence proof)	§2 anchor, ADR-030, App. A, `ic_oracle` test	oracle + idioms taken; core model kept
Second-opinion review (two external LLM drafts, 2026-09-26)	caught the `Process` bug; suggested conditional writes, a FAULT row, deterministic parallel IDs, EIP-191, a libsecp pin, and "confluence ≠ consensus"	§3, §6, §17, §19	each item verified independently before adoption

NOT READ (DO NOT CITE AS SOURCES)

HVM3/HVM4 sources, Interaction-Calculus repo, `hvm5.c` itself (only its Bend port was read), BendRT paper PDF, Narwhal/Bullshark/Hashgraph papers, SQLite WAL docs, RFC 8949, EIP-155 text, Bend hub wire protocol and hash algorithm. If a later revision relies on any of these, read it first and add a row.

PART G · HVM5 runtime specifications (bend-runtime/hvm5, verbatim)

The actual HVM5 spec tree. v1: system, calculus, HVM4 execution, contract compiler, Bend2→inet support, self-hosted compiler, core IR, homoiconic code, types/isolation, subspaces, registry contract, lifecycle, cluster, API/CLI, security, benchmarks, acceptance, traceability + the goal/system/type/space/lifecycle/code/risk maps + plan/gates and the legacy v0 spec, the net-v0 note and the lattice-v0 design top files. v2: a newer rewrite (README, review, overview, calculus, runtime, compiler, code-as-data, islands, wire formats, trust, map, plan). reference/hvm5-mini: the early Bend port (main/laws/proof) kept as history. These are the documents that say what HVM5 is: compiled interaction nets reduced on BendRT, with contracts as inet packages.

Source root: `bend-runtime/hvm5` · **66 files · 423.6 KB**

▼ In this part (file list)

- **G.1** · `bend-runtime/hvm5/README.md`
- **G.2** · `bend-runtime/hvm5/version1/README.md`
- **G.3** · `bend-runtime/hvm5/version1/spec/00-system.md`
- **G.4** · `bend-runtime/hvm5/version1/spec/01-calculus.md`
- **G.5** · `bend-runtime/hvm5/version1/spec/01a-hvm4-execution.md`
- **G.6** · `bend-runtime/hvm5/version1/spec/01b-contract-compiler.md`
- **G.7** · `bend-runtime/hvm5/version1/spec/01c-bend2-to-inet-support.md`
- **G.8** · `bend-runtime/hvm5/version1/spec/01d-self-hosted-compiler.md`
- **G.9** · `bend-runtime/hvm5/version1/spec/01e-core-ir.md`
- **G.10** · `bend-runtime/hvm5/version1/spec/02-homoiconic-code.md`
- **G.11** · `bend-runtime/hvm5/version1/spec/03-types-isolation.md`
- **G.12** · `bend-runtime/hvm5/version1/spec/04-subspaces.md`
- **G.13** · `bend-runtime/hvm5/version1/spec/05-registry-system-contract.md`
- **G.14** · `bend-runtime/hvm5/version1/spec/06-lifecycle.md`
- **G.15** · `bend-runtime/hvm5/version1/spec/07-cluster.md`
- **G.16** · `bend-runtime/hvm5/version1/spec/08-api-cli.md`
- **G.17** · `bend-runtime/hvm5/version1/spec/09-security.md`
- **G.18** · `bend-runtime/hvm5/version1/spec/10-benchmarks.md`
- **G.19** · `bend-runtime/hvm5/version1/spec/11-acceptance.md`
- **G.20** · `bend-runtime/hvm5/version1/spec/12-traceability.md`
- **G.21** · `bend-runtime/hvm5/version1/spec/legacy/hvm5-inet-contract-runtime-spec-v0.md`
- **G.22** · `bend-runtime/hvm5/version1/spec/README.md`
- **G.23** · `bend-runtime/hvm5/version1/maps/00-goal.md`
- **G.24** · `bend-runtime/hvm5/version1/maps/10-system.md`
- **G.25** · `bend-runtime/hvm5/version1/maps/20-types.md`
- **G.26** · `bend-runtime/hvm5/version1/maps/30-spaces.md`
- **G.27** · `bend-runtime/hvm5/version1/maps/40-lifecycle.md`
- **G.28** · `bend-runtime/hvm5/version1/maps/50-code.md`
- **G.29** · `bend-runtime/hvm5/version1/maps/60-risks.md`
- **G.30** · `bend-runtime/hvm5/version1/maps/README.md`
- **G.31** · `bend-runtime/hvm5/version1/plan/00-plan.md`
- **G.32** · `bend-runtime/hvm5/version1/plan/01-slices.md`
- **G.33** · `bend-runtime/hvm5/version1/plan/02-gates.md`
- **G.34** · `bend-runtime/hvm5/version1/plan/03-risks.md`
- **G.35** · `bend-runtime/hvm5/version1/plan/legacy/hvm5-bendrt-contract-runtime-plan-v0.md`
- **G.36** · `bend-runtime/hvm5/version1/plan/README.md`
- **G.37** · `bend-runtime/hvm5/version1/bench/results/w3-2026-10-20.md`
- **G.38** · `bend-runtime/hvm5/version1/compiler/README.md`

- **G.39** · bend-runtime/hvm5/version1/contracts/README.md
- **G.40** · bend-runtime/hvm5/version1/src/README.md
- **G.41** · bend-runtime/hvm5/version1/legacy/net-v0/readme.md
- **G.42** · bend-runtime/hvm5/version1/legacy/lattice-v0/fitness.md
- **G.43** · bend-runtime/hvm5/version1/legacy/lattice-v0/gaps.md
- **G.44** · bend-runtime/hvm5/version1/legacy/lattice-v0/goal.md
- **G.45** · bend-runtime/hvm5/version1/legacy/lattice-v0/graph.md
- **G.46** · bend-runtime/hvm5/version1/legacy/lattice-v0/invariants.md
- **G.47** · bend-runtime/hvm5/version1/legacy/lattice-v0/owners.md
- **G.48** · bend-runtime/hvm5/version1/legacy/lattice-v0/placement.md
- **G.49** · bend-runtime/hvm5/version1/legacy/lattice-v0/rejects.md
- **G.50** · bend-runtime/hvm5/version1/legacy/lattice-v0/seams.md
- **G.51** · bend-runtime/hvm5/version2/README.md
- **G.52** · bend-runtime/hvm5/version2/REVIEW.md
- **G.53** · bend-runtime/hvm5/version2/spec/00-overview.md
- **G.54** · bend-runtime/hvm5/version2/spec/01-calculus.md
- **G.55** · bend-runtime/hvm5/version2/spec/02-runtime.md
- **G.56** · bend-runtime/hvm5/version2/spec/03-compiler.md
- **G.57** · bend-runtime/hvm5/version2/spec/04-code-as-data.md
- **G.58** · bend-runtime/hvm5/version2/spec/05-islands.md
- **G.59** · bend-runtime/hvm5/version2/spec/06-formats.md
- **G.60** · bend-runtime/hvm5/version2/spec/07-trust.md
- **G.61** · bend-runtime/hvm5/version2/spec/attic/05-islands-leader.md
- **G.62** · bend-runtime/hvm5/version2/maps/00-map.md
- **G.63** · bend-runtime/hvm5/version2/plan/00-plan.md
- **G.64** · bend-runtime/hvm5/reference/hvm5-mini/main.bend
- **G.65** · bend-runtime/hvm5/reference/hvm5-mini/laws.bend
- **G.66** · bend-runtime/hvm5/reference/hvm5-mini/proof.bend

G.1 · bend-runtime/hvm5/README.md

HVM5

This directory holds the HVM5 lineage. It is versioned because the runtime specification, the graph calculus, and the cluster protocol will change between versions. Do not edit an old version to make a new experiment pass.

```
hvm5/
  reference/
    hvm5-mini/       vendored pure-HVM5 evaluator written in Bend 2
  version1/
    spec/            normative HVM5 v1 + cluster contract specification
    plan/            implementation slices and gates
    maps/            goal, system, type, space, lifecycle, code, risk maps
    src/             runnable Bend 2 homoiconic DSL, cluster, and CLI
    tests/           Bend 2 unit, adversarial, and end-to-end tests
    bench/           reduction and cluster benchmarks
    scenarios/       executable ten-node simulation
    bend/            Bend 2 laws and proof gate
    contracts/       sample HVM5/Bend 2 contract terms
    legacy/          early v0 net model, lattice, and TS oracle, kept as evidence
```

VERSIONS

Version	Status	Meaning
`reference/hvm5-mini`	early draft	A port of `hvm5.c`; kept as reference and proof evidence, not v1
`version1`	active	Bend 2 homoiconic DSL, partitioned contract runtime, ten-node simulation

READ ORDER

1. `version1/spec/00-system.md` — what is being built.
2. `version1/maps/00-goal.md` — the goal and observables.
3. `version1/spec/01-calculus.md` — the net calculus.
4. `version1/spec/02-homoiconic-code.md` — code as data and fork/merge.
5. `version1/spec/03-types-isolation.md` — capability types and isolation.
6. `version1/spec/04-subspaces.md` — partitioned nets and authority.
7. `version1/spec/06-lifecycle.md` — deploy, fork, test, merge.
8. `version1/plan/00-plan.md` — how the implementation is staged.

The old `version1/spec/legacy/` and `version1/plan/legacy/` files are Draft 0. They are useful history, but they are not normative.

G.2 · `bend-runtime/hvm5/version1/README.md`

HVM5 v1 — partitioned homoiconic contract runtime

> **Superseded by `../version2`.** v1 stays runnable as evidence. Its reducer is a name-substitution tree evaluator (not an inet), and its fork, hashing and partition models are corrected in `../version2/REVIEW.md`.

HVM5 v1 is a reference runtime for one live interaction net that is partitioned into typed subspaces, replicated across light nodes, and updated by replacing graphs rather than by mutating running code in place.

The short version:

```
signed package graph
  -> system contract admission
  -> subspace graft
  -> shared invoke capability + linear update capability
  -> send = connect a message graph and reduce under fuel
  -> fork = dup the live state/code graph
  -> test = reduce and replay on the shadow branch
  -> merge = collapse the superposition / flip the alias atomically
```

The design is intentionally not a blockchain. It is a single logical net with explicit trust boundaries, replicated for availability. Consensus, money, and finality are separate layers.

NORMATIVE SET

Document	Scope
`spec/00-system.md`	system definition, goals, non-goals, vocabulary
`spec/01-calculus.md`	agents, ports, wires, rules, confluence, fuel
`spec/02-homoiconic-code.md`	package graphs, holes, dup/sup, fork/merge
`spec/03-types-isolation.md`	capability types, isolation relation, proofs
`spec/04-subspaces.md`	space tree, authority, replication, failure
`spec/05-registry-system-contract.md`	keys, envelopes, registry, genesis contract
`spec/06-lifecycle.md`	deploy, send, fork, test, merge, rollback
`spec/07-cluster.md`	ten-node topology, log, quorum, recovery
`spec/08-api-cli.md`	CLI and wire operations
`spec/09-security.md`	threat model and invariants
`spec/10-benchmarks.md`	how speed is measured

Document	Scope
spec/11-acceptance.md	executable acceptance matrix
spec/12-traceability.md	knowledge-base idea to requirement map

IMPLEMENTED REFERENCE

The runtime in `src/` is Bend 2, compiled by BendRT. It is not a TypeScript mirror. The homoiconic core is a Bend 2 DSL interpreter:

```
Term = Var | Lam | App | Dup | Sup | Era | Num | Add | Eq | Tag | Match | Ref
```

Code, state, messages, patches, and programs are all `Term` values. The interpreter is the runtime; BendRT executes it. The graph calculus in the spec is the semantic model the term language elaborates to.

The v1 Bend model currently implements:

- a Bend 2 contract parser/compiler for the v1 subset (`src/bendc.bend`), plus a bootstrap frontend (`compiler/bootstrap/compile.ts`);
- a canonical inet package format and parser (`src/hvm5/parse.bend`);
- a Bend 2 inet reducer with substitution, a CEK-style machine, fuel, and the core IC interactions (`src/hvm5/eval.bend`);
- a checked Bend 2 contract source (`contracts/counter.bend`);
- a deploy-shaped pipeline: source → inet package → reducer → state.

The end-to-end command is:

```
bun hvm5/version1/compiler/bootstrap/compile.ts \
  hvm5/version1/contracts/counter.bend \
  build/counter.inet
bend hvm5/version1/src/hvm5/run.bend build/counter.inet
# done Counter{2} fuel=0
```

The official-frontend backend path (no second parser):

```
bun hvm5/version1/tools/emit-inet.ts \
  hvm5/version1/contracts/counter.bend \
  build/counter-official.inet
bend hvm5/version1/src/hvm5/run.bend build/counter-official.inet
# done Counter{2} fuel=0
```

This emitter imports the vendored official Bend 2 parser/checker and lowers the checked first-order contract fragment to inet.

Native BendRT build of the same reducer:

```
bend hvm5/version1/src/hvm5/run.bend -o build/run
./build/run build/counter.inet
```

Deploy from a real Bend 2 source file compiles automatically; there is no emit flag:

```
bend hvm5/version1/src/deploy.bend hvm5/version1/contracts/counter.bend
# DEPLOY Counter{0} PKG <u32> DIGEST <u32>
```

The full ten-node model:

```
bend hvm5/version1/src/sim.bend
# NODES 10
# PUB Counter{4}
# PRIV Counter{3}
# N4 Counter{3}
# ISOLATED True
# PUB-OK True
# PRIV-OK True
# CATCHUP-OK True
# REPLICAS-OK True
# QUORUM-OK True
# BRIDGE-OK True
# CAP-OK True
# PASS
```

The same model is also a gate and a proof-gated topology:

```
bend hvm5/version1/tests/run.bend
bend hvm5/version1/PROOF.bend
```

Fork, test, and merge a candidate transition. The candidate compiles to a `Term` at fork time; merge swaps the already-compiled code and tested state:

```
bend hvm5/version1/examples/fork_merge.bend
# AFTER Counter{100}
```

Benchmark:

```
bend hvm5/version1/bench/bench.bend
# BENCH sends=100 ms=...
# STATE Counter{100}
```

The following are still future slices: package hashing, native signed deploy, a real process/network cluster, WAL persistence, and the full HVM4 rule set.

`hvm5/reference/hvm5-mini/` is kept as an early-draft reference from the Bend repository. It is not the v1 semantics and is not imported by v1.

The honest limits are listed in `spec/11-acceptance.md` and `plan/03-risks.md`.

G.3 · bend-runtime/hvm5/version1/spec/00-system.md

00 — System definition

Status: normative for HVM5 v1.
Supersedes: `legacy/hvm5-inet-contract-runtime-spec-v0.md`.

1. WHAT THE SYSTEM IS

HVM5 v1 is one logical interaction net, partitioned into addressable subspaces and replicated over light nodes. A deployed Bend 2 package becomes a graph subnet with explicit capability roots. A send connects a message graph to one of those roots and lets local interaction rules reduce it under fuel. A fork duplicates a live instance graph; a merge collapses the chosen branch and atomically changes the visible root.

The system is homoiconic in the strict sense:

```
code   = graph
state  = graph
patch  = graph
fork   = DUP of a graph
merge  = collapse of a SUP / atomic root alias flip
```

There is no separate bytecode format, AST format, storage format, and migration format at the semantic layer. There is one graph encoding, one hash, and one set of rules.

2. ENTITIES

Entity	Definition
`Package`	A closed graph plus a manifest and an export table, addressed by hash
`Instance`	A grafted package graph with a live state root and an owner
`Capability`	A typed reference to a root: `shared` or `linear`
`Space`	A path in the subspace tree, with an explicit listener set
`Node`	A light runtime that listens to one or more spaces and reduces their active pairs
`System contract`	The genesis instance that authorizes registrations and deploys
`Envelope`	A signed, domain-separated request (`deploy` , `update` , `merge`)
`Receipt`	The old hash, new hash, fuel used, status, and sequence of a send

`PackageHash` identifies content. It never grants authority. A `Capability` grants authority. An `Identity` identifies a signer. None of these three is interchangeable.

3. LAYERS

```
L7  CLI / client / signing wallet
L6  System contract + package registry + name pointers
L5  Cluster: subspace routing, replication, quorum, catch-up
L4  Membrane: capabilities, linearity, epochs, receipts, quotas
L3  HVM5 graph kernel: agents, wires, rules, labels, fuel, partitions
L2  Contract ABI: state/message graphs, transition roots, reply roots
L1  Host: Ed25519, SHA-256, storage, clock, sockets (trusted, outside graphs)
L0  Process / OS / container
```

L0–L1 are trusted. L3–L4 are the security boundary. L5–L7 are untrusted clients of the same capability protocol.

4. DESIGN PRINCIPLES

1. **Connectivity is authority.** A subnet interacts only over wires it holds. No wire, no active pair, no interaction.
2. **No ambient authority.** There is no global function lookup, no global book, and no process-wide registry visible inside a contract graph.
3. **Hashes identify, wires authorize.** A package hash is not a capability.
4. **Linearity is explicit.** A linear capability cannot be duplicated and is consumed by its send. A shared capability may be copied.
5. **Fuel is stop-reducing.** One interaction is one countable step on every path, including copy, erase, match, and serialization.
6. **Failed operations expose nothing.** A failed send commits neither a graph nor a receipt. A failed merge leaves the old root visible.
7. **Confluence or declared boundary.** Every rule extension is either locally confluent or marked as a boundary that voids agreement claims.
8. **Code is data.** Deploy, fork, test, merge, and migrate are graph transformations, not privileged in-place edits of running code.
9. **Partition before replication.** A node may reduce only the subspaces it is authorized to listen to. Replication is a property of a subspace, not of the whole cluster.
10. **Proof is necessary, not sufficient.** A green `PROOF.bend` does not prove Ed25519, the filesystem, the network, or the OS sandbox.

5. GOALS

1. Deploy a signed Bend 2 package as a private subnet of the live net.
2. Mint a shared invoke capability and a linear update capability.
3. Send a message graph over a capability and return a receipt.
4. Pass a capability to another instance by sending it as a graph value.
5. Prove isolation: if no capability path exists, no interaction can occur.

6. Fork a live instance into a shadow branch without stopping the old branch.
7. Test the shadow with replayed input and proofs.
8. Merge atomically, or abandon the fork without changing the live root.
9. Partition the net into subspaces and authorize listeners per subspace.
10. Replicate each subspace three ways; survive one node failure without downtime.
11. Measure reductions per second and deploy/send latency on one machine and across the simulated cluster.
12. Keep the full HVM5 term evaluator (`reference/hvm5-mini`) as the semantic oracle for the expression layer.

6. NON-GOALS FOR V1

- Byzantine consensus or proof-of-work.
- Monetary settlement or a native token.
- General-purpose native tenant code.
- Arbitrary tenant I/O.
- Confidentiality of source code from the runtime operator.
- Browser execution of the full native runtime.
- A production WASM ABI.
- Full Bend 2 to HVM5 compilation of every language feature.
- Claiming that a book index is a security boundary.

7. VOCABULARY

Active pair. Two agents whose principal ports are connected. Only active pairs reduce.

Agent. A node with one principal port and zero or more auxiliary ports.

Capability. A typed, unforgeable reference to a root. v1 has `shared` and `linear` capabilities.

Collapse. Choose one branch of a superposition and erase the other, or extract the branch into the visible root.

Fork. Duplicate the live instance graph into a shadow branch. The old branch keeps serving.

Fuel. A non-negative interaction budget. One rewrite consumes one unit.

Graft. Copy a package subnet into the live net at a chosen space and connect its export roots.

Homoiconic. Code, state, and patches are the same graph representation.

Instance. A grafted package plus its live state root, owner, version, and capability set.

Isolation. The absence of a capability path between two instances or subspaces.

Label. A correlation tag on `DUP` , `SUP` , and typed constructor agents. Same-label `DUP` / `SUP` annihilate; different labels do not.

Linear. A capability that may be used exactly once and cannot be copied.

Merge. Atomically make a tested shadow branch the visible root, then drain and erase the old branch.

Partition. The map from a space path to the subgraph owned by that space.

Space. A path in the subspace tree, such as `root/public/acme` .

Subspace. A space and all of its descendants.

Superposition. Two alternatives in one graph position, written `SUP(L,a,b)` .

System contract. The genesis instance that authorizes deployers and registrations.

8. EXECUTION STACK — HOST VERSUS OBJECT LANGUAGE

This distinction is load-bearing:

```
contract: Bend 2 smart-contract source
          -> Bend 2 frontend (check + lower)
          -> HVM5 inet package (Book of Terms)
          -> signed deploy / graft
          -> reducer

host:     HVM5 reducer source written in Bend 2
          -> BendRT compiler (official Bend 2)
          -> one native C program with BendRT's flat evaluator

object:   HVM5 `Term` values (the compiled inet)
          -> interpreted/reduced by `src/hvm5/eval.bend`
          -> the HVM5 interaction calculus
```

BendRT does **not** expose agents, ports, wires, or active pairs. Writing `Term` in Bend 2 does not make BendRT an inet machine. The HVM5 net is an object-level model interpreted by the host. A true inet-native release needs a graph kernel (agents, ports, wires, active pairs) written in Bend 2 or emitted to the host, with the term language as its frontend.

The current `eval.bend` is therefore:

- **homoiconic** at the object level: code, state, messages, and patches are all `Term` values;
- **Bend 2 on BendRT** at the host level;
- **term-reducing**, not yet a first-class graph reducer: `Dup` / `Sup` are term constructors and the four core interactions are implemented at that level.

The graph calculus in `01-calculus.md` is the semantic target for the term language. The graph kernel is a separate implementation slice, not an automatic consequence of using Bend 2.

9. RELATIONSHIP TO THE HVM5 REFERENCE

`hvm5/reference/hvm5-mini` is the vendored early-draft evaluator from the Bend repository. It defines one term syntax, parser, evaluator, printer, and interaction count. It is not the v1 semantics.

The v1 DSL is the new implementation in `src/hvm5/`. It does not import the early draft. The early draft remains as history and as a comparison point for term syntax and interaction counting.

G.4 · `bend-runtime/hvm5/version1/spec/01-calculus.md`

01 — HVM5 v1 calculus

Status: normative for HVM5 v1.
Reference: Lafont, *Interaction Combinators* (1997); Taelin, *Interaction Calculus*; `hvm5/reference/hvm5-mini`.

1. NETS

A net is a tuple

```
N = (A, W, R, S)
```

where:

- `A` is a finite set of agents;
- `W` is a set of wires, each joining two distinct ports;
- `R` is a partial map from root names to ports;
- `S` is the space assignment, a map from agents and roots to space paths.

An agent is

```
a = (id, kind, label, arity, space)
```

with one principal port `a.0` and auxiliary ports `a.1 ... a.arity`.

A port is a pair `(agent, index)`. Each port is incident to at most one wire. A root is a named port that is visible to the membrane.

An **active pair** is a wire joining `a.0` and `b.0`. A wire joining a principal port to an auxiliary port is passive.

2. AGENT KINDS IN V1

Kind	Arity	Meaning
`ERA`	0	Eraser. Releases the reachable graph.
`CON`	2	Binary constructor or pair. `label` names the constructor.
`CTOR`	n	Named constructor with n fields. `payload` is inert data.
`DUP`	2	Lazy duplicator with a label.
`SUP`	2	Superposition with a label.

`DUP` and `SUP` are dual. `CON` and `CTOR` are data. `ERA` is the only agent that can release a graph without producing a value.

`CTOR` with `arity = 0` is a leaf (a number, a string, a unit). Its payload is not inspected by the reducer.

3. INTERACTION RULES

Rules are local: they inspect only the two agents of an active pair and their ports. No rule searches the net.

3.1 ERASURE

```
ERA - ERA   =>  (empty)

ERA - X     =>  for each auxiliary port X.i:
                  create ERA' in X.space
                  connect ERA'.0 to the far end of X.i
                remove X
```

The first rule removes both agents. The second propagates erasure to every child. Each rewrite costs one fuel unit. The created erasers are reduced later by the same rules.

3.2 ANNIHILATION

```
CON(L) - CON(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
DUP(L) - DUP(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
SUP(L) - SUP(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
DUP(L) - SUP(L)   =>  connect a.1-b.1, a.2-b.2; remove a,b
```

Same-label `DUP` / `SUP` is the primitive fork/merge elimination: the two alternative branches are paired with the two uses.

Different `CON` labels are a **typed clash**, not an interaction. The pair is reported as an error and never silently reduces.

3.3 DUPLICATION OF DATA

A `DUP` facing a constructor duplicates the constructor's fields:

```
DUP(L) principal - CON(K) principal
with DUP aux d1,d2 and CON fields c1..cn
=>
create CON1(K), CON2(K)
connect d1 <-> CON1.principal
connect d2 <-> CON2.principal
for each field i:
  create DUPi(L)
  connect the old field wire to DUPi.principal
  connect DUPi.aux1 <-> CON1.aux_i
  connect DUPi.aux2 <-> CON2.aux_i
remove the original DUP and CON
```

The same schema applies to `CTOR` with `n` fields. Each field gets its own duplicator. This is the only rule that grows the net in the contract fragment.

3.4 DIFFERENT-LABEL DUP / SUP

The general interaction calculus commutes different labels. v1 does not use this rule on the contract path: the type checker requires equal labels at a fork/merge boundary. A future version may enable the full commutation rule and must then carry the confluence proof for it.

3.5 TYPED STUCKNESS

Any other active pair is a **stuck typed net**, not a silent erase. Examples:

- `CON(A) - CON(B)` with `A != B`;
- `CTOR` principal connected to a data `CON`;
- a `DUP` whose label is outside the typed boundary.

The reducer returns `Stuck{at, reason}`. This is an error value, not a crash.

4. REDUCTION RELATION

Define a one-step relation `N -> N'` by choosing an active pair and applying its rule. A reduction sequence is a finite or infinite chain of steps.

A net is **strongly normalizing under fuel** if every sequence of at most `f` steps either reaches a normal form or stops with `OutOfFuel`.

The reducer is:

```
reduce(net, root, fuel, listenerSpaces) -> Result
Result = Done{net, root, interactions}
       | OutOfFuel{net, interactions}
       | Stuck{net, at, reason, interactions}
       | Denied{space, reason}
```

`listenerSpaces` is the set of spaces the calling node is authorized to reduce. An active pair whose agents do not both belong to an allowed space is not reduced; it remains in the net. This is the partition rule.

5. CONFLUENCE

For the v1 typed fragment:

- erasure rules commute with every rule that does not touch the erased agents;
- annihilation rules are local and deterministic;
- data duplication is a local copy schema with fresh agents and labels;
- typed clashes are terminal for the pair and do not rewrite neighbours.

Claim (local diamond). If two active pairs are disjoint, their rewrites commute. If they overlap, they are the same pair. Therefore the fragment is locally confluent.

Boundary. Different-label `DUP` / `SUP` commutation, fusion, packed-number fast paths, and host effects are outside the v1 diamond claim. They may be added only with a rule-specific argument or a `NonConfluent{reason}` marker that removes the agreement claim from the affected subspace.

6. FUEL

Fuel is a natural number. Every rewrite consumes exactly one unit:

Path	Cost
erasure step	1
annihilation	1
data duplication	1
typed clash detection	1
serialization of a graph	0 (outside reduction)

Path	Cost
membrane verification	0 (outside reduction)

The fuel counter is part of the receipt. A send that exhausts fuel commits nothing. The membrane may reserve fuel for the reply root; the reservation is explicit in the send envelope.

7. LABELS

Labels are strings or 64-bit integers. They have two jobs:

1. **Correlation.** Same-label `DUP` / `SUP` annihilate; different labels do not.
2. **Typing.** A constructor label identifies a constructor in a package.

Labels are not capabilities and are not secret. A `DUP` label is not a host reference count. A `SUP` is not a list.

8. ROOTS AND CAPABILITIES

A root is a named port. The membrane hides the raw port and exposes a capability token. A capability has:

```
Cap = (root, space, mode, epoch)
mode = shared | linear
```

- A `shared` capability may be used many times.
- A `linear` capability is consumed by one send and cannot be duplicated.
- An epoch binds the capability to a runtime generation. A capability from an old epoch is dead even if its token is replayed.

The capability record lives only in the membrane. It is never a graph node and never serialized raw.

9. HOMOICONIC ENCODING

A graph is encoded canonically as:

```
Graph :=
  agents: sorted [id, kind, label, payload, arity, space]
  wires:  sorted [[agent, port], [agent, port]]
  roots:  sorted [name, agent, port, space]
```

Agent ids are local to the graph. The canonical encoding sorts by stable keys and rejects duplicate ports. The content hash is

```
hash(G) = SHA-256("hvm5.graph.v1\0" || canonical(G))
```

Domain separation prevents a graph hash from colliding with a deploy envelope or a contract state hash.

10. HOMOICONIC TERM SURFACE

v1 exposes the calculus as a Bend 2 DSL:

```
Term = Var | Lam | App | Dup | Sup | Era | Num | Add | Eq | Tag | Match | Ref
```

`Term` is a Bend 2 `Data` type. Code, state, messages, patches, and programs are `Term` values. The interpreter is also Bend 2 and runs on BendRT. A term elaborates to the graph model in sections 1–8; the graph is the semantic model, not a separate host runtime.

Important: BendRT is a flat C evaluator, not an interaction-net machine. This DSL is an object-level HVM5 interpreter running on BendRT. The current evaluator reduces terms; it does not yet maintain a first-class agent/port/wire store. A true inet-native implementation requires the graph kernel described in sections 1–8 to be implemented in Bend 2 (or emitted to the host) and used by the contract runtime. The term interpreter is the frontend/reference for that kernel, not a replacement for it.

The v1 evaluator supports APP-LAM, APP-SUP, DUP-SUP with the same label, ground-data duplication (`Dup` of a `Tag` , `Num` , or `Era`), and the typed `Num` / `Add` / `Eq` / `Tag` / `Match` extension. DUP-LAM and different-label DUP/SUP are explicit v1 boundaries: a term that needs them returns `Stuck` , never a silent approximation. The graph semantics remains the target for those rules.

Binder names in the source format are globally unique. This is a well-formedness restriction that makes substitution capture-free; a frontend may rename before deploy.

11. REFERENCE INTERPRETER

`hvm5/reference/hvm5-mini/main.bend` is an early-draft port of `hvm5.c` . Its `LAWS.bend` and `PROOF.bend` are useful evidence for parser cursor, printer packing, and interaction counting, but it is **not** the v1 semantics.

v1 is the new Bend 2 DSL in `src/hvm5/` . It does not import the early draft. The draft stays in the tree as history and as a comparison point for the term syntax and interaction count.

G.5 · `bend-runtime/hvm5/version1/spec/01a-hvm4-execution.md`

01a — HVM4 as the execution reference

Status: informative, with normative consequences for v1.
Source checked: `HigherOrderCO/HVM4` , `src/hvm.c` plus `docs/hvm/core.md` , `docs/hvm/memory.md` , and `docs/hvm/interactions/*.md` (2026-10-20).

1. WHAT HVM4 ACTUALLY IS

HVM4 is a single-file C runtime for the Interaction Calculus. It is **not** an explicit graph store with `Agent` / `Port` / `Wire` records. It represents a term as one 64-bit word:

```
TAG (8 bits) | EXT (24 bits) | VAL (32 bits)
```

and reduces terms through a book and a heap:

- **book**: immutable static terms for `@name` definitions;
- **heap**: mutable dynamic terms;
- **SUB cells**: a binder's body/expr slot is replaced by a substitution cell, and linked variables resolve through it;
- **linked variables**: `VAR` , `DP0` , `DP1` point at their binder cell;
- **quoted variables**: `BJV` , `BJ0` , `BJ1` use de Bruijn levels for collapse;
- **DUP nodes**: `DP0` / `DP1` share the duplicated expression slot;
- **ALO**: a static book term plus a bind list, expanded one layer at a time;
- **UNS**: unscoped binding support.

The interaction vocabulary includes:

```
APP-LAM, APP-SUP, DUP-SUP, DUP-LAM,
DDU-SUP, DSU-SUP, ALO-LAM, ALO-DUP,
MAT/SWI/USE, OP2/EQL/AND/OR,
ERA, SUP, DUP, CTR, NUM, REF, NAM/DRY
```

2. THE IMPORTANT CORRECTION

HVM4 is not a port/wire graph even though its semantics is the Interaction Calculus. It is a **term encoding of the calculus** with sharing, binders, labels, substitution cells, and a static book.

Therefore HVM5 v1 should be judged the same way:

> An HVM5 term machine that compiles to BendRT is a legitimate inet representation if it preserves the IC interactions, labels, sharing, and observable normal forms.

The explicit `Agent` / `Port` / `Wire` model in `01-calculus.md` remains the semantic model. It is not required to be the heap representation.

3. WHAT V1 ADOPTS FROM HVM4

HVM4 idea	v1 use
static book + dynamic heap	`Book` of `Term` definitions plus a `Heap` of dynamic cells
linked variables and SUB cells	binder cells for `Lam` and `Dup`; substitution without a global map
`DP0` / `DP1`	the two sides of a `Dup`
`DUP` / `SUP` labels	correlation and fork/merge semantics
APP-LAM / APP-SUP	application rules
DUP-SUP / DUP-LAM	duplication rules
ALO	a static package term expanded lazily at graft time
collapse	turn a net with `SUP` / `DUP` into a set of pure terms
`-s` interaction count	fuel and benchmark evidence

4. WHAT V1 DOES NOT ADOPT YET

- The exact 64-bit packing. Bend 2 has `U32` and `Nat`, but the reference machine uses a `Term` ADT so it is readable and checkable. A bit-packed representation is an optimization slice.
- The C runtime, GPU lanes, and `-C` collapse implementation.
- The full dynamic-label (`DDU/DSU`) syntax.
- Full ALO/UNS optimization. v1 can expand a book reference eagerly or one-layer-at-a-time; the observable result must match.
- HVM4's exact parser. v1 starts with a smaller Bend 2 subset and a canonical `Term` builder.

5. PIPELINE V1 SHOULD IMPLEMENT

```
Bend 2 source (subset)
  -> parse / check
  -> core IR
  -> lower to HVM5 `Term` (Lam, App, Dup, Sup, Ctr, Mat, Num, Op)
  -> insert DUP/ERA for non-linear binders
  -> Book of static terms
  -> heap instantiation (ALO-like)
  -> reduce (APP-LAM, APP-SUP, DUP-SUP, DUP-LAM, ...)
  -> quote / collapse
  -> state hash + receipt
```

The current `eval.bend` implements the middle of this pipeline (term reduction) but not the Bend 2 frontend, the static book/heap, or the full rule set.

6. WHY THIS GIVES SELF-MODIFICATION

Because the book is data:

- a definition is a `Term` value, not a compiled native function;
- a patch is a `Term` transformation;
- a fork is a `Dup` of a live term/state and a new book entry;
- a merge is a `Sup` collapse plus a visible-root change;
- a book reference can be redirected without rewriting the whole heap, as long as the old term stays reachable for in-flight uses.

That is the homoiconic property the runtime needs. It does not require the host to expose `Port` / `Wire` records.

7. GATES

1. `bend src/hvm5/main.bend` runs the DSL on BendRT.
2. `bend src/hvm5/main.bend -o main` produces a native binary that prints the same output.
3. A Bend 2 subset compiles to `Term` and reduces to the same value as the reference term.

4. APP-LAM, APP-SUP, DUP-SUP, and DUP-LAM tests pass.
5. A `Book` reference can be replaced atomically while an old instance continues to reduce.
6. The term machine has a fuel count and a canonical term digest.

Gates 1–2 are verified. Gates 3–6 are the next slices.

G.6 · `bend-runtime/hvm5/version1/spec/01b-contract-compiler.md`

01b — Bend 2 contract → HVM5 inet compiler

Status: normative for v1.
Input: a Bend 2 contract source file.
Output: an HVM5 inet package (a `Book` of `Term` definitions plus metadata).

1. BOUNDARY

The runtime does **not** parse Bend 2 contract source at send time.

```
author writes:      counter.bend
deploy:             check -> lower -> hash -> sign -> package
runtime receives:   compiled inet package
send:               graft + reduce the inet, never parse Bend source
```

Every deploy runs the compiler automatically. There is no user-facing `--emit=inet` choice and no “compile later” state: a deploy of Bend 2 source either produces an admitted inet package or is rejected. The source may be kept for audit, but the deployed artifact is the inet package.

This is the distinction between a Bend interpreter and an HVM5 runtime. The compiler is a trusted deploy component; the deployed artifact is inet data.

2. FRONTEND

The frontend reuses the official Bend 2 checker (`bend --check-only`) for syntax, kinds, affinity, and termination. It then lowers a typed core subset.

v1 subset:

```
type T is Data:
  Ctor{field: U, ...}

def f(x: A, y: B) -> R:
  match x:
    case Ctor{+f1, f2}:
      <expr>

expr ::= x | 0n | 1n+p | Ctor{expr, ...}
       | match x { case Ctor{...}: expr; ... }
       | f(expr, ...)
       | Nat.add(expr, expr) | Nat.sub(expr, expr)
       | Nat.is_le(expr, expr) | Nat.is_eq(expr, expr)
       | let x = expr; expr
```

No effects, clocks, randomness, foreign calls, or `@unsafe` on the transition path. A contract that needs them is rejected at admission.

3. LOWERING

Bend 2	HVM5 inet term
`x`	`Var "x"`
`0n`	`Num 0`
`Ctor{a,b}`	`Ctr "Ctor" [A,B]`
`match x { case Ctor{a}: body; _: d }`	`Mat x "Ctor" ["a"] body d`
`f(a,b)`	`App (App (Ref f) A) B`

Bend 2	HVM5 inet term
`Nat.add(a,b)`	`Op2 Add A B`
`Nat.sub(a,b)`	`Op2 Sub A B`
`Nat.is_le(a,b)`	`Op2 Le A B`
`let x = v; body`	`App (Lam x body) V`
`def f(x) = body`	`Book[f] = Lam x Body`
`init()`	`Book["init"] = Body`
`transition(s,m)`	`Book["transition"] = Lam s (Lam m Body)`

The compiler emits fresh names for match binders and dup labels. Binder names must be globally unique in the emitted term (the v1 reducer assumes this).

4. AFFINITY LOWERING

Bend 2 binders are affine. The compiler computes use counts:

```
0 uses  -> Era
1 use   -> Var
n > 1   -> Dup with n-1 nested dups
```

Example:

```
λx. (x, x)
=>
λx. ! d &L = x; Ctr "Pair" [d0, d1]
```

The DUP label is fresh per duplication site. This preserves the source semantics while keeping the inet term affine.

5. MATCH LOWERING

```
match s:
  case Counter{value}:
    match m:
      case Inc{by}:
        Counter{Nat.add(value, by)}
      case Reset{}:
        Counter{0n}
```

becomes:

```
Match(s, "Counter", ["value"],
  Match(m, "Inc", ["by"],
    Ctr("Counter", [Op2(Add, Var "value", Var "by")]),
    Match(m, "Reset", [],
      Ctr("Counter", [Num 0]),
      Stuck)),
  Stuck)
```

A missing case is `Stuck`, not a silent identity. A typed contract should be total; the compiler can insert `Stuck` only where the checker proves the case unreachable, or reject the contract.

6. PACKAGE

```
Package = {
  format: "hvm5.package.v1",
  name, version, space,
  sourceHash,
  book: [Def],
  init: Term,
  transition: Term,
  stateType, msgType,
  exports: [Invoke, Update],
  laws: [LawRef],
  effects: []
}
```

The package hash covers the canonical inet term, not the source string. Two sources that lower to the same inet have the same execution identity; the source hash is retained for audit.

7. SERIALIZATION

v1 uses a canonical prefix S-expression:

```
(lam s (lam m
  (match1 (var s) "Counter" "value"
    (match1 (var m) "Inc" "by"
      (tag1 "Counter" (add (var value) (var by)))
      (match0 (var m) "Reset"
        (tag1 "Counter" (num 0))
        (stuck)))
    (stuck))))
```

The parser in `src/hvm5/parse.bend` is small and total over the format. The runtime loads the package, not the Bend source.

8. SELF-MODIFICATION

The compiled package is inet data. A patch is a `Term` transformation over the book:

```
newBook = patch(oldBook, patchTerm)
newHash = hash(newBook)
```

Deploying the patch is a graft of `newBook`; merging is a visible-root change. The old book stays addressable for in-flight sends. No native code is recompiled and no running heap is edited in place.

9. GATES

1. `bend contracts/counter.bend --check-only` passes.
2. The compiler emits a canonical inet package and a stable hash.
3. The Bend 2 term parser/reducer loads the package and reduces `transition(init, Inc{by:2})` to `Counter{2}`.
4. The same source compiled twice yields the same package hash.
5. A patch over the book yields a new hash and preserves the old package.
6. The runtime rejects a package whose source hash or inet hash does not match.

Gate 1 passes. Gate 2 passes for `counter.bend`. Gate 3 passes for the current `counter.bend → build/counter.inet → run.bend` path. Gates 4–6 (stable hash, patch, signature binding) are still open.

G.7 · `bend-runtime/hvm5/version1/spec/01c-bend2-to-inet-support.md`

01c — Does Bend 2 officially compile to inets?

Status: evidence note. This is a support matrix, not a wish list.

1. SHORT ANSWER

In principle yes. Officially, not today.

The pure core of Bend 2 can be lowered to interaction nets or an HVM4-style interaction-calculus term representation:

```
λx.body        -> LAM
f(a)           -> APP
Ctor{a,b}      -> CTR
match x {...}  -> MAT
affine binder  -> ERA / DUP insertion
Dup/Sup        -> DUP / SUP with labels
```

But the official Bend 2 compiler does not have an inet backend, and the papers in `bendlang/bend` do not specify one.

2. WHAT THE REPOSITORY ACTUALLY CONTAINS

Artifact	What it is	Inet backend?
`bend2/comp.ts`	Bend 2 compiler	No; emits C / JS targets
`paper/BendTT.pdf`	Type theory: terms, datatypes, kinds, dead code, paradoxes	No
`paper/BendRT.pdf`	Runtime: compilation, tail/cut/fork, memory, machine	No
`guide/GUIDE.md`	Language guide	No inet backend
`demos/pure_hvm5_mini/`	A port of `hvm5.c` written in Bend 2	No; an early-draft evaluator
`HigherOrderCO/HVM4`	Separate Interaction Calculus runtime (`hvm.c`)	Yes, but not a Bend 2 backend
`VictorTaelin/Interaction-Calculus`	Reference IC implementation	Yes, but separate

The only HVM5 artifact in `bendlang/bend` is `demos/pure_hvm5_mini` . It is a term evaluator written in Bend 2, not a compiler from Bend 2 source to nets.

3. WHAT THE PAPERS DO SAY

`BendTT.pdf` is a type-theory paper. Its table of contents includes:

- programs and proofs;
- quantities;
- kinds;
- dead code;
- terms, datatypes, kinds;
- Curry's and Girard's paradoxes.

`BendRT.pdf` is a runtime paper. Its table of contents includes:

- the program and its contract;
- compilation: tail, cut, fork;
- the segment;
- memory: terms, ownership, arrays;
- the machine.

Neither paper specifies an inet encoding, a port/wire graph, or a Bend2-to-HVM lowering. The official runtime is BendRT's flat C evaluator.

4. WHY "NO PROBLEM" IS TOO STRONG

A compiler from full Bend 2 to inets has to settle:

1. **Affinity and sharing.** Insert `ERA` for unused binders and `DUP` for reused binders without changing semantics.
2. **Pattern matching.** Encode ADTs, match order, defaults, and nested patterns into `CTR` / `MAT` terms or nets.
3. **Labels.** Preserve `DUP` / `SUP` correlation and choose fresh labels.
4. **Numbers and operators.** Lower `Nat` , `U32` , `F32` , arithmetic, and comparisons to either constructor encodings or host ops with a confluence boundary.
5. **Effects and IO.** Keep `IO` , files, sockets, and foreign effects outside the pure net, or define an effect membrane.
6. **Arrays and maps.** Decide whether they are constructor trees, host primitives, or a separate linear-heap protocol.
7. **Totality and `@unsafe` .** Decide what a proof means after lowering, and how `@unsafe` is represented.

8. **Modules and imports.** Resolve names before lowering; no ambient lookup in the net.
9. **Performance.** Preserve sharing/optimality and avoid a slow naive substitution machine.
10. **Conformance.** Prove the lowered term observes the same values as BendRT for a generated corpus.

Our `counter.bend` compiler is a small first slice, not a solution to 1–10.

5. WHAT OUR V1 COMPILER SUPPORTS

`compiler/bootstrap/compile.ts` currently handles:

- one `type ... is Data` contract state and message set;
- `init`;
- `transition` with nested `match` on constructors;
- constructor values;
- `Nat` literals, `Nat.add`;
- affine binders from the source pattern.

It emits a postfix inet package consumed by `src/hvm5/parse.bend` and reduced by `src/hvm5/eval.bend`. It does **not** compile the full Bend 2 language.

6. THE CORRECT CLAIM

> Bend 2's pure core is a reasonable source language for an inet backend, and the language has the features needed (affinity, ADTs, matching, lambdas). The official Bend 2 compiler does not ship that backend, and the official papers do not define it. HVM5 v1 is building an experimental subset compiler and must prove conformance against BendRT before claiming general support.

That is the honest position. Do not claim "Bend 2 compiles to inets" as a language feature until the full conformance gate exists.

The sustainable route for arbitrary Bend 2 is the checked core IR boundary in `01e-core-ir.md`: the official Bend 2 frontend parses, checks, and emits `hvm5.core.v1`; the Bend 2 `bendc.bend` lowers that IR to inet. Do not grow a second layout parser/typechecker in `bendc.bend`.

G.8 · `bend-runtime/hvm5/version1/spec/01d-self-hosted-compiler.md`

01d — Self-hosted Bend 2 → inet compiler

Status: plan for the compiler bootstrap.
Goal: `bendc.bend` is a Bend 2 program, compiled by BendRT, that reads a Bend 2 contract and emits an HVM5 inet package.

Deploy rule: the deploy membrane calls `compile_for_deploy(source)` automatically. `--emit=inet` may exist as a developer diagnostic, but it is not the production deploy path and users do not choose it.

1. END STATE

```
bendc.bend                  Bend 2 compiler source
  -> BendRT native binary   `bend bendc.bend -o bendc`
  -> reads contract.bend
  -> checks / lowers / serializes
  -> contract.inet

hvm5-runtime.bend           Bend 2 inet reducer
  -> BendRT native binary
  -> reads contract.inet
  -> reduces the compiled inet
```

Both sides are Bend 2 compiled by BendRT. The contract source is never read by the runtime. This is the intended "every deploy is an inet deploy" story.

2. WHY A TWO-STAGE BOOTSTRAP

A full self-hosted compiler needs:

1. lexer and layout parser;

2. kind/type/affinity checker;
3. core IR;
4. lowering to inet terms;
5. label/fresh-name allocation;
6. canonical package writer;
7. diagnostics and source locations;
8. conformance tests against BendRT.

That is too much to land as one slice. The stages are:

STAGE A — BOOTSTRAP FRONTEND (CURRENT)

`compiler/bootstrap/compile.ts` parses the v1 contract subset and emits the postfix inet package. It is build-time tooling, not the runtime, and is marked **bootstrap only**.

Gate: `counter.bend -> counter.inet -> Counter{2}`.

STAGE B — BEND 2 LOWERING/SERIALIZER

`src/bendc.bend` starts as a Bend 2 program that consumes the **core IR** (not Bend 2 source) and emits the postfix inet package. This proves:

- a Bend 2 program, compiled by BendRT, can produce deployable inet data;
- the canonical package writer is self-hosted;
- the output is byte-identical to the bootstrap frontend.

Gate: core IR for `counter -> counter.inet -> Counter{2}`, with the same package hash as Stage A.

STAGE C — OFFICIAL CHECKED CORE IR

Do **not** move the whole Bend 2 layout/typechecker into `bendc.bend`. That would fork the language. Instead:

- the official Bend 2 frontend gains `--emit-core`;
- it emits the checked, first-order `hvm5.core.v1` module;
- `bendc.bend` lowers that module to inet.

See `01e-core-ir.md`. This is the path to arbitrary Bend 2 layout and typechecking without a second frontend.

Gate: `bend contract.bend --emit-core -> bendc.bend ->` inet package, for a corpus beyond the counter shape.

STAGE D — CONFORMANCE

Generate a corpus of Bend 2 programs. For each:

```
bendc.bend file.bend -> file.inet
hvm5-runtime.bend file.inet -> inet result
bend file.bend -> BendRT result
```

The observable values must match. Until this gate passes, Bend 2 → inet is an experimental backend, not a language guarantee.

3. CLI SHAPE

Production:

```
deploy(contract.bend) -> compile_for_deploy(source) -> admit/graft
```

Developer diagnostics only:

```
bend src/bendc.bend contracts/counter.bend --check-only
```

The production deploy path does not take an emit flag. The compiler is invoked by the membrane and its output is the package that admission hashes and signs.

4. WHY THE COMPILER CAN BE BEND 2

The compiler does not need to be fast in the same way the reducer does. It can use Bend 2 strings, lists, maps, and pattern matching, and it can be compiled to a native BendRT binary. It may use `@unsafe` for the parser loop, but that forfeits proof guarantees for the compiler itself; the output package still carries its own hash and proof metadata.

The hard constraint is **determinism**: the same source must produce the same inet package hash on every run and every host.

5. CURRENT STATUS

- Stage A is implemented (`compiler/bootstrap/compile.ts`).
- Stage B is implemented for the first-order contract fragment: `tools/emit-inet.ts` imports the official parser/checker and emits inet.
- Stage B2 is implemented as a Bend 2 source parser for the v1 contract shape (`src/bendc.bend`), including names/fields/parameters/numbers.
- Stage C (stable `hvm5.core.v1` file + generic Bend 2 lowering) is not implemented.
- Stage D (conformance corpus) is partial.

The bootstrap frontend remains in `compiler/bootstrap/` and must not be described as the final compiler.

G.9 · `bend-runtime/hvm5/version1/spec/01e-core-ir.md`

01e — The checked core IR boundary

Status: normative architecture for arbitrary Bend 2 → inet.
Purpose: stop the inet compiler from reimplementing Bend 2's layout parser, dependent typechecker, quantity checker, module resolver, law/proof checker, and effect checker.

1. THE RULE

```
Bend 2 source
  -> official Bend 2 frontend (parse + check)
  -> checked core IR
  -> HVM5 inet lowering (Bend 2 / BendRT)
  -> package
```

The official frontend is the language authority. The inet backend consumes a stable, layout-free, HOAS-free IR. `bendc.bend` never parses Bend 2 layout or re-implements the dependent type system.

This is how "arbitrary Bend 2 layout/typechecking" is supported without two compilers drifting apart.

2. WHY NOT A SECOND FRONTEND

Bend 2 has:

- layout-sensitive syntax;
- dependent types and one universe;
- quantities and kinds (`&0` , `&1` , `&2` , `<&>`);
- totality/termination checking and dead code;
- `@unsafe` , erased arguments, templates;
- laws, proofs, holes, rewrites, and the gate;
- modules/imports and namespace rules;
- effects and the host boundary;
- arrays, maps, strings, numbers, and shorthands.

The official frontend is thousands of lines of TypeScript and is still moving. A second parser in `bendc.bend` would be a second language implementation. It would reject valid programs, accept invalid ones, and drift on every release. That is not a compiler backend; it is a fork.

3. CORE IR V1

The IR is a canonical, checked, first-order representation. It is the only input `bendc.bend` accepts for the general path.

```
Module = {
  format: "hvm5.core.v1",
  sourceHash,
  compilerVersion,
  types: [TypeDef],
  defs: [Def]
}

TypeDef = {
  name,
  params: [Name],
  ctors: [Ctor]
}

Ctor = {
  name,
  fields: [{ name, type: TypeRef }]
}

Def = {
  name,
  params: [{ name, type: TypeRef }],
  result: TypeRef,
  body: Expr
}

Expr =
  | Var{name}
  | Ref{name}
  | Ctr{name, args: [Expr]}
  | App{fn, args: [Expr]}
  | Mat{value, cases: [{ ctor, binders: [Name], body: Expr }]}
  | Lit{value: Nat | Str | Bool}
  | Op{op: add | sub | mul | div | eq | lt | le, args: [Expr]}
  | Let{name, value, body}
  | Lam{param, body}
  | Era
  | Dup{label, left, right, value, body}
  | Sup{label, left, right}
```

Rules:

- The IR is **checked** before emission. The inet compiler may assume constructor arity and field order are valid.
- Names are globally unique in `bendc.bend`'s input. The frontend renames binders before emission.
- Quantities are already reflected: unused binders become `Era`, reused binders become `Dup`. The frontend or a lowering pass emits them; the inet compiler does not infer them.
- Effects are excluded from the transition path or marked as a separate boundary. The first IR version has no effect nodes.
- `Lam` is first-order: a binder name and a body, not a HOAS function.
- Source spans and names may be carried as inert metadata; they do not affect reduction.

4. EMITTER

The official compiler gains a build-time emitter:

```
bend contract.bend --emit-core build/contract.core
```

This is a frontend change in `bendlang/bend` (or a pinned plugin). It does not change the language; it exposes what the checker already knows.

The emitter must:

1. run the normal checker;
2. reject open laws, holes, and `@unsafe` transitions;
3. lower the checked HOAS terms to first-order `Expr`;
4. make quantities explicit;
5. canonicalize names and constructor order;
6. write `hvm5.core.v1`.

5. LOWERING IN BEND 2

`src/bendc.bend` accepts a `Module` and emits an inet package:

```
Module
  -> lower Expr to Term
  -> lower TypeDef/Ctor to Tag0/Tag1/Tag2 (v1 supports <= 2 fields)
  -> build init and transition roots
  -> package hash
```

This lowering is generic over the IR. It is not counter-specific. The current source-shape parser is a temporary bootstrap and must not be presented as the general frontend.

5A. CURRENT PROTOTYPE

`tools/emit-inet.ts` is the first implementation. It imports the vendored official Bend 2 parser/checker and traverses the checked `Book` / `HTerm` directly, emitting the postfix inet package. It is a backend prototype, not a second frontend: layout, dependent types, quantities, and laws are checked by the official compiler.

Verified:

```
bun tools/emit-inet.ts contracts/counter.bend build/counter-official.inet
bend src/hvm5/run.bend build/counter-official.inet
# done Counter{2} fuel=0

bun tools/emit-inet.ts contracts/tally.bend build/tally-official.inet
bend src/hvm5/run.bend build/tally-official.inet
# done Tally{7} fuel=0
```

The stable `hvm5.core.v1` file format and a Bend 2 lowering pass are still the target; the direct checked-AST emitter freezes the backend behavior first.

6. GATES

1. The official frontend checks `contract.bend`.
2. `--emit-core` writes a stable `hvm5.core.v1` module.
3. `bendc.bend` lowers the module to an inet package.
4. The reducer evaluates the package.
5. A corpus of counter, vault, list, and nested-match contracts produces the same observable values as BendRT.
6. Re-emitting the same source twice gives the same core hash and package hash.
7. The inet compiler never imports the Bend 2 layout parser.

Until gates 1–6 pass, “arbitrary Bend 2 → inet” is not claimed. The current `src/bendc.bend` covers the v1 counter shape only.

G.10 · `bend-runtime/hvm5/version1/spec/02-homoiconic-code.md`

02 — Homoiconic code, packages, and merges

Status: normative for HVM5 v1.

1. ONE REPRESENTATION

A package, a state, a message, a patch, and a snapshot are all `Term` values in the sense of `01-calculus.md`. Each term elaborates to a graph. The runtime does not have a privileged source-AST path for updates: the source compiler produces a `Term`, the registry stores the `Term`, the membrane grafts it, and the interpreter rewrites it.

```
Bend 2 source
  -> frontend (check, erase, lower)
  -> package graph (closed subnet + manifest)
  -> hash
  -> registry
  -> graft into the live net
```

The frontend may keep source spans and names as inert `CTOR` payloads. They do not participate in reduction.

2. PACKAGE

A package is:

```
Package = {
  format: "hvm5.package.v1",
  name: string,
  version: string,
  compiler: string,
  sourceHash: Hash,
  code: Program,
  exports: [Export],
  laws: [LawRef],
  effects: [EffectDecl],
  dependencies: [PackageHash]
}
```

`code` is a `Program` of `Term` definitions with named free roots. `exports` names the roots the membrane may expose. `laws` references proof artifacts; it is metadata, not authority. `effects` declares host effects; a package with an undeclared effect is rejected at admission.

The package hash covers the canonical encoding of the whole record, with `hash` itself excluded.

3. OPEN GRAPHS AND HOLES

An open graph has typed free roots:

```
Open = (Graph, Δ)
Δ = root name -> Type
```

A **hole** is a free root whose type is known but whose wire is not connected. Filling a hole is a link operation:

```
fill(Open, hole, Graph) = Open'
```

A fill is atomic from the membrane's point of view. If the types do not match, the fill is rejected and the old graph remains.

Holes are not strings looked up in a table. They are typed ports. This is why a fill cannot accidentally capture an ambient definition.

4. DEPLOY AS A GRAFT

`graft(package, space, suppliedCapabilities)`:

1. verify the package hash;
2. verify the deploy signature and sequence;
3. verify that every export has a declared type;
4. allocate a fresh instance id in `space`;
5. bind the package's `Program` term to the instance;
6. connect supplied roots to the package's free roots;
7. create the instance root and state root;
8. commit the graft atomically, or roll back on any failure.

The graft binds an immutable `Term`; the registry blob is immutable and may be shared by many instances. The interpreter receives the term and reduces it under fuel.

5. CODE AS DATA

Because code is a `Term`, a patch is a `Term` transformation. The v1 patch language is deliberately small:

```
Patch = {
  target: PackageHash,
  replace: [NodePattern -> NodeTemplate],
  add: [Graph],
  remove: [RootName],
  exports: [Export]
}
```

A patch is applied to a copy of the target `Term`. The result is a new package with a new hash. The old package remains addressable forever.

This is the homoiconic rule:

```
deploy(new) = graft(applyPatch(old, patch))
```

The runtime does not mutate the old graph. It constructs the new graph and atomically changes which root is visible.

6. FORK

A fork creates a shadow branch without stopping the live branch.

```
fork(instance, candidate):
  shadowCode  = copy(instance.code)
  shadowState = copy(instance.state)
  root        = SUP(L, instance.root, shadowRoot)
  fork        = Fork{id, label L, old: instance.root, new: shadowRoot, phase: Testing}
```

The two branches share immutable package blobs. Their mutable state graphs are separate copies. The label `L` is fresh and is recorded in the fork record.

During `Testing`:

- the live branch continues to accept sends;
- the shadow branch accepts test sends only;
- the membrane keeps the branch capability private;
- a test failure does not touch the live root.

7. TEST

A fork test is a deterministic replay:

```
test(fork, inputs, proofs):
  for each input in inputs:
    old = reduce(copy(oldBranch), input)
    new = reduce(copy(newBranch), input)
    record difference
  verify candidate laws
  verify fuel and graph-size bounds
  return Report{differences, proofs, fuel, accepted}
```

The test never merges automatically. A report with no differences and green laws is necessary but not sufficient; the owner decides.

8. MERGE

A merge has three phases.

8.1 PREPARE

- verify the fork is in `Testing`;
- verify the candidate hash;
- replay the final input log;
- verify the proof gate;
- reserve the new epoch.

8.2 COLLAPSE

```
collapse(SUP(L, old, new), choose = new):
  erase(old)
  root := new
  epoch := epoch + 1
  phase := Draining
```

The collapse is a single membrane transition. Either the whole transition happens or the old root remains visible.

8.3 DRAIN

Old capabilities are not invalidated immediately. They are marked `draining`; sends already in flight finish against the old root. New sends use the new epoch. When the drain deadline passes, the old root is erased and its capabilities are severed.

This is the Erlang current/old discipline expressed in graph terms.

9. ABANDON

Abandon erases the shadow root and its state graph, severs the private test capability, and leaves the live root untouched. The fork record is retained as evidence.

10. ATOMICITY

The following operations are atomic:

- graft (all nodes or none);
- merge collapse (old root or new root, never a mixture);
- epoch flip (old capabilities dead or new capabilities live);
- receipt append (both graph hash and receipt or neither).

The implementation uses a single commit point per operation. A crash before the commit leaves the old graph. A crash after the commit leaves the new graph. A crash during serialization is recovered from the last valid snapshot.

11. HASH DOMAIN SEPARATION

```
graphHash   = SHA-256("hvm5.graph.v1\0"   || canonicalGraph)
packageHash = SHA-256("hvm5.package.v1\0" || canonicalPackage)
stateHash   = SHA-256("hvm5.state.v1\0"   || canonicalValue)
envelopeHash= SHA-256("hvm5.envelope.v1\0"|| canonicalEnvelope)
receiptHash = SHA-256("hvm5.receipt.v1\0" || canonicalReceipt)
```

A package hash is never valid as a capability. A state hash is never valid as a signature. Domain separation is part of the type discipline, not a convention.

12. WHAT IS NOT HOMOICONIC IN V1

- Ed25519 keys and signatures are host values, not graph nodes.
- Membrane tokens are host values, not graph nodes.
- The OS process, sockets, and filesystem are outside the graph.
- The reducer's fuel counter is host state, not a graph node.

These are intentional boundaries. They are listed in `09-security.md` and are excluded from the inet confluence claim.

G.11 · bend-runtime/hvm5/version1/spec/03-types-isolation.md

03 — Types, capabilities, and isolation

Status: normative for HVM5 v1.
Method: a small type theory over graphs and capabilities, with runtime checks where the host boundary makes a static proof impossible.

1. WHY TYPES

The security claim is not "the code is trusted". It is:

> If there is no capability path from A to B, then no interaction between A and B can occur.

Types make that claim checkable before deployment and observable at runtime. The two central type constructors are:

```
Term[S]            a term whose elaborated agents live in space S
Cap[S, T; q]       a capability from space S to a root of type T, quantity q
```

`q` is `1` (linear) or `ω` (shared). `S` and `T` are space paths, not ambient names.

2. KINDS AND JUDGMENTS

```
κ ::= Value | Code | Net | Cap | Proof
q ::= 1 | ω
S,T ::= root / segment / ...
```

Judgments:

```
Γ ⊢ v : Value
Γ ⊢ t : Term[S]
Γ ⊢ p : Package[S]
Γ ⊢ c : Cap[S,T;q]
Γ ⊢ f : Fork[I]
Γ ⊢ r : Receipt[I]
```

`Γ` maps root names to types and capabilities. It contains no global module table and no ambient function namespace. `Term[S]` is the deployed code/data value; its elaboration is a `Net[S]` in the sense of `01-calculus.md`.

3. CAPABILITY RULES

3.1 MINT

```
deploy(owner, package, S) :
  Instance[S] ⊗ Cap[S, Invoke; ω] ⊗ Cap[S, Update; 1]
```

The update capability is linear. It cannot be copied by the type system, and the runtime rejects a DUP whose reachable graph contains a linear capability record.

3.2 SEND

```
send(c, m) :
  Cap[S,T;q] ⊗ Msg[S] -> Reply[T'] ⊗ Cap[S,T;q']
q' = 0  if q = 1
q' = ω  if q = ω
```

A linear send consumes the capability. A shared send preserves it. A reply wire is linear by default.

3.3 SHARE

```
share(c) : Cap[S,T;ω] -> Cap[S,T;ω]
```

Sharing a shared capability creates a new membrane record, not a new graph wire. Attenuation may remove rights, never add them.

3.4 PASS

A capability may be sent as a message only if the receiving type expects it:

```
pass(c, m) : Cap[S,T;q] ⊗ Msg[Cap[S,T;q]] -> Reply
```

The wire is marshalled as a membrane presence, not as a raw port. The receiver gets a new capability record bound to its own epoch.

4. SPACE-INDEXED TYPES

Each agent and root has a space. The typing rule for a wire is:

```
a : Term[S]    b : Term[T]
wire(a.0, b.0) : Cap[S,T;1]     if S != T
```

A cross-space wire is a **bridge**. It is only legal when both subspaces have authorized the bridge and the membrane has minted a bridge capability. A local wire (`S = T`) is internal and needs no external capability.

Consequences:

- Agents in disjoint subspaces cannot form an active pair.
- A bridge is explicit in the capability graph.
- Replication within a subspace does not create cross-space authority.

5. ISOLATION RELATION

Define the capability graph `G = (V, E)` where:

- `V` is the set of instances and spaces;
- an edge `A -> B` exists if A holds a capability to B.

Define:

```
Reach(A) = { B | A ->* B }
Isolated(A,B) = B ∉ Reach(A)
```

The runtime exposes `isolated(a,b) -> {isolated: bool, witness: Path | null}` . The witness is a concrete capability path. This makes the claim auditable rather than a boolean assertion.

THEOREM 1 — NO PATH, NO INTERACTION

If `Isolated(A,B)` , then no active pair can contain an agent from A and an agent from B.

Proof sketch. An active pair requires a wire. A cross-space wire is a bridge capability. If there is a wire from A to B, its bridge is an edge in the capability graph, so B is reachable from A. Contradiction.

THEOREM 2 — LINEAR NON-DUPLICATION

A `Cap[S,T;1]` cannot be a subgraph of a `DUP` duplication result.

Proof sketch. Capability records are membrane-only and are not graph nodes. The `pass` rule creates a fresh membrane record and transfers the linear record. The runtime gates every graft and message boundary: if a linear record would be copied, the operation is rejected before any rewrite is committed.

THEOREM 3 — ISOLATION IS MONOTONE UNDER ATTENUATION

If `Isolated(A,B)` and `A'` is obtained from `A` by removing capabilities, then `Isolated(A',B)` .

Proof sketch. Removing edges cannot create a path.

THEOREM 4 — MERGE ATOMICITY

A merge either leaves the old root visible or makes the new root visible, never a mixture.

Proof sketch. The visible root and epoch are one membrane record. The collapse is one compare-and-swap on that record. Branch erasure happens after the swap.

6. CONTRACT ABI TYPES

A contract exposes:

```
type State
type Msg
type Reply

init       : State
transition : State × Msg -> State × Reply
migrate    : OldState × Extra -> NewState
```

The checker must establish:

1. `init` is total and terminating;
2. `transition` is total and terminating;
3. `migrate` is total;

4. the result type is the declared `State`;
5. no effect, clock, random, or foreign call occurs on the transition path.

The first four are checked by the Bend frontend and by the v1 runtime's contract validator. The fifth is a manifest and effect-declaration check, not a theorem about C.

7. TYPE ERRORS

The following are errors, not stuck states:

```
TypeMismatch{expected, observed}
LinearEscapes{capability, site}
CrossSpaceUnapproved{a, b}
IsolationViolation{from, to, witness}
StaleEpoch{tokenEpoch, currentEpoch}
ForbiddenEffect{name, site}
```

Errors are values. A failed type check commits nothing.

8. STATIC AND DYNAMIC SPLIT

Property	Static	Runtime
constructor field types	Bend checker + contract validator	decode check
transition totality	Bend checker	fuel + step limit
linear capability	v1 type checker over the capability graph	membrane gate at graft/send
isolation	capability graph reachability	listener authorization
space ownership	authority map	node listener check
hash identity	canonical encoding	SHA-256 verify
signature	envelope schema	Ed25519 verify
fuel bound	declared budget	counter

The static side is a proof obligation. The runtime side is the enforcement. A green proof does not excuse a missing runtime check.

9. MINIMAL TRUSTED BASE

The trusted base is:

- SHA-256 and Ed25519 implementations;
- the canonical encoder;
- the membrane record store;
- the fuel counter;
- the OS sandbox.

The graph reducer, the contract evaluator, the registry, the system contract, and the cluster protocol are all untrusted code from the perspective of the security claim. They may be wrong; the trusted base must still reject a malformed commit.

G.12 · `bend-runtime/hvm5/version1/spec/04-subspaces.md`

04 — Partitioned subspaces

Status: normative for HVM5 v1.

1. SPACE IS A PATH

A space is a slash-separated path:

```
root
root/public
root/public/acme
root/public/acme/counter
root/private
root/private/vault
```

The tree is ordered by prefix. A subspace is a space and all descendants. A space is not a DNS name, a package name, or a capability. It is a coordinate in the graph partition.

2. PARTITION

Every deployed `Term` and root carries a space; its elaborated agents inherit that space. The partition function is:

```
part(N) = { S -> inducedSubnet(N, S) }
```

where `inducedSubnet` keeps agents whose space is `S` or a descendant of `S`, and keeps wires whose both endpoints are in that set. A wire with endpoints in different spaces is a **bridge** and belongs to neither induced subnet until a bridge capability is supplied.

Reduction is local:

```
reduce(N, S, fuel) = reduce(inducedSubnet(N,S), fuel)
```

A node may reduce only the spaces in its `listen` set. An active pair outside that set stays in the net and is not counted as fuel.

3. AUTHORITY

An authority row is:

```
Authority = {
  space: string,
  owner: PublicKey,
  listeners: [NodeId],
  replicas: 3,
  deploy: "owner" | "system",
  bridges: [Space]
}
```

Rules:

1. The system contract creates authority rows at genesis or by signed update.
2. Only listed listeners may reduce a space.
3. Only the owner or the system contract may deploy into a space.
4. A bridge from `S` to `T` must be present in both authority rows.
5. Removing a listener immediately stops new local reductions. In-flight reductions finish or are abandoned with fuel exhaustion.
6. The root space's authority is owned by the genesis system contract.

4. THE TEN-NODE TOPOLOGY

v1 uses ten light nodes:

Node	Listen spaces	Role
`n0`	`root`, `root/sys`	system contract replica, genesis
`n1`	`root`, `root/sys`	system contract replica
`n2`	`root`, `root/sys`	system contract replica
`n3`	`root/public`	public replica A

Node	Listen spaces	Role
n4	root/public	public replica B
n5	root/public	public replica C
n6	root/private	private replica A
n7	root/private	private replica B
n8	root/private	private replica C
n9	none	client / observer / gateway



Three replicas per reducing subspace. A quorum is two of three. One node may fail without stopping the subspace. Two failures stop new commits in that subspace but do not corrupt committed state.

5. REPLICATION

Each subspace has an append-only log:

```
LogEntry = {
  seq: nat,
  epoch: nat,
  op: Graft | Send | Merge | Authorize | Sever,
  hash: Hash,
  stateHash: Hash,
  fuel: nat
}
```

A leader is chosen deterministically as the lowest live node id in the subspace. A commit requires the leader plus one follower (quorum 2/3). The follower applies the same deterministic operation and returns its state hash. Mismatch is a `ReplicaDivergence` error and stops the commit.

A node that was down catches up by replaying the log from its last committed sequence. If the log is compacted, it fetches a signed snapshot at a sequence and replays the tail.

6. FAILURE

A node failure is a host-level event. v1 simulates it by marking a node `down` and rejecting its operations.

- Leader failure: the next live node by id becomes leader; the epoch increments.
- Follower failure: the leader continues with the remaining follower.
- Quorum loss: the subspace enters `ReadOnly`; no new commits, reads still return the last committed hash.
- Recovery: a returning node catches up and becomes a follower.

No failover changes the graph semantics. The same input log produces the same canonical state hash on every replica.

7. PRIVATE AND PUBLIC

The public/private distinction is a consequence of authority, not a flag in a contract:

- `root/public` has `n3,n4,n5` as listeners and may allow shared capabilities from public clients.
- `root/private` has `n6,n7,n8` as listeners and accepts only capabilities minted by the private owner or the system contract.
- No wire exists between the two induced subnets unless a bridge is explicitly authorized in both rows.

The isolation check is therefore:

```
isolated(publicInstance, privateInstance)
  = no bridge in both authority rows
  ∧ no capability path in the membrane
```

A public node cannot ”peek” at private state by asking the reducer: the private agents are not in its partition, and the private capability is not in its membrane.

8. BRIDGES

A bridge is a pair of mirror roots:

```
Bridge = {
  id,
  from: Space,
  to: Space,
  fromRoot: Root,
  toRoot: Root,
  mode: shared | linear,
  approvals: [Signature]
}
```

The graft creates one root in each subspace and a membrane record that maps them. A send across the bridge consumes the bridge capability according to its mode. The bridge is a capability edge in the isolation graph, so `isolated` reports it as a witness path.

Bridges are explicit, signed, and revocable by severing the membrane record. They are not implicit parent-child access. A child space does not automatically hold a capability to its parent.

9. SPACE TYPES

The type checker assigns:

```
agent : Net[S]        root : Net[S]
bridge : Cap[S,T;q]   listener : Node -> Space -> Bool
```

A contract package declares its `space`. A package cannot be grafted into a space it did not declare, and a listener cannot reduce a space it is not authorized for.

10. INVARIANTS

1. **Partition soundness.** Every agent belongs to exactly one space.
2. **Listener soundness.** A node reduces only agents in its listen set.
3. **Quorum soundness.** A commit has at least two matching replica hashes.
4. **Bridge soundness.** A cross-space wire requires a bridge record approved by both spaces.
5. **Failure safety.** A failed node never causes a different committed state.
6. **No ambient parent authority.** Parent and child spaces are isolated by default.

11. WHAT THE SIMULATION MUST SHOW

The executable scenario in `src/sim.bend` must demonstrate:

1. ten nodes start; each reports its listen set;
2. the system contract is replicated on `n0,n1,n2`;
3. a public instance is deployed and invoked by a client through `n3`;
4. a private instance is deployed and invoked only through `n6`;
5. a public caller cannot reach the private instance and the witness is empty;
6. one node in the public subspace is killed; the remaining two commit;
7. the killed node restarts and catches up to the same state hash;
8. a fork is created, tested, merged, and the old root drains;
9. the final cluster has one committed state hash per subspace.

G.13 · bend-runtime/hvm5/version1/spec/05-registry-system-contract.md

05 — Keys, registry, and the system contract

Status: normative for HVM5 v1.

1. IDENTITY

Identity is Ed25519.

```
KeyPair  = (privateKey, publicKey)
Address  = base58(SHA-256("hvm5.address.v1\0" || publicKey))
```

The private key never enters the runtime. The client signs an envelope and sends the signature, the public key, and the payload. The runtime verifies and discards the public key except for the address in the envelope.

v1 uses a deterministic key file for the simulation:

```
{"address":"...","publicKey":"...","privateKey":"..."}
```

The file is written with mode `0600` by the CLI. The simulation uses an in-memory keyring. No private key is written into a graph, a log entry, or a receipt.

2. ENVELOPES

All mutating operations use a domain-separated envelope:

```
Envelope = {
  domain: "hvm5.deploy.v1" | "hvm5.update.v1" | "hvm5.merge.v1",
  kind: "deploy" | "update" | "merge",
  owner: Address,
  publicKey: bytes,
  space: string,
  packageHash: Hash,
  previousHash: Hash | null,
  version: nat,
  sequence: nat,
  nonce: bytes,
  body: object,
  signature: bytes
}
```

The signed bytes are the canonical encoding of every field except `signature`, prefixed with the domain and a zero byte. The runtime verifies:

1. `domain` matches `kind`;
2. the public key hashes to `owner`;
3. `sequence` is strictly greater than the last accepted sequence for `owner`;
4. `previousHash` matches the current instance root for an update or merge;
5. the signature verifies over the canonical bytes;
6. the nonce was not seen before for this owner;
7. the owner is authorized for `space` by the system contract.

A failed check returns a typed `Reject`, never a partial commit.

3. REGISTRY, TWO LAYERS

3.1 CONTENT LAYER

```
PackageBlob = { hash, bytes, manifest }
```

The hash is `packageHash`. The blob is immutable. Anyone may fetch it and verify it locally. Names and versions are metadata, not identity.

3.2 NAME LAYER

```
NameRecord = {
  name: string,            // e.g. acme/counter
  packageHash: Hash,
  owner: Address,
  sequence: nat,
  validFrom: nat,
  validUntil: nat,
  signature: bytes
}
```

The name layer is a signed pointer. Higher sequence wins. A name is a discovery hint; the hot path uses the pinned hash and the live capability.

v1 stores name records in the system contract state. It does not implement a DHT or a blockchain. A name record is valid only while the system contract replica set can serve it.

4. SYSTEM CONTRACT

The genesis instance is deployed into `root/sys` with the state:

```
SystemState = {
  epoch: nat,
  admins: [Address],
  authority: [Authority],
  packages: [PackageBlobRef],
  names: [NameRecord],
  sequences: [Address -> nat],
  nonces: [bytes],
  instances: [InstanceRef],
  bridges: [Bridge],
  receipts: [Receipt]
}
```

The system contract exposes roots:

Root	Mode	Purpose
`register`	shared	publish a package blob and a signed name pointer
`authorize`	linear	create or replace an authority row
`deploy`	shared	verify an envelope and graft a package
`update`	linear	start an update or fork
`merge`	linear	commit or abandon a fork
`resolve`	shared	read a name or package hash
`inspect`	shared	read the current system state
`bridge`	linear	create a cross-space bridge

The `authorize`, `update`, `merge`, and `bridge` roots are linear. The `register`, `deploy`, `resolve`, and `inspect` roots are shared.

5. GENESIS

Genesis is a signed record with no predecessor:

```
Genesis = {
  format: "hvm5.genesis.v1",
  epoch: 0,
  admins: [rootAddress],
  authority: [
    {space:"root",       owner:rootAddress, listeners:[n0,n1,n2], replicas:3},
    {space:"root/sys",   owner:rootAddress, listeners:[n0,n1,n2], replicas:3},
    {space:"root/public",owner:rootAddress, listeners:[n3,n4,n5], replicas:3},
    {space:"root/private",owner:rootAddress,listeners:[n6,n7,n8], replicas:3}
  ],
  systemPackage: PackageHash
}
```

The root address is derived from the genesis key. The genesis key is generated once and stored by the operator; it is not derived from a contract.

6. ADMISSION

A deploy is admitted only if:

1. the package blob hash matches the envelope;
2. the package manifest declares the same space as the envelope;
3. the owner is authorized for the space;
4. the sequence is fresh;
5. the signature verifies;
6. the package's dependencies are present;
7. the package's laws are either green or explicitly waived by the owner;
8. the package's effects are declared and allowed in the space;
9. the resulting graft fits the space's graph-size limit;
10. the system contract has quorum.

Admission is a pure function over the system state plus the envelope. It can be replayed on every replica and must produce the same result.

7. NAMESPACES

v1 uses hierarchical names:

```
acme
acme/counter
acme/counter/v2
```

A name is owned by an address. The owner of `acme` may authorize a delegate for `acme/counter`. A delegate cannot create `acme/other` unless the `acme` owner grants it. The system contract stores `NameRecord`s and checks the parent authority on registration.

An unqualified name such as `counter` is public and first-come, first-served within the genesis policy. It can be renamed by its owner but never transferred without a signed transfer record.

8. REGISTRY AND CAPABILITY ENTANGLEMENT

The registry never mints a capability. Deployment does. The separation is:

```
registry : name -> packageHash          (discovery)
deploy   : packageHash + signature -> instance + capabilities (authority)
```

A client may resolve a name to a hash without holding any capability. A client must hold a capability to invoke. This prevents a name from becoming ambient authority.

9. SYSTEM CONTRACT LAWS

The v1 implementation checks these as executable predicates:

1. `sequence_strict`: an accepted envelope has a strictly higher sequence than the previous one for the same owner.
2. `hash_binds`: the grafted package hash equals the signed package hash.
3. `space_binds`: the grafted space equals the signed space.

4. `deployer_binds` : the signer is the owner and is authorized for the space.
5. `no_replay` : a nonce can be accepted at most once per owner.
6. `quorum` : a committed system operation has at least two replica hashes.
7. `bridge_two_sided` : a bridge is approved by both authority rows.

The Bend `LAWS.bend` model in `../bend/` states the pure versions of 1–5 and 7. Ed25519 and quorum are host tests, not Bend theorems.

10. RECOVERY

If the system contract's subspace loses quorum, new deploys stop. Existing instances in other subspaces continue to run if their own subspaces have quorum. This is intentional: the system contract is a control plane, not a data-plane dependency for every send.

A recovered system replica catches up from the signed log. A divergent replica is rejected and must resync.

G.14 · `bend-runtime/hvm5/version1/spec/06-lifecycle.md`

06 — Deploy, send, fork, test, merge

Status: normative for HVM5 v1.

1. INSTANCE STATES

```
Registered
   |
   v
Deployed --send--> Live --fork--> Forking --test--> Testing
                                  |                  |
                                  |                  +--abandon--> Live
                                  +--merge----------> Merging
                                                        |
                                                        v
                                                     Draining
                                                        |
                                                        v
                                                      Live
```

A failed operation returns to the previous state. The state is a membrane record, not a graph node, so a failed state transition cannot leave a half-written graph.

2. DEPLOY

```
deploy(source, envelope):
  1. compile the Bend 2 source to an inet package (automatic)
  2. reject if compilation fails or produces Stuck
  3. verify manifest, space, dependencies, laws, effects
  4. verify Ed25519 envelope, owner authority, sequence, nonce
  5. system contract quorum check
  6. graft the compiled inet package into the space
  7. evaluate init to a state term
  8. mint shared invoke capability and linear update capability
  9. append receipt and commit
```

The deployed artifact is the inet package. The source hash and compiler version are in the receipt for audit; the runtime never parses the source.

If any step fails, no instance and no capability exists. The package blob and the name record may already exist; those are content, not authority.

Deploy receipt:

```
Receipt {
  kind: "deploy",
  instance: InstanceId,
  packageHash: Hash,
  stateHash: Hash,
  space: Space,
  owner: Address,
  epoch: nat,
  sequence: nat,
  fuel: nat,
  status: "committed" | "rejected"
}
```

3. SEND

A send is a membrane operation plus a local reduction.

```
send(capability, messageGraph, fuel):
  1. resolve token -> (instance, root, mode, epoch)
  2. reject if stale epoch or not live
  3. verify message graph is closed and well typed
  4. check that the caller node listens to the instance space
  5. connect message root to instance invoke root
  6. reduce in the instance space under fuel
  7. evaluate the contract transition to a new state graph
  8. compute state hash and receipt
  9. replicate to subspace quorum
 10. commit
```

For a linear update capability, step 10 consumes the capability. For a shared invoke capability, the capability remains live.

Send receipt:

```
Receipt {
  kind: "send",
  sendId: Hash,
  instance: InstanceId,
  capability: CapId,
  oldState: Hash,
  newState: Hash,
  reply: Graph,
  fuelUsed: nat,
  status: "committed" | "rejected" | "out-of-fuel" | "stuck"
}
```

A rejected send returns the old state hash and commits nothing.

4. TWO UPDATE MODES

4.1 ALIAS FLIP (INSTANT)

For a package whose state layout is unchanged:

```
update(instance, newPackage, migrate?):
  verify package, signature, authority, sequence
  if migrate is absent:
    newState = oldState
  else:
    newState = migrate(oldState)
  atomically swap instance.code and instance.stateRoot
  epoch := epoch + 1
  keep old code/state until drain
```

This is the cheap path. It is still a graph operation: the visible root alias changes from the old package root to the new package root. The old graph remains addressable by the old epoch until it drains.

4.2 FORK-TEST-MERGE (SAFE)

For a package that changes behavior or state layout:

```
fork(instance, candidate):
  branch = copy(instance)
  fork = {old: instance.root, new: branch.root, label: fresh}
  instance.root = SUP(label, old, new)
  phase = Testing
```

The live branch keeps reducing. The shadow branch reduces only test sends.

`test(fork, inputs, proofs)` replays inputs against both branches and returns a report. It is deterministic and side-effect free.

`merge(fork, promote)`:

- if `promote = false`, erase the shadow branch and return to `Live`;
- if `promote = true`, collapse the SUP to the new branch, increment the epoch, enter `Draining`, and erase the old branch after the drain deadline.

5. CODE-AS-DATA TRANSFORM

A candidate package is produced by applying a patch to the old package `Term`:

```
patch = {
  target: oldHash,
  replace: [...],
  add: [...],
  remove: [...]
}

candidate = applyPatch(oldGraph, patch)
candidateHash = packageHash(candidate)
```

The patch is itself a `Term` and can be signed, stored, and tested. There is no "edit the running function" operation. The candidate is a new `Term`. The merge changes which term is visible.

6. TRANSACTION BOUNDARIES

Every mutating operation has one commit point:

Operation	Commit point
deploy	append deploy receipt + instance record
send	append send receipt + new state hash
fork	append fork record + SUP root
test	append test report (no live state change)
merge	compare-and-swap visible root + epoch
abandon	append abandon record + erase branch
authorize	append authority record
sever	mark capability dead

Before the commit point, a crash leaves the old state. After it, a crash leaves the new state. Recovery reads the log from the last committed sequence.

7. CONCURRENCY

- Each instance has a single-writer lock keyed by `(space, instance, epoch)`.
- A graft takes the subspace graft lock; sends do not.
- A merge takes the instance lock and the system-contract lock, in that order.
- No lock is held across a host effect (signature verification is done before the lock; log append is the only effect inside the commit section).
- A linear capability can be consumed by exactly one send. A second concurrent send gets `Consumed`.

This matches the v0 invariant: no lock across effects, one commit point per operation.

8. REPLAY

Every committed operation stores:

- the pre-state hash;
- the post-state hash;
- the canonical operation bytes;
- the fuel used;
- the epoch and sequence;
- the signature or capability identity.

Replaying the log from genesis must reproduce every state hash. A mismatch is a `ReplayDivergence` and stops the node. This is the primary determinism gate.

9. FAILURE CASES

Case	Result
bad signature	`Reject{Signature}` ; no state change
stale sequence	`Reject{Sequence}` ; no state change
wrong space	`Reject{Space}` ; no state change
no quorum	`ReadOnly` ; no new commit
fuel exhausted	`OutOfFuel` ; old state visible
typed clash	`Stuck` ; old state visible
fork test fails	fork stays `Testing` ; live branch untouched
merge crash before CAS	old root visible
merge crash after CAS	new root visible; old drains later
node death	quorum continues; dead node catches up

10. THE V1 STORY

The executable scenario is:

1. Genesis creates `root/sys` , `root/public` , `root/private` .
2. A client deploys `counter@1` to `root/public` with a signed envelope.
3. A client invokes it; `Inc` changes the state.
4. The owner forks `counter@2` with a code-as-data patch.
5. The shadow is tested with the recorded inputs.
6. The owner merges; the old root drains.
7. A private `vault` is deployed to `root/private` .
8. The public client tries to reach the vault; the isolation witness is empty.
9. `n4` is killed; `n3` and `n5` continue the public subspace.
10. `n4` restarts and catches up to the committed hash.

This is the acceptance scenario in `src/sim.bend` .

G.15 · `bend-runtime/hvm5/version1/spec/07-cluster.md`

07 — Ten-node cluster protocol

Status: normative for HVM5 v1.

1. NODE

A node is:

```
Node = {
  id: "n0".."n9",
  listen: Set<Space>,
  reducers: Map<Space, Reducer>,
  log: Map<Space, LogEntry[]>,
  state: Map<Space, Hash>,
  alive: bool,
  epoch: nat,
  peers: Map<Space, NodeId[]>
}
```

A node owns no global book. It owns a set of subspace partitions and the membrane records for the capabilities it is allowed to see.

2. TOPOLOGY

Space	Listeners	Replicas	Quorum
`root/sys`	n0,n1,n2	3	2
`root/public`	n3,n4,n5	3	2
`root/private`	n6,n7,n8	3	2
observer	n9	0	0

`n9` is the client gateway. It holds no reducer and no private key material for the contracts it invokes. It signs with the client key only when acting as the client.

3. MESSAGE TYPES

```
Graft{space, instance, packageHash, stateHash, authority}
Send{space, instance, capId, message, fuel, expectedEpoch}
Fork{space, instance, candidateHash, label}
Test{space, fork, inputs, proofs}
Merge{space, fork, promote, expectedEpoch}
Abandon{space, fork}
Authorize{space, row, sequence}
Bridge{from, to, mode, approvals}
Sever{capId}
Snapshot{space, seq, stateHash, blob}
```

Every mutating message is signed by its client or, for internal replication, by the leader. Internal messages carry the leader's epoch and sequence.

4. LEADER

For a subspace, the leader is the lowest live node id in the listener set:

```
leader(S) = min({n in listeners(S) | n.alive})
```

When the leader dies, the next live node becomes leader and increments the subspace epoch. Old leader messages with a lower epoch are rejected.

The leader is only an ordering point. It cannot change the semantics of an operation. All replicas execute the same pure function on the same pre-state.

5. COMMIT

```
propose(op):
  leader assigns seq
  followers apply op to pre-state
  collect state hashes
  if at least 2 hashes match:
    append entry with that hash
    return committed
  else:
    return ReplicaDivergence
```

The commit is a quorum of matching hashes, not a majority vote on different results. A divergent follower is marked `CatchingUp`, not accepted.

6. CATCH-UP

A returning node:

1. asks a peer for the latest committed sequence and hash;
2. if it has the same hash, it is caught up;
3. if it is behind, it replays entries after its last sequence;
4. if the gap is compacted, it fetches a signed snapshot and replays the tail;
5. it verifies every hash before applying.

A node that cannot verify the chain stays out of the listener set.

7. FAILURE AND DOWNTIME

A node is `down` in v1 when the simulator marks it down. A real implementation would use missed heartbeats. The semantics are the same.

- **One of three down:** the remaining two form quorum. Sends continue.
- **Two of three down:** no new commits. Reads return the last committed state.
- **Leader down:** a new leader is chosen; the epoch increments.
- **Return:** the node catches up before it can be leader.

There is no consensus across subspaces. Each subspace has its own log and quorum. A cross-space operation is a capability send, not a distributed transaction.

8. DETERMINISM

Every replica must produce the same canonical state hash for the same pre-state and operation. The sources of nondeterminism are excluded or injected:

- time: passed in the operation or not used;
- randomness: supplied by the envelope nonce, not by the reducer;
- hash map ordering: canonical encoding sorts keys;
- parallel scheduling: the v1 reducer is sequential; a future parallel scheduler must be observationally equivalent;
- host effects: outside the transition, recorded in the log.

9. SPACE PARTITIONS

A node's reducer rejects an operation whose space is not in `listen`. This is a hard check before any graph mutation. It is not a soft routing preference.

```
apply(node, op):
  if op.space not in node.listen:
    return Denied{Space}
  ...
```

A client can still hold a capability for a private space and route the send through an authorized node; the client itself does not reduce it. This is the difference between **holding** a capability and **listening** to a space.

10. CLUSTER INVARIANTS

1. No node reduces a space it does not listen to.
2. No committed state differs between replicas.
3. A committed operation is in the log before it is visible.
4. An old epoch cannot mutate a new root.
5. A bridge is the only way a cross-space active pair can exist.
6. A node failure does not erase committed state.
7. A recovered node catches up before serving writes.
8. The system contract's control-plane failure does not stop already-running data-plane subspaces with quorum.

11. TEN-NODE RUN

The runner starts all ten nodes, deploys genesis on `n0,n1,n2`, then runs the scenario. It prints, per step:

```
STEP <n> <kind> <space> <leader> <quorum> <state-hash-prefix>
```

The final report contains:

```
NODES 10
SUBSPACES 3 x 3 replicas
COMMITTED <count>
FAILOVER n4 -> n3
CATCHUP n4 <hash>
ISOLATED public private true
FORK <id> tested <n> merged <hash>
BENCH <interactions/s>
PASS
```

The CLI exits nonzero if any step fails or any replica hash diverges.

G.16 · bend-runtime/hvm5/version1/spec/08-api-cli.md

08 — CLI and wire API

Status: normative for HVM5 v1.

1. DESIGN

The CLI is a thin client over the same envelopes the membrane accepts. It never has a privileged mutation path. A command that changes state signs an envelope or presents a capability token.

```
client key  -> sign envelope -> membrane verify -> system contract -> log -> graph
capability  -> send message -> membrane resolve -> reducer -> receipt
```

2. CURRENT ENTRYPOINTS

These are implemented and runnable now:

```
bend hvm5/version1/src/deploy.bend hvm5/version1/contracts/counter.bend
bend hvm5/version1/src/sim.bend
bend hvm5/version1/tests/run.bend
bend hvm5/version1/bench/bench.bend
bend hvm5/version1/PROOF.bend
bend hvm5/version1/src/hvm5/run.bend build/counter.inet
```

`src/deploy.bend` reads a Bend 2 contract source and automatically compiles it to inet as part of deploy. There is no emit flag.

The commands below are the target CLI surface. They are not all implemented yet; the current simulation exposes them as pure functions and test entrypoints.

3. TARGET CLI

KEYGEN

```
bend hvm5/version1/src/cli.bend keygen --out .hvm5/keys/alice.json
```

Writes a key file with `0600` permissions and prints the address.

SIMULATE

```
bend hvm5/version1/src/cli.bend simulate
bend hvm5/version1/src/cli.bend simulate --json
```

Runs the ten-node acceptance scenario. Exit 0 only if every check passes.

BENCH

```
bend hvm5/version1/src/cli.bend bench
bend hvm5/version1/src/cli.bend bench --nodes 10 --sends 1000
```

Prints reduction throughput, deploy latency, send latency, replication cost, and failover time.

DEPLOY

```
bend hvm5/version1/src/cli.bend deploy \
  --key .hvm5/keys/alice.json \
  --package counter@1 \
  --space root/public \
  --source hvm5/version1/contracts/counter.bend
```

The CLI hashes the source and compiled package, signs the envelope, and asks the public subspace leader to admit it. Output:

```
{"instance":"inst_...","invoke":"cap_...","update":"cap_...","state":"sha256:...","receipt":"..."}
```

SEND

```
bend hvm5/version1/src/cli.bend send \
  --cap cap_... \
  --message 'Inc{by:2}' \
  --fuel 10000
```

Output:

```
{"status":"committed","oldState":"...","newState":"...","fuelUsed":12,"reply":"..."}
```

FORK

```
bend hvm5/version1/src/cli.bend fork \
  --key .hvm5/keys/alice.json \
  --instance inst_... \
  --candidate counter@2 \
  --patch patch.json
```

Output includes a fork id and a private test capability.

TEST

```
bend hvm5/version1/src/cli.bend test \
  --fork fork_... \
  --replay .hvm5/inst_.../inputs.jsonl \
  --proofs hvm5/version1/contracts/counter.PROOF
```

Output is a report with differences, proof status, fuel, and `accepted`.

MERGE

```
bend hvm5/version1/src/cli.bend merge \
  --key .hvm5/keys/alice.json \
  --fork fork_... \
  --promote
```

`--abandon` drops the fork and keeps the live root.

INSPECT

```
bend hvm5/version1/src/cli.bend inspect --instance inst_...
bend hvm5/version1/src/cli.bend inspect --space root/public
```

Returns the state hash, version, epoch, capability count, and log length. It never returns a raw wire or a private key.

ISOLATED

```
bend hvm5/version1/src/cli.bend isolated --from inst_public --to inst_private
```

Output:

```
{"isolated":true,"witness":null}
```

or a witness path if a capability chain exists.

VERIFY

```
bend hvm5/version1/src/cli.bend verify --package counter@1 --source contracts/counter.bend
```

Checks the package hash, manifest, and optional Bend source.

4. WIRE PROTOCOL

The native service and the Worker use the same JSON envelopes over a framed transport. v1's CLI runs in-process. The shape is:

```
request  = {op, id, body}
response = {id, status, body, receipt?}
```

`op` is one of:

```
register deploy send fork test merge abandon authorize bridge sever resolve inspect
```

All mutating requests carry either an `envelope` or a `capability`. The membrane never accepts a raw `wire` or `port` field.

5. CAPABILITY TOKEN FORMAT

```
cap_<base64url(random 32 bytes)>
```

The token is an opaque membrane handle. It is not a hash, not a port, and not a graph node. It is bound to:

```
{instance, space, root, mode, epoch, rights}
```

A token from an old epoch is dead. A token bound to a linear root is consumed on first send. A token cannot be forged by guessing because it is high-entropy and checked against the membrane table.

6. ERROR MODEL

```
Reject{reason}
Denied{space, node}
OutOfFuel{fuel}
Stuck{at, reason}
Consumed{capability}
StaleEpoch{token, current}
ReplicaDivergence{space, seq, hashes}
NotFound{kind, id}
```

The CLI prints `{"error":{"tag":"...","message":"..."}}` and exits nonzero. No error path writes a state change.

7. AGENT-FACING OUTPUT

Every command supports `--json`. The default human output is a short line per step. The JSON output is stable enough to be consumed by another agent:

```
{
  "ok": true,
  "command": "simulate",
  "steps": [{"step":1,"kind":"genesis","space":"root/sys","leader":"n0","hash":"..."}],
  "checks": {"ten_nodes":true,"three_replicas":true,"isolated":true,"failover":true},
  "hashes": {"sys":"...","public":"...","private":"..."}
}
```

8. VERSIONING

The CLI protocol is `hvm5.api.v1`. A field is never reused with a different meaning. A breaking change creates `hvm5.api.v2`. The package format and the graph format have independent version tags.

G.17 · bend-runtime/hvm5/version1/spec/09-security.md

09 — Security model

Status: normative for HVM5 v1.

1. ASSETS

1. private keys held by clients;
2. capability authority over a live instance;
3. contract state and its history;
4. package integrity;
5. the isolation of public and private subspaces;
6. the liveness of a subspace with quorum.

The runtime operator is trusted to run the code and to see the graphs it hosts. Source confidentiality from the operator is not a v1 goal.

2. BOUNDARIES

```
client / wallet        untrusted
  |
  v
CLI / HTTP membrane    untrusted transport
  |
  v
membrane               trusted for token checks and epochs
  |
  v
cluster / system        untrusted logic, verified by tests
  |
  v
graph reducer          untrusted logic, bounded by fuel
  |
  v
Ed25519 / SHA-256      trusted primitives
  |
  v
OS / container         trusted until independently audited
```

A trust boundary does not mean "the code is correct". It means a failure below the boundary can invalidate the security claim; a failure above it must be rejected by the boundary.

3. INVARIANTS

1. **No wire, no interaction.** A pair without a wire cannot reduce.
2. **No ambient lookup.** A contract cannot resolve a global function or book.
3. **Hashes identify, wires authorize.**
4. **Linear is not duplicable.** A linear capability is consumed once.
5. **Capabilities are not graph data.** Tokens never enter a graph.
6. **Spaces are enforced.** A node cannot reduce outside its listen set.

7. **Bridges are explicit.** Cross-space wires require two-sided authority.
8. **Failed operations expose nothing.** The old state remains.
9. **Epochs are monotone.** An old epoch cannot mutate a new root.
10. **A quorum matches hashes.** Divergence is not a majority vote.
11. **The proof gate is necessary, not sufficient.**
12. **The host is not proven by the graph.**

4. THREATS AND MITIGATIONS

4.1 FORGED CAPABILITY

An attacker guesses a token or constructs a `Wire` value.

- Tokens are 32 random bytes, held in the membrane.
- The graph has no `Wire` / `Port` constructors available to packages.
- A raw port is never serialized or accepted by the API.
- A linear token is consumed on use.

Residual risk: a compromised membrane table. The table is in the trusted base.

4.2 REPLAY OF A DEPLOY OR UPDATE

An attacker resends a signed envelope.

- `sequence` is strictly increasing per owner.
- `nonce` is stored per owner.
- `previousHash` binds an update to the current root.
- An old epoch token is dead.

4.3 WRONG-KEY DEPLOY

An attacker signs with a different key.

- The address is derived from the public key.
- The system contract checks the authority row for the space.
- The signature covers the space, package hash, and previous hash.

4.4 CROSS-SPACE READ

A public instance tries to invoke a private one.

- The private agents are not in the public node's partition.
- The private capability is not in the public membrane.
- No bridge exists in either authority row.
- `isolated` returns a witness if a path exists.

4.5 LINEAR CAPABILITY DUPLICATION

A contract tries to pass a linear capability into a duplicator.

- Capability records are membrane-only, not graph nodes.
- The graft/send gate checks the reachable capability set before a rewrite.
- The type checker rejects a DUP whose closure contains a linear record.
- `pass` transfers the record; it does not copy it.

4.6 FUEL EXHAUSTION

A malicious contract loops or grows the graph.

- Every rewrite consumes fuel.
- The graph-size limit is checked at admission and after each send.
- Over-budget sends commit nothing.

4.7 REPLICA DIVERGENCE

A node returns a different state hash.

- The leader collects hashes and requires a matching quorum.
- A divergent node is excluded and must resync.
- The log stores the canonical operation and pre/post hashes.

4.8 SYSTEM CONTRACT COMPROMISE

An attacker takes over the control plane.

- The genesis admin key is the root of authority.
- A compromised admin can authorize malicious deploys. This is a governance failure, not a graph-isolation failure.
- Data-plane subspaces with quorum continue to run.
- v1 does not implement key rotation or a multi-sig admin; this is a gap.

4.9 HOST EFFECTS

A contract tries to call the filesystem, network, clock, or random source.

- The contract transition path has no effect declarations.
- The manifest is checked at admission.
- Any future effect is a linear reply wire through the membrane.
- Effects are excluded from the confluence claim.

4.10 OPERATOR READS PRIVATE STATE

The operator can read process memory and disk.

- v1 does not claim confidentiality from the operator.
- A future version could use encryption or a TEE; that is a separate trust model.

5. TRUSTED BASE

The v1 trusted base is deliberately small:

- Ed25519 verify and SHA-256;
- canonical encoding and hash domain separation;
- membrane token table and linearity gate;
- fuel counter and graph-size limit;
- OS process isolation.

The graph reducer, contract evaluator, registry, system contract, cluster protocol, and CLI are untrusted code. They are covered by tests, not by a security proof.

6. SECURITY ACCEPTANCE

A v1 release candidate must pass:

1. forged token tests;
2. replay and stale-sequence tests;
3. wrong-key and wrong-space tests;
4. linear duplication rejection;
5. no-wire-no-interaction adversarial test;
6. public/private isolation test with a witness check;
7. fuel exhaustion test on every rewrite path;
8. replica divergence test;
9. node failure and catch-up test;
10. old-epoch capability rejection;
11. merge crash tests before and after the commit point;
12. a review of the trusted base.

A green `bend PROOF.bend` is not one of these tests. It is a separate gate.

7. KNOWN GAPS

- No key rotation or recovery.
- No multi-sig admin.
- No TLS in the in-process v1 CLI.
- No Durable Objects or disk WAL in the reference runner.
- No native wire opacity; the Bend 2 membrane uses opaque numeric tokens, not memory-safe handles.
- No Byzantine fault tolerance. A lying follower is detected by hash mismatch, but a lying leader can censor.
- No side-channel analysis of the host.

G.18 · `bend-runtime/hvm5/version1/spec/10-benchmarks.md`

10 — Benchmarks and performance claims

Status: normative for HVM5 v1.

1. WHAT IS MEASURED

Metric	Definition
`interactions`	Counted reducer steps, one per rewrite
`interactions/s`	Single-node reduction throughput
`graph-copy/s`	Canonical copy throughput for fork/shadow state
`deploy p50/p99`	Envelope verify + graft + init + commit
`send p50/p99`	Membrane resolve + transition + replicate + commit
`replication factor`	Leader + follower work divided by one-node work
`failover ms`	Time from kill to a new quorum commit
`catchup ms`	Time for a recovered node to reach the committed hash
`bytes/state`	Canonical state size
`fuel/send`	Fuel consumed by a representative message

2. REFERENCE WORKLOADS

W1 — REDUCTION MICROBENCHMARK

A chain of `CON` / `DUP` / `ERA` graphs with a known number of active pairs. The runner reports interactions and wall time. This isolates the kernel from the membrane.

W2 — DEPLOY

Deploy the `counter` package 100 times to an empty public subspace. Report deploy latency and graph-size distribution.

W3 — SEND

Invoke `Inc{by:1}` 10,000 times. Report send latency, fuel per send, and state growth. This is the hot path.

W4 — FORK/TEST/MERGE

Fork `counter@1` to `counter@2`, replay 1,000 recorded inputs, merge. Report:

- copy time;
- replay time;
- comparison time;
- merge time;

- drain time;
- zero failed live sends during the fork.

W5 — TEN-NODE CLUSTER

Deploy the public and private instances, run sends through the leaders, kill one public node, and continue. Report committed operations, failover time, catch-up time, and isolation checks.

W6 — ISOLATION

Build a capability graph with N instances and E edges. Time the reachability check and the witness extraction. This is a graph query, not a reduction.

3. METHODOLOGY

Rules:

1. Record CPU model, core count, RAM, OS, Bun version, and commit.
2. Warm up once; report median and p99 over at least 5 runs.
3. Keep the graph size and fuel budget fixed across compared runs.
4. Separate kernel time from membrane and serialization time.
5. A comparison with another runtime must use the same workload and the same output hash.
6. A performance claim without a checked hash is not a benchmark.
7. Do not quote a number from a different machine.

4. TARGETS

These are targets, not claims:

Target	v1 target
`interactions/s` on one core	at least 1,000,000
send p50 (in-process, no replication)	under 1 ms
send p50 (3 replicas)	under 5 ms
deploy p50	under 20 ms
failover	under 100 ms in the simulator
fork copy of a 1 MB state	under 10 ms
isolation query on 1,000 nodes	under 1 ms

A target is met only by a recorded run in `bench/results/`.

5. COMMANDS

```
bend hvm5/version1/bench/bench.bend
# The current runner is the W3-style repeated-send benchmark.
# W1/W5 variants are future runners.
```

The output is JSON so it can be committed as evidence:

```
{
  "workload":"w3",
  "commit":"...",
  "machine":{"cpu":"...","cores":8,"bun":"1.4.0"},
  "runs":5,
  "p50Ms":0.42,
  "p99Ms":0.91,
  "interactionsPerSecond":1234567,
  "hash":"sha256:..."
}
```

6. WHAT NOT TO CLAIM

- Do not claim ”as fast as C” from the Bend 2 reference interpreter. It is a reference, not the tuned native runtime.
- Do not claim ”HVM5 speed” from a term interpreter without the same workload.
- Do not claim linear scaling with node count. Replication adds work.
- Do not claim zero downtime from a single node run. Downtime is a cluster property and must be measured with a failure.
- Do not compare a cold start with a warm loop.

G.19 · `bend-runtime/hvm5/version1/spec/11-acceptance.md`

11 — Acceptance matrix

Status: normative for HVM5 v1.

1. DEFINITION OF DONE

The v1 goal is met when the following are all true in one run of `bend hvm5/version1/src/sim.bend`:

1. ten nodes start and report their listen sets;
2. three subspaces have three replicas each;
3. the system contract is replicated on `n0,n1,n2`;
4. a signed package deploys into `root/public`;
5. a signed package deploys into `root/private`;
6. a public send changes state and returns a receipt;
7. a private send changes state and returns a receipt;
8. the public instance cannot reach the private instance, and the witness is null;
9. one public node is killed and the remaining two commit;
10. the killed node restarts and catches up to the same hash;
11. a code-as-data patch produces a new package hash;
12. a fork is created and the live branch continues to serve;
13. the fork test replays inputs and reports no differences;
14. merge flips the visible root atomically and drains the old branch;
15. every subspace reports one committed state hash;
16. the proof gate is green.

2. EXECUTABLE TESTS

ID	Test	Gate
A1	`tests/eval.bend` APP-LAM, APP-SUP, DUP-SUP	`EVAL True`
A2	`tests/eval.bend` stuck/label boundary	`EVAL True`
A3	`src/sim.bend` replica count and listener topology	`REPLICAS-OK True`
A4	`src/deploy.bend` source → automatic inet compile	`DEPLOY Counter{0} PKG <u32> DIGEST <u32>`
A5	`tests/deploy_compile.bend` auto-compile + send	`DEPLOY-COMPILE done Counter{2}`

ID	Test	Gate
A6	`tests/run.bend` genesis, authority-shaped deploy	`PUB-OK True / PRIV-OK True`
A7	`tests/run.bend` ten nodes, quorum, failover, catch-up	`QUORUM-OK True / CATCHUP-OK True`
A8	`tests/run.bend` isolation, bridge, capability gate	`ISOLATED True / BRIDGE-OK True / CAP-OK True`
A9	`tests/run.bend` fork, test, merge	`PUB Counter{4}`
A10	`tests/run.bend` deterministic state digests	`PASS`
A11	`src/sim.bend` full scenario output	`PASS`
A12	`bend PROOF.bend` for the v1 model	`All terms check.`

3. ADVERSARIAL TESTS

ID	Attack	Expected
X1	forged capability token	`Reject`
X2	replay a deploy envelope	`Reject{Sequence}`
X3	wrong key	`Reject{Signature}`
X4	deploy into a space the owner does not own	`Reject{Authority}`
X5	send a linear capability twice	second `Consumed`
X6	public node reduces a private pair	`Denied{Space}`
X7	bridge without two approvals	`Reject{Bridge}`
X8	fuel exhaustion	`OutOfFuel` , old state
X9	divergent replica hash	`ReplicaDivergence`
X10	merge crash before CAS	old root
X11	merge crash after CAS	new root
X12	old epoch token	`StaleEpoch`

4. CURRENT STATUS (2026-10-20)

This section is deliberately not aspirational.

Implemented:

- `Term` , `Def` , `Program` as Bend 2 `Data` types;
- capture-free substitution under globally unique binder names;
- a CEK-style machine with an explicit frame stack and fuel mode;
- APP-LAM, APP-SUP, DUP-SUP (same label), ground-data `Dup` , `Num` / `Add` / `Eq` , `Tag0/1/2` , `Match0/1/2` , `Ref` ;
- a postfix inet package format and a Bend 2 parser;
- a Bend 2 contract subset compiler (`src/bendc.bend`) that parses constructor names, field binders, parameter names, and numeric literals, and is called automatically by deploy (`compile_for_deploy`), plus the bootstrap frontend;
- a checked Bend 2 contract (`contracts/counter.bend`);
- end-to-end: `counter.bend` → inet package → reducer → `Counter{2}` ;
- native BendRT build of `run.bend` executes the same package;
- a ten-node pure model: three sys, three public, three private replicas, plus a client; quorum, kill, heal, catch-up;
- deploy from an actual Bend 2 source file with automatic inet compilation;
- public and private deploy, send, isolation without a bridge;

- a negative bridge test (`BRIDGE-OK`);
- fork/test/merge over the inet package;
- a repeated-send benchmark (`BENCH sends=100 ...`);
- closed topology/isolation laws in `LAWS.bend` / `PROOF.bend` .

Not yet implemented:

- full DUP-LAM and different-label DUP/SUP commutation;
- a first-class agent/port/wire heap (HVM4-style terms are current);
- package hashing and native Ed25519 signed envelopes in the v1 pipeline;
- a true process/network cluster and WAL persistence;
- native crypto adapters, or a Worker ABI;
- the stable `hvm5.core.v1` file and full generic lowering; the current official-AST emitter and `bendc.bend` source parser cover the first-order contract fragment only.

The old TypeScript files are in `legacy/ts-oracle/` . They are **not** the runtime and are not imported by `src/` .

5. WHAT IS PARTIAL OR FUTURE

Area	Current	Required for v1
HVM5 term evaluator	Bend 2 CEK machine; APP-LAM/SUP, DUP-SUP, Add, Tag/Match pass	full tested IC fragment, DUP-LAM
Inet graph kernel	semantic spec; term representation	`src/hvm5/graph.bend + rules.bend`
Bend 2 → inet	official frontend + `tools/emit-inet.ts` first-order fragment	stable core IR + broader lowering
Contract model	`Counter` and `Tally` end-to-end	`Vault` and more ADTs
Cluster	ten-node pure model, replicas, quorum, catch-up	real processes/network
Native wire opacity	numeric capability model	native `law` handles
WASM Worker	v0 prototype	single-thread WASM ABI
Persistence	none	WAL, fsync, crash injection
Consensus	hash-quorum model working	process-level failover
Key management	deterministic numeric model	host Ed25519 adapter
Benchmarks	W3 repeated-send working	W1/W5 plus pinned hardware

The distinction matters. The term interpreter is a real Bend 2 program on BendRT. It is not yet an inet-native graph runtime, and the cluster is not yet implemented in Bend 2.

6. HONEST STATEMENT

A passing v1 run proves:

- the partition and capability model is executable;
- a signed deploy and a fork/merge lifecycle work under node failure;
- the public/private isolation claim has a runtime witness;
- the reducer is deterministic and fuel-bounded on the tested fragment.

It does **not** prove:

- that the Bend 2 reference interpreter is a high-performance HVM5;
- that the native membrane is memory-safe;
- that a C effect, the OS sandbox, or Ed25519 is correct;
- that a Bend 2 source program outside the contract fragment compiles;
- that the cluster survives a Byzantine leader.

Those are separate gates. They are listed in `plan/03-risks.md` .

G.20 · `bend-runtime/hvm5/version1/spec/12-traceability.md`

12 — Traceability: knowledge base to requirements

Status: informative map for v1. It records which ideas were extracted from `knowledge/` and where they landed in the spec and implementation.

1. SOURCE FAMILIES

Source	What it contributed
`knowledge/09-hvm5-cluster-runtime/`	homoiconic substrate, fork/merge, registry, OCap, layers, API, update flows, similar projects
`knowledge/07-smart-contracts-dynamic/`	Kindelia stateful functions, hub fork/upgrade, signed runs, metering, transition purity
`knowledge/04-runtime-bendrt-hvm5/`	HVM5-mini reference, BendRT memory/scheduler, fuel, laws gate
`knowledge/03-hvm-inet-theory/`	interaction nets, symmetric combinators, confluence, optimal reduction
`knowledge/08-roadmap-hvm5-spec/`	knowns vs gaps, reference index, spec TODO
`docs/tutorial/`	explicit Bend 2 syntax, laws/proofs, ownership, subspaces of the language
`hvm5/reference/hvm5-mini/`	executable term evaluator and proof gate

2. IDEA MAP

Extracted idea	Requirement	Spec
Code and running graph share one representation	Package/state/patch are graphs	02 §1
Hole is an affine free root	Typed holes and fills	02 §3
Fill is an atomic link	Graft atomicity	02 §4, 06 §6
`dup` = fork	Fork duplicates live graph	02 §6, 06 §4.2
`sup` = merge candidate	Superposition branch and collapse	02 §6–8
Labels control correlation	Same-label DUP/SUP annihilate	01 §3, §7
Never duplicate a duplicator	Type/label restriction on fork	01 §3.4, 03 §3
Old branch drains after cutover	Draining state	06 §4.2, §8
Immutable code hash	PackageHash and registry	02 §2, 05 §3
Mutable name pointer	NameRecord sequence	05 §3.2
Resolve once, pin hash	Name is discovery only	05 §3–4
Capability is a wire, not a name	Capability type and membrane	03 §3, 05 §8
Linear capability cannot be copied	Linear gate	03 §5, 09 §4.5
Eventual send and reply wire	Send and receipt	06 §3
Turn/vat isolation	Instance and subspace isolation	03 §4–5, 04
Telemetry/metering	Fuel and receipts	01 §6, 10
Proof gate before merge	Merge requires green laws	06 §4.2, 11
State migration is explicit	`migrate` type and alias flip	03 §6, 06 §4.1

Extracted idea	Requirement	Spec
Current/old dual generation	Drain and epoch	06 §4.2, §8
Registry content layer	Package blob store	05 §3.1
Registry name layer	Signed pointer	05 §3.2
System contract as genesis	Genesis and authority rows	05 §4–5
Signed envelopes	Ed25519, sequence, nonce	05 §2
Partitioned inet	Space tree and induced subnet	04 §1–2
Listener authorization	Node listen set	04 §3, 07 §9
Three replicas per subspace	Ten-node topology	04 §4, 07 §2
Node failure without downtime	Quorum and failover	04 §6, 07 §7
Isolation as no capability path	Reachability and witness	03 §5
Bridges are explicit	Two-sided approval	04 §8
Type-level shape constraints	Space-indexed types	03 §2, §4
Benchmark with pinned machine	Reproducible runs	10 §3

3. IDEAS DELIBERATELY NOT ADOPTED

Idea	Why
Global mutable namespace	Violates no-ambient-authority
Runtime `grant` / `revoke` primitives	Grant is wire-passing; revoke is sever
Facet type as a first-class runtime value	Facets are separate capabilities
Blockchain consensus in v1	Separate layer; v1 is one logical net
Native tenant code	Escapes the net and the fuel model
Copying a linear wire through DUP	Destroys revocation
Treating `PROOF.bend` as a security proof of C/OS	Different trust boundary
Book-index isolation as the security gate	Books are a prototype, not a boundary
Silent different-label DUP/SUP commutation	Requires a separate confluence argument

4. REFERENCE INDEX

The v0 research index is kept at `knowledge/08-roadmap-hvm5-spec/references-index.md`. The v0 spec and plan are kept at `spec/legacy/` and `plan/legacy/`. They remain useful for the detailed comparative research and the original slice wording, but this v1 set wins on conflict.

5. MAINTENANCE RULE

When an idea from `knowledge/` changes the runtime:

1. update the spec section;
2. update this table;
3. add or change an executable test;
4. record the decision in `plan/03-risks.md` if it changes a security claim.

A knowledge note is never normative by itself. A spec section plus a test is.

G.21 · `bend-runtime/hvm5/version1/spec/legacy/hvm5-inet-contract-runtime-spec-v0.md`

HVM5 Inet-Native Contract Runtime

Status: Draft 0 — normative architecture specification
Target: native Linux BendRT host + single-thread Cloudflare WASM Worker
Companion: `docs/hvm5-bendrt-contract-runtime-plan.md`

This document specifies what the runtime is. The companion document specifies the implementation order. If they conflict, this specification wins and the conflict is recorded before implementation.

1. SYSTEM DEFINITION

The primary runtime is one trusted Linux process containing an HVM5 interaction-net reducer. The same pure reducer is also compiled to a single-thread WASM module for a Cloudflare Worker. A package is an immutable graph subnet grafted into a live net. A capability is a wire/root port. A send connects a message graph to that port, forming an active pair; reduction performs the call.

```
signed package
  -> admission membrane
  -> private graph subnet
  -> exported wire roots
  -> send(message, reply-wire)
  -> local interaction reductions
  -> fuel-bounded result graph
  -> serialized subgraph + receipt
```

There is no runtime-level `invoke`, `grant`, `revoke`, or `facet` operation:

- invoke is a send on a wire;
- grant is sending a wire as a message;
- revoke is severing a linear wire or killing a user-level proxy;
- facets are separate wires;
- update is a signed replacement message on an owner wire.

This is smart-contract-like execution on one host, not a blockchain. Consensus, replication, finality, and monetary settlement are separate systems.

2. NORMATIVE LANGUAGE

MUST, **MUST NOT**, **REQUIRED**, **SHOULD**, **SHOULD NOT**, and **MAY** are normative. A conforming implementation satisfies every MUST in its scope.

3. GOALS AND NON-GOALS

3.1 GOALS

The native release MUST provide:

1. signed package grafting;
2. private subnets with explicit exported wires;
3. public and contract-to-contract sends;
4. owner-authenticated replacement and migration;
5. shared and linear wire semantics;
6. interaction fuel and graph/resource limits;
7. crash-safe graph/WAL persistence;
8. deterministic pure execution;
9. explicit host-effect membranes;
10. Bend laws, rule-table tests, differential tests, and adversarial tests;
11. native/WASM parity for the pure reducer and protocol ABI.

3.2 NON-GOALS

The first release MUST NOT promise blockchain consensus, source confidentiality from the operator, arbitrary native tenant code, arbitrary tenant IO, a browser page/DOM runtime, revocation of already-delegated shared wires, or memory safety beyond the trusted native/WASM runtime and host boundary.

4. ARCHITECTURE LAYERS

```
L6  wallet/client/external protocol
L5  deploy-and-send membrane
L4  package admission and graph linker
L3  HVM5 net kernel and reduction rules
L2  contract ABI and persistent process patterns
L1  BendRT host and effect runtime
L0  Linux process / Cloudflare Worker host, storage, sockets, cryptography
```

L0 — HOST

Provides credentials, memory, files, sockets, entropy, signature verification, durability, and resource limits. L0 is outside HVM5 and is trusted.

L1 — BENDRT/WASM HOST

Runs the Bend implementation of the reducer and membrane. Native mode is BendRT; Worker mode is an Emscripten/WASM module instantiated by TypeScript. It MUST NOT expose raw C pointers, file descriptors, allocator addresses, or arbitrary foreign functions to contract graphs.

The browser-page `-pthread` target from Bend PR #866 is not the Worker target: it requires `SharedArrayBuffer`, COOP/COEP, browser Workers, and `Window`. The Cloudflare target is single-thread and has no DOM, Window, WASI filesystem, TCP, or audio effects.

L2 — CONTRACT ABI

Defines persistent receive processes, message/reply shape, update continuations, state roots, and effect request/reply protocols.

L3 — HVM5 NET KERNEL

Defines agents, ports, wires, active pairs, `SUP`, `DUP`, erasure, annihilation, commutation, copying, collection, and interaction fuel.

L4 — PACKAGE LINKER

Validates closed graphs and typed holes, binds explicit imports, creates private subnets, and exports roots. It MUST reject ambient definition lookup.

L5 — MEMBRANE

Verifies signatures, maps external opaque handles to live roots, enforces quotas, serializes requests, prevents token leakage, and records receipts.

L6 — CLIENT

Holds private keys and external handles. Private keys MUST never be uploaded.

5. DOMAIN TYPES

5.1 IDENTITY AND CONTENT

```
Identity    = hash(public_key)
PackageHash = Hash(CanonicalPackageBytes)
GraphHash   = Hash(CanonicalGraphBytes)
DeployId    = Hash(SignedDeployEnvelope)
SendId      = Hash(SignedOrBearerSendEnvelope)
Sequence    = monotonic owner/runtime sequence
```

`PackageHash` identifies content; it MUST NOT grant authority. `Identity` identifies a signer; it MUST NOT grant graph access without a valid operation or held wire.

5.2 NET TYPES

```
Agent    = AgentKind × PrincipalPort × OrderedAuxPorts
Port     = (AgentId, PortIndex)                # internal only
Wire     = Edge(Port, Port) | Root(Port)
WireMode = Shared | Linear
Net      = Agents × Wires × ExternalRoots
Subnet   = OpenNet(Net, TypedHoles, ExportRoots)
Graph    = ClosedNet(Net)
```

`Port`, `Wire`, and `Root` are not ordinary user data. User code MUST NOT construct them from integers, inspect host identity, enumerate them, or serialize raw pointers.

5.3 HVM5 AGENTS

The exact rule table MUST be frozen from the selected HVM5 reference implementation before implementation. The semantic vocabulary is:

```
SUP = correlated superposition/branching agent
DUP = lazy duplication/distribution agent
ERA = erasure agent
CON = constructor/data agent, where applicable
LAM = abstraction agent, where applicable
APP = application agent, where applicable
```

`SUP` is not a normal list. `DUP` is not a host reference count. `ERA` releases reachable graph resources. Labels/correlation MUST survive `SUP` / `DUP` rules. Packed and fused fast paths MUST be observationally equivalent to the reference rule table.

5.4 CONTRACT TYPES

```
Export     = WireMode × WireRoot × MessageABI
Contract   = CodeSubnet × StateGraph × ExportSet × UpdateRoot
Message    = PayloadGraph × Optional<ReplyWire>
ReplyWire  = Linear wire by default
Effect     = RequestGraph × LinearReplyWire
Receipt    = SendId × OldGraphHash × NewGraphHash × Status × FuelUsed
```

State is a graph reachable from a contract root. There is no separate mutable contract-variable abstraction in the net kernel.

5.5 PORT, WIRE, TOKEN — THREE DISTINCT OBJECTS

These MUST NOT be merged in design, code, or documentation:

Object	Lives in	Forgeable?	Crosses process/disk?	Survives restart?
Port / agent	live net (kernel)	no (kernel-owned)	never	as graph, only if reachable from a persisted root
Wire (capability)	net + membrane	no (`law` handle + affinity)	never raw	only via epoch rebound
Token (external handle)	API / WAL / client	bearer, high-entropy	yes, TLS/Unix only	epoch-bound; old epoch is dead

Consequences: the `Capability = wire + linearity + target` struct is membrane-only state, never user data. Graphs are homoiconic; capabilities are extra-graph. Any design showing a user-constructible capability struct is rejected.

6. INET SEMANTICS

6.1 CONNECTIVITY IS AUTHORITY

Only a principal/principal pair is reducible. Therefore:

```
no wire -> no active pair -> no interaction -> no authority
```

This is true inside the trusted reducer. An external handle is a membrane capability resolving to a root; it is not a physical wire.

6.2 LOCAL RULES

Every reduction MUST be a local rewrite of one active pair and its auxiliary ports. The reducer MUST NOT search a global package table during reduction.

The rule table MUST cover `SUP` / `SUP`, `DUP` / `SUP`, `DUP` against every duplicable agent, erasure of nested graphs, packed constructors, and reference expansion if references remain in the implementation.

6.3 LINEARITY

Shared wires MAY be delegated and duplicated. They cannot be revoked after an untrusted holder receives a copy; use a proxy if revocation is required.

Linear wires MUST occur once, MUST NOT pass through `DUP`, MUST transfer or be consumed exactly once, and MUST leave no reachable copy after severance. The checker and runtime MUST enforce the same rule.

6.4 CONFLUENCE

The pure rule table MUST have a local-confluence/diamond argument. Native numbers, fusion, packed constructors, host effects, and graft operations need a confluence proof, a deterministic boundary, or an explicit order/receipt model. Tests alone do not prove confluence.

6.5 FUEL

One defined interaction is the native cost unit. Fuel MUST be decremented on every reduction path, including copying, fusion, erasure, matching, allocation, and packed fast paths. Graph-size, input/output, and host-effect limits are separate quotas. Overflow MUST be impossible. Out-of-fuel MUST publish nothing.

7. CONTRACT PROCESS MODEL

A public contract is persistent, not a one-shot function:

```
receive(message, reply-wire)
  -> connect message graph
  -> reduce
  -> emit response/effects
  -> reconstruct/continue receive process
```

Shared public wires accept many messages. Reply wires are linear by default. The first release serializes sends per state root and specifies reentrancy, self-send, recursive-send, fairness, queue bounds, and failure behavior.

Contracts call other contracts by sending on a received wire. There is no global package-name lookup or selector dispatch.

7.1 PERSISTENT-RECEIVER STATE MACHINE (NORMATIVE FOR DRAFT 0)

```
idle → connected → reducing → commit | abort
```

- `idle` : receive process waits on a shared invoke wire; no state held.
- `connected` : a send plugged a message + reply wire (active pair formed).
- `reducing` : bounded reduction runs; nested sends queue under the per-state-root savepoint policy, never mutating a second root concurrently.
- `commit` : output emitted on the reply wire, subgraph snapshotted, receipt appended; the receive continuation is reconstructed for the next message.
- `abort` : fuel exhausted, failure, or crash — reply wire reports failure, no subgraph committed, continuation restored to `idle` shape.

Reply wires are one-shot: reuse is rejected. A successful replacement MUST return a fresh linear update continuation wire.

7.2 CONCURRENCY LOCKS (NORMATIVE FOR DRAFT 0)

One mutex per state root for sends; one global graft lock for graft/replace. Grafting a disjoint root during a reduce is allowed; graft touching a reducing root waits. No lock may be held across a host-effect boundary.

8. GRAFT, DEPLOY, AND REPLACEMENT

8.1 GRAFT

```
Open subnet + typed holes + exports
  -> signature/admission checks
  -> connect supplied wires to holes atomically
  -> graft into live net
  -> return exported roots
```

The current demo's private books are an interim implementation technique, not the final wire-native security proof.

8.2 REPLACEMENT

An owner replacement MUST:

1. verify identity, sequence, old package hash, and signature;
2. validate the replacement subnet;
3. run migration within fuel limits;
4. switch the contract root atomically;
5. return a fresh update continuation wire;
6. retain the old graph if policy requires it.

Failure leaves the old root live.

9. EXTERNAL PROTOCOL

```
DEPLOY(package, manifest, owner_signature, fuel_limit, fuel_price, sequence)
  -> DeployReceipt { deploy_id, package_hash, exported_handles }

SEND(handle, message, optional_reply_handle, fuel_limit)
  -> SendReceipt { send_id, graph_hash, fuel_used, status }

LISTEN(handle, depth)
  -> serialized messages or exposed state
```

There are no protocol-level `INVOKE`, `GRANT`, `REVOKE`, `SNAPSHOT`, or `ROLLBACK` calls. They are graph/message patterns or host recovery procedures.

External handles MUST be high-entropy bearer or connection-bound capabilities, TLS-protected or Unix-socket-only, epoch-bound after restart, and absent from logs, URLs, and error strings.

10. STATE MACHINES

GRAFT

```
Received -> Authenticated -> Canonicalized -> Parsed
  -> RuleChecked -> ResourceAdmitted -> Grafted -> Committed
```

Any failure before `Committed` leaves no reachable partial subnet.

SEND

```
Received -> HandleResolved -> WireAuthorized -> Queued -> Reducing
  -> Committed | OutOfFuel | Rejected | Aborted
```

Only `Committed` publishes output, state, or effect records.

11. PERSISTENCE AND ENCODING

Canonical graph encoding MUST preserve agent kind, port order, sharing, cycles allowed by the calculus, `SUP` labels, `DUP` state, binders/cells, package-local references, and export roots. It MUST contain no raw host addresses.

External handles and live sockets are restored through an epoch-bound membrane table. WAL records MUST define `fsync`, atomic replacement, locking, and crash recovery semantics.

12. SECURITY BOUNDARY

PROTECTED BY THE NET/MEMBRANE MODEL

- unconnected interactions;
- forged integer/name/hash capabilities;
- duplicated linear wires;
- unauthorized replacement;
- replayed signed operations;
- over-budget reduction and graph growth;
- cross-subnet lookup.

NOT PROTECTED BY INET SEMANTICS ALONE

- BendRT/HVM5 native bugs;
- membrane or cryptographic bugs;
- compromised Linux operator/root;
- stolen bearer handles;
- incorrect effect behavior;
- unproved new reduction rules.

The trusted computing base is Linux isolation, BendRT, the HVM5 rule table, linearity checker, serializer, membrane, signature library, persistence effects, and effect adapters.

13. PROOF AND ACCEPTANCE OBLIGATIONS

PURE LAWS/FORMAL OBLIGATIONS

- parser and canonical-encoding round trips;
- graft-hole and package-closure correctness;
- `SUP` / `DUP` / `ERA` equations;
- linear-wire non-duplication;
- severance leaves no reachable linear copy;
- fuel monotonicity and overflow exclusion;
- send/replacement atomicity;
- identity/sequence/old-hash checks.

DIFFERENTIAL/ADVERSARIAL OBLIGATIONS

- compare with the HVM5 reference implementation;
- reduce identical pure graphs under varied schedules;
- fuzz all rule pairs and graph encodings;
- attempt forged ports, cross-subnet references, wire duplication, and use-after-sever;
- exhaust fuel in copier/fusion/erase paths;
- crash at every WAL boundary;
- test replay, reentrancy, reply reuse, stale handles, and key rotation.

`PROOF.bend` passing cannot prove native memory safety, cryptography, disk durability, or Linux security.

14. WASM PACKAGING AND CLOUDFLARE BOUNDARY

The build artifact for the Worker is a `.wasm` module, not the generated JavaScript kernel mirror.

```
net/api.bend + HVM5 reducer
  -> Bend C output
  -> Emscripten emcc -std=gnu11 -O3 -mtail-call
  -> hvm5.wasm
  -> Cloudflare Worker WebAssembly.instantiate()
```

Nix SHOULD provide the pinned Emscripten toolchain and Bend compiler source; Alchemy SHOULD bundle the resulting `.wasm`. The Worker TypeScript layer MAY implement HTTP, token/epoch membranes, and host storage, but MUST NOT reimplement contract semantics.

The `.wasm` ABI MUST expose a bounded pure reducer/graft/send surface and MUST not require browser `Window`, `SharedArrayBuffer`, pthread workers, X11, ALSA, TCP, WASI filesystem, or Emscripten page glue. Worker persistence belongs in a Durable Object or explicitly selected Cloudflare storage; an in-isolate Map is only a local smoke test.

The page target from Bend PR #866 / `metal-bending` is a separate conformance target. It uses `-pthread`, one browser Worker per core, COOP/COEP, and canvas; those assumptions MUST NOT leak into the Cloudflare module.

15. CURRENT IMPLEMENTATION GAP

`demos/pure_hvm5_mini` is a sequential Bend implementation of an HVM5-like evaluator. It has templates, terms, cells, continuations, duplication-related machinery, and a hard-coded program, but it does not yet have:

- first-class external wires;
- linear capability typing;
- true live-net grafting;
- persistent receive processes;
- canonical graph persistence;
- signature/membrane/WAL service;
- formal confluence proof for all optimized rules.

The first implementation milestone is the HVM5 net kernel, its exact `SUP` / `DUP` / `ERA` rule table, and linear-wire model—not HTTP.

16. DRAFT-0 BINDING DECISIONS

These decisions close the ambiguities found during adversarial review.

1. **Reference mode is not the security mode.** Private per-package books are an executable prototype only. The native-release security gate requires a first-class wire/graft membrane; a book-index isolation test cannot satisfy it.
2. **The first signature scheme is Ed25519.** The signed bytes use a canonical encoding and domain-separated contexts for `DEPLOY` and `REPLACE`.
3. **The first replay guard is a per-owner sequence.** There is no block height or wall-clock validity in the single-node runtime. A sequence is monotonic per owner and bound to the runtime ID and instance ID.
4. **Fuel price is accounting-only in Draft 0.** The runtime records `fuel_used`; it does not pretend to settle money until an account/escrow/ settlement layer is specified.
5. **Public entry wires are shared.** Owner update wires and reply wires are linear. A successful replacement returns a fresh update continuation wire.
6. **Sends are serialized per state root in the first release.** Reentrant and recursive sends have explicit queue/savepoint semantics; arbitrary parallel state mutation is deferred.
7. **Pure net reduction excludes host effects.** Effects are suspended request graphs sent through effect wires. The receipt records effect requests; an effect adapter defines retry and commit semantics separately.
8. **No package is admitted until its wire rules are checked.** A package may not use `@unsafe`, arbitrary foreigns, global book lookup, or a `DUP` path that contains a `Linear` wire.
9. **The OS boundary is primary defense in depth until the reducer is audited.** A native BendRT memory-safety bug is a process-wide compromise even when the net model is correct.
10. **Replaced versions stay; invoke wires do not follow.** Draft 0 policy: an existing public invoke wire remains attached to the old version. Callers move via an explicit forwarder package. No implicit forwarding root.
11. **Failed sends expose no subgraph.** An in-memory leftover from an aborted reduction is an implementation detail, never an API value; only committed snapshots and structured errors cross the seam.
12. **First-release inet criterion.** A sequential reducer satisfies "inet-native" iff: no global book lookup exists, failed rewrites are atomic, ports are not array indexes, and the same rule table drives the later parallel reducer. Otherwise "inet-native" MUST NOT be claimed.

17. RELEASE BLOCKERS

The implementation MUST stop before external API work if any blocker remains:

- the exact HVM5 rule table is not versioned and tested;
- `Wire`, `SharedWire`, and `LinearWire` have no checker/runtime semantics;
- graft/ungraft and partial-failure behavior are unspecified;
- persistent receive continuation is not implemented;
- canonical graph encoding cannot preserve `SUP` / `DUP` sharing;
- fuel does not cover copier, fusion, erase, match, allocation, and serialization;
- signature bytes, sequence rules, and key rotation are undefined;
- WAL durability and crash recovery are untested;
- confluence is being claimed for effects or unproved native fast paths;
- external handles lack a TLS/Unix-socket boundary and restart epoch.

G.22 · `bend-runtime/hvm5/version1/spec/README.md`

HVM5 v1 specification

The normative set starts here. Read in order for a first pass, then use it as a lookup.

#	Document	Question it answers
00	System	What is being built, and what is out of scope?
01	Calculus	What is a net, an agent, an active pair, and a rewrite?
01a	HVM4 execution	How does HVM4 represent and reduce the calculus?
01b	Contract compiler	How does a Bend 2 contract become an inet package?
01c	Bend2-to-inet support	Does the language officially compile to inets? What is evidence vs. goal?
01d	Self-hosted compiler	How does `bendc.bend` itself compile to BendRT and emit inet?
01e	Checked core IR	How is arbitrary Bend 2 passed to the inet backend without a second frontend?
02	Homoiconic code	How are packages, holes, forks, and merges graphs?
03	Types and isolation	How do we know two contracts cannot communicate?
04	Subspaces	How is one net partitioned, authorized, and replicated?
05	Registry and system contract	How are keys, names, packages, and admission handled?
06	Lifecycle	What exactly happens on deploy, send, fork, test, merge?
07	Cluster	How do ten nodes commit, fail, and catch up?
08	API and CLI	What can a client call?
09	Security	What is trusted, what is attacked, what is claimed?
10	Benchmarks	How is speed measured honestly?
11	Acceptance	What must one run prove?
12	Traceability	Which knowledge-base ideas became which requirements?

Normative keywords. MUST, MUST NOT, REQUIRED, SHALL, SHOULD, SHOULD NOT, MAY follow RFC 2119.

Conflict rule. If this spec and a plan, map, legacy document, or test disagree, the spec wins and the conflict is recorded in `plan/03-risks.md` before code changes.

Version rule. A breaking semantic change creates a new `hvm5/versionN` directory. A clarification may update v1 in place, with a changelog entry.

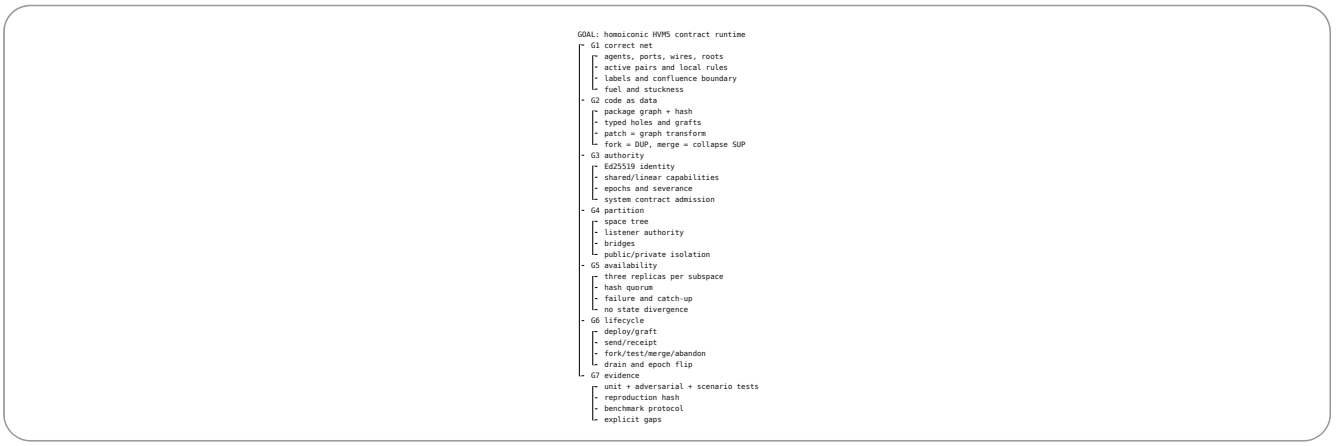
G.23 · `bend-runtime/hvm5/version1/maps/00-goal.md`

Goal map

THE ONE-SENTENCE GOAL

A signed Bend 2 program can be deployed into one live, partitioned interaction net, invoked by capability, forked into a shadow branch, tested, and merged without stopping the running branch.

GOAL TREE



OBSERVABLES

1. `bend hvm5/version1/src/sim.bend` prints `PASS`.
2. Ten nodes start; three subspaces have three replicas.
3. One public node dies; the other two commit.
4. The dead node restarts and reaches the same hash.
5. A signed deploy creates an instance and two capabilities.
6. A counter send changes state and returns a receipt.
7. A public instance cannot reach a private one; the witness is null.
8. A fork is tested while the live branch serves.
9. Merge flips the root atomically; abandon leaves it untouched.
10. The proof gate is green.

NON-GOALS

- Blockchain consensus, money, finality.
- Native tenant code.
- Source confidentiality from the operator.
- Full Bend 2 compilation in v1.
- Native HVM5 speed in the Bend 2 reference interpreter.
- Browser runtime.

FAILURE OF THE GOAL

The goal is not met if any of these is true:

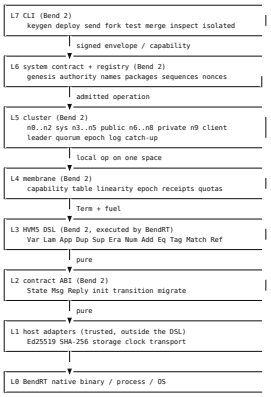
- isolation is asserted but no adversarial test exists;
- a node reduces outside its listen set;
- a fork blocks the live branch;
- a merge can leave a mixed state;
- a linear capability can be duplicated;

- a benchmark has no output hash;
- the ten-node run does not include a failure and a catch-up.

G.24 · `bend-runtime/hvm5/version1/maps/10-system.md`

System map

HIGH LEVEL



LOW LEVEL

```
src/
  README.md          actual module layout and import rules
  bendc.bend         source parser, package model, deploy compile, digest
  deploy.bend        deploy entrypoint; automatic compile
  sim.bend           consolidated system/space/instance/fork/cap model
  hvm5/
    term.bend        Term, Def, Program
    parse.bend       postfix inet package parser
    eval.bend        APP-LAM/SUP, DUP-SUP, Add, Tag/Match, fuel
    show.bend        canonical display
    hash.bend        structural digest
    main.bend        small reducer entrypoint
    run.bend         package runner
```

`model/` and `runtime/` do not exist. The model is currently consolidated in `src/sim.bend`; the deploy/package model is in `src/bendc.bend`. A future split must be a real extraction with updated imports and tests, not empty directories.

DEPENDENCY DIRECTION

```
official Bend 2 frontend -> tools/emit-inet.ts -> inet package
src/bendc.bend (bootstrap) --------------------> inet package
inet package -> parse -> eval
term -> eval
term -> show -> hash

sim -> term, parse, eval, show, hash, bendc
deploy -> sim, bendc
tests -> sim, bendc, hvm5
bench -> sim, term
PROOF -> sim, LAWS
```

The high-level rows (membrane, system contract, cluster, fork) are conceptual layers. In the current reference slice they are implemented as data and functions inside `src/sim.bend`; they are not separate modules yet.

No runtime module imports a host language or a TypeScript file. The only host effects are at executable boundaries (`deploy.bend`, `run.bend`, `main.bend`, tests, bench) and, in a native release, in the explicitly declared adapters.

OWNERSHIP

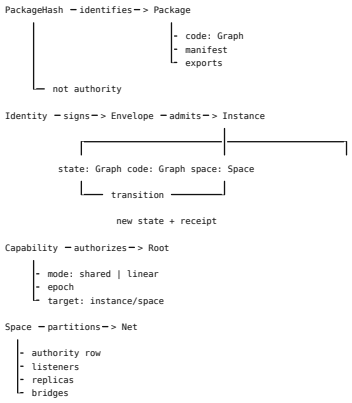
Component	Owns	Must not own
hvm5	term semantics, fuel	keys, tokens

Component	Owns	Must not own
membrane	capability table, epochs	term semantics
system	authority, names, sequences	user state
cluster	log, leader, quorum	term semantics
fork	shadow, Sup, drain	authority rows
CLI	parsing, signing	hidden mutation

G.25 · `bend-runtime/hvm5/version1/maps/20-types.md`

Type and data map

CORE TYPES



VALUE TREE

```
Value =
  | Ctor{name, fields: Value[]}
  | Nat{value: bigint}
  | Str{value: string}
  | Bool{value: boolean}
```

The graph encoding of a `Value` is a `CTOR` tree:

```
Ctor("Counter", [Nat(7)])
  -> CTOR("Counter", arity=1, space=S)
       aux0 -> CTOR("Nat", arity=0, payload=7)
```

CONTRACT PACKAGE

```
ContractPackage {
  format
  name
  version
  space
  stateType
  msgType
  init: Expr
  transition: Expr
  sourceHash
  code: Graph
  effects: []
}
```

`init` and `transition` are pure ASTs. The package validator checks totality and result type. The graph is the serializable code object.

CAPABILITY

```
CapRecord {
  id
  instance
  space
  root
  mode
  epoch
  rights
  live
}
```

The token is not in the record. The membrane maps `token -> CapRecord`.

FORK

```
Fork {
  id
  instance
  label
  oldRoot
  newRoot
  oldCode
  newCode
  phase: Testing | Merging | Draining | Abandoned
  inputs
  report
}
```

INVARIANTS AS TYPES

- `PackageHash` is opaque and not assignable to `Capability`.
- `LinearCap` is consumed by `send`; the result no longer contains it.
- `Space` is an index, not a string capability.
- `Receipt` is the only result of a mutating operation.
- `Fork` is not an `Instance` until merge.

G.26 · bend-runtime/hvm5/version1/maps/30-spaces.md

Space and authority map

TREE

```
root
├─ root/sys n0 n1 n2 system contract, quorum 2/3
├─ root/public n3 n4 n5 public contracts, quorum 2/3
│  ├─ root/public/acme
│  └─ root/public/demo
└─ root/private n6 n7 n8 private contracts, quorum 2/3
   └─ root/private/vault

n9 = client / observer, listen = ∅
```

AUTHORITY ROWS

Space	Listeners	Deploy	Bridges
root/sys	n0,n1,n2	system	none
root/public	n3,n4,n5	owner/system	none by default

Space	Listeners	Deploy	Bridges
root/public/acme	n3,n4,n5	acme owner	none
root/private	n6,n7,n8	private owner	none
root/private/vault	n6,n7,n8	vault owner	none

PARTITION RULE

```
listen(node, space) = space in node.listen or space is a descendant of one
reduce(node, op) =
  if not listen(node, op.space) then Denied{Space}
  else kernel.reduce(inducedSubnet(net, op.space), op.fuel)
```

A node can hold a capability for a space it does not listen to. It can route a send to a listener. Holding is not reducing.

REPLICATION

```
            n3 (leader)          n4 (follower)        n5 (follower)
send  -----> apply op             apply op             apply op
            state hash H         state hash H         state hash H
            commit if 2 hashes match
```

If `n3` dies, `n4` becomes leader and increments the public epoch. `n5` remains. If `n4` also dies, public is `ReadOnly`.

BRIDGE

```
root/public/acme  ---- bridge(B, shared) ----  root/private/vault
      ^                                              ^
   approval 1                                    approval 2
```

A bridge is an explicit capability edge. The isolation query reports it as a witness. Without both approvals, graft is rejected.

PRIVATE/PUBLIC PROOF

```
publicCapabilities = edges from public instances
privateCapabilities = edges into private instances
path(public, private) = ∅
therefore isolated(public, private) = true
```

The implementation prints the witness, not just the boolean.

FAILURE MAP

```
all alive       -> commit
n4 down         -> n3 + n5 commit
n3 down         -> n4 + n5 commit, epoch++
n3,n4 down      -> n5 alone, ReadOnly
n4 returns      -> catch up, follower again
```

G.27 · bend-runtime/hvm5/version1/maps/40-lifecycle.md

Lifecycle map

DEPLOY

```
client                membrane              system              subspace
  |                      |                     |                    |
  |-- signed deploy ---->|                     |                    |
  |                      |-- verify key ------>|                    |
  |                      |                     |-- authority ------>|
  |                      |                     |-- sequence ------->|
  |                      |                     |-- admit ---------->|
  |                      |                     |                    |-- graft
  |                      |                     |                    |-- init
  |                      |<--------------------|--------------------|
  |<-- instance + caps --|                     |                    |
```

SEND

```
client --cap + msg--> membrane --resolve--> instance
                        |                     |
                        |                     +-- connect msg root
                        |                     +-- reduce fuel
                        |                     +-- transition
                        |                     +-- new state hash
                        |<--------------------+
                        |-- replicate quorum -> n_leader,n_follower
                        |-- commit receipt
<-- receipt -------------|
```

FORK/TEST/MERGE

```
live root ----+
              |
              +-- DUP(L) --> shadow root
                                |
                            test inputs
                                |
                     +----------+----------+
                     |                     |
                  abort                  merge
                     |                     |
              erase shadow          collapse SUP(L)
                     |                     |
              live root              new root + epoch
                                           |
                                       drain old
```

CODE-AS-DATA

```
old package graph
      |
   patch graph
      |
      v
applyPatch -> new package graph -> hash -> package blob
                                     |
                                  graft copy
                                     |
                                  new instance
```

ATOMIC COMMIT

```
prepare -> validate -> compare-and-swap root+epoch -> append receipt -> drain
              |                |                          |
           failure          failure                    failure
              |                |                          |
           old root         old root                 old root or
                                                    idempotent replay
```

STATES

```
Registered -> Deployed -> Live
                    ^         |
                    |         +-- fork --> Testing --merge--> Draining --> Live
                    |                        |
                    +------------------------+--abandon--> Live
```

RECEIPT FIELDS

```
kind, id, instance, space, oldHash, newHash, fuel, epoch, sequence, status
```

No mutation is complete without a receipt.

G.28 · bend-runtime/hvm5/version1/maps/50-code.md

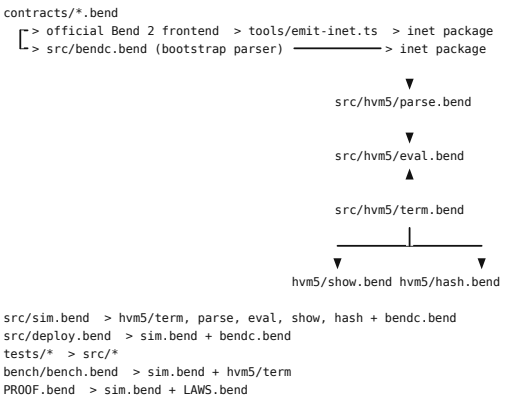
Code map

TREE

```
hvm5/version1/
├── README.md
├── LAWS.bend laws used by PROOF.bend
├── PROOF.bend topology/isolation/bridge proof gate
├── spec/ normative documents
├── plan/ phases, slices, gates, risks
├── maps/ this map set
├── compiler/ bootstrap compiler tooling
│   ├── README.md
│   └── bootstrap/
│       └── compile.ts bootstrap source -> inet package
├── tools/
│   └── emit-inet.ts official checked-AST -> inet backend
├── contracts/
│   ├── counter.bend checked Bend 2 contract source
│   ├── tally.bend renamed shape, proves name extraction
│   └── README.md
├── src/
│   ├── README.md actual module layout and import rules
│   ├── bendc.bend Bend 2 source parser, package model, deploy compile
│   ├── deploy.bend deploy entrypoint; auto-compiles source
│   ├── sim.bend ten-node system/space/instance/fork model
│   └── hvm5/
│       ├── term.bend Term, Def, Program (code == data)
│       ├── parse.bend postfix inet package parser
│       ├── eval.bend CEK-style inet reducer
│       ├── show.bend canonical term display
│       ├── hash.bend structural digest
│       ├── main.bend small reducer entrypoint
│       └── run.bend package runner
├── examples/
│   └── fork_merge.bend runnable fork -> test -> merge example
├── tests/
│   ├── eval.bend reducer cases
│   ├── pipeline.bend package -> reducer round trip
│   ├── deploy_compile.bend automatic deploy compile
│   ├── compiler_reject.bend rejection path
│   ├── compiler_names.bend renamed contract shape
│   └── run.bend full ten-node model gate
├── bench/
│   └── bench.bend W3-style repeated-send benchmark
└── legacy/ old v0 work; not normative, not imported
```

There is no `src/model/` or `src/runtime/` directory. The v1 reference slice consolidates those concerns into `src/sim.bend` and `src/bendc.bend`; see `src/README.md`. Do not create placeholder directories for a future split.

DEPENDENCY GRAPH



IMPORT RULES

1. `src/hvm5/*.bend` imports Base and other `hvm5` modules.
2. `src/bendc.bend` imports `hvm5` only.
3. `src/sim.bend` imports `hvm5` and `bendc.bend`.

4. `src/deploy.bend` imports `sim.bend` and `bendc.bend`.
5. Tests may import `src/*`; they must not duplicate the model.
6. `src/README.md` is the source of truth for the actual layout.
7. No circular imports; Bend definitions must appear before use.
8. The runtime does not import TypeScript, Node, or a JavaScript host.
9. `tools/emit-inet.ts` is build-time backend tooling, not a runtime.

NAMING

- Files: lowercase, underscore-free, one concern.
- Types: PascalCase.
- Functions: lowercase, descriptive.
- Term constructors: `Var`, `Lam`, `App`, `Dup`, `Sup`, `Era`, `Num`, `Add`, `Eq`, `Stuck`, `Tag0/1/2`, `Match0/1/2`, `Ref`.
- Space paths: `root/sys`, `root/public`, `root/private`.
- Node ids: `n0` .. `n9` as `Nat` 0..9.

TEST MAP

File	Covers
`tests/eval.bend`	APP-LAM, APP-SUP, DUP-SUP, `Add`, `Eq`, tags/matches, fuel, stuck
`tests/pipeline.bend`	postfix package parse and one-send reduction
`tests/deploy_compile.bend`	automatic deploy compile and reduction
`tests/compiler_reject.bend`	unsupported source is rejected, not miscompiled
`tests/compiler_names.bend`	renamed constructors/fields/parameters
`tests/run.bend`	ten nodes, replicas, quorum, kill/heal, catch-up, isolation, fork/test/merge, digest

External gates:

```
bun tools/emit-inet.ts contracts/counter.bend build/counter-official.inet
bend src/hvm5/run.bend build/counter-official.inet

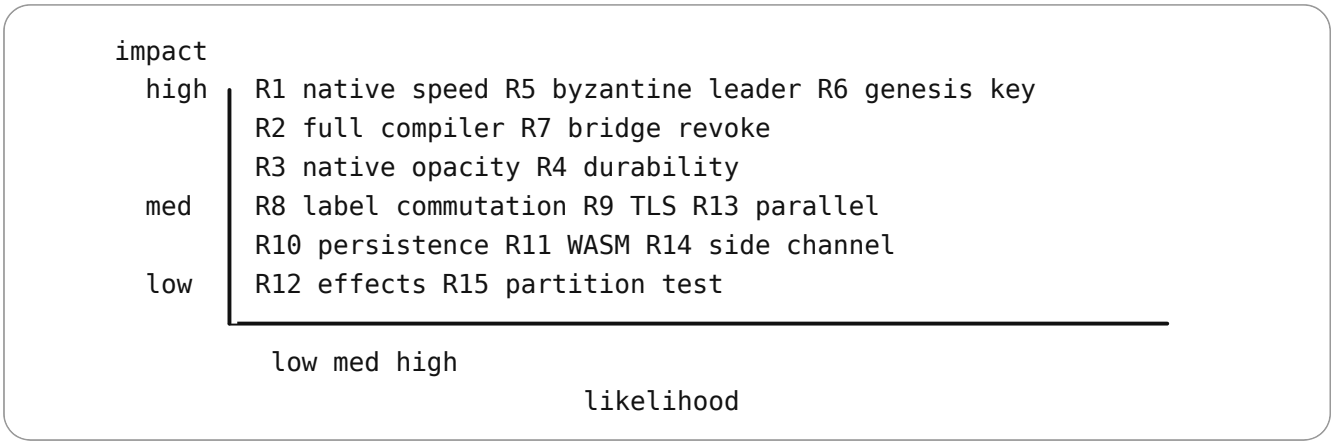
bun tools/emit-inet.ts contracts/tally.bend build/tally-official.inet
bend src/hvm5/run.bend build/tally-official.inet
```

Expected: `done Counter{2} fuel=0` and `done Tally{7} fuel=0`.

G.29 · bend-runtime/hvm5/version1/maps/60-risks.md

Risk map

HEAT MAP



STOP RULES

Stop the release if:

- a node can reduce outside its listen set;
- a linear capability survives two sends;
- a public instance has a path to a private instance;
- two replicas commit different state hashes;
- a merge leaves the old root reachable as live after the epoch flip;
- a benchmark hash does not match the expected output.

MITIGATION SEQUENCE

1. Partition and linearity (R3/R7) before cluster work.
2. Hash quorum (R5) before failover claims.
3. Durable snapshot (R4) before crash claims.
4. Native handle (R3) before security release.
5. Full compiler (R2) before calling it a Bend runtime.
6. Key rotation (R6) before production governance.

ACCEPTANCE EVIDENCE

Each mitigation needs:

- a test name;
- a spec section;
- a command;
- an output hash or receipt;
- a note in `plan/03-risks.md`.

Prose is not evidence.

G.30 · bend-runtime/hvm5/version1/maps/README.md

Maps

The maps are the navigation layer. The spec is normative; the maps are the fastest way to keep the whole system in view.

Map	Use
`00-goal.md`	the goal tree, observables, and failure conditions
`10-system.md`	layers, components, dependency direction
`20-types.md`	types, values, capabilities, forks
`30-spaces.md`	space tree, 10 nodes, quorum, bridges
`40-lifecycle.md`	deploy, send, fork, test, merge sequences
`50-code.md`	file layout, imports, tests
`60-risks.md`	risk heat map and stop rules

Read `00` and `10` before changing code. Read `50` before adding a file. Read `60` before claiming a security property.

G.31 · `bend-runtime/hvm5/version1/plan/00-plan.md`

HVM5 v1 implementation plan

Normative spec: `../spec/`
Status: executable reference plan. The spec defines what; this defines the order and the gates.

Current (2026-10-20): Phase 0 done; official-frontend emitter (`tools/emit-inet.ts`) works for the first-order contract fragment; the Bend 2 source compiler and auto-deploy compile work; the ten-node pure cluster model, fork/test/merge, isolation, capability gate, and benchmark work; full HVM4 rules, stable core-IR file, native crypto, real process/network cluster, and WAL remain open. See `../spec/11-acceptance.md` §4.

1. MISSION

Make the v1 lifecycle executable end to end on one machine, with a ten-node simulation that demonstrates:

- a genesis system contract;
- signed package admission;
- public and private subspaces;
- three replicas per subspace;
- a node failure with no downtime;
- a code-as-data fork, test, and merge;
- an isolation witness;
- a benchmark with a checked output hash.

The implementation is a reference, not the native release. Native HVM5 performance and full Bend 2 compilation are separate milestones.

2. WORKING RULES

1. Spec first. A behavior that is not in the spec is not implemented by accident; it is either added to the spec or recorded as a gap.
2. One commit point per operation.
3. The pure core has no I/O. Crypto, clocks, files, and sockets are at the membrane or CLI boundary.
4. Every state-changing operation has a receipt and a replay check.
5. A law or test is not weakened to make a gate pass. A gap is recorded instead.
6. The knowledge base is evidence, not authority.
7. Do not overwrite a version. Add `version2` when semantics change.

3. PHASES

PHASE 0 — GROUND TRUTH

- Keep `hvm5/reference/hvm5-mini/` as an **early draft** and proof reference, not as the v1 semantics.

- Keep the v0 net model and lattice as evidence.
- Pin the Bend 2 compiler version used for the proof gate.
- Freeze the DSL term grammar and the fuel rule.

Gate: `bend hvm5/reference/hvm5-mini/proof.bend` passes, and the v1 DSL grammar is documented in `spec/01-calculus.md`.

PHASE 1 — HVM5 DSL KERNEL IN BEND 2

- `src/hvm5/term.bend`: `Term`, `Def`, `Program` — code and data are one type.
- `src/hvm5/eval.bend`: substitution, APP-LAM, APP-SUP, DUP-SUP, ground data copy, `Num` / `Add` / `Eq` / `Tag` / `Match`, fuel, stuck results.
- `src/hvm5/show.bend`: canonical display for terms.
- `src/hvm5/hash.bend`: structural digest for quorum and state comparison.
- Tests: `tests/eval.bend`.

Gate: `bend tests/eval.bend` prints PASS; APP-LAM, APP-SUP, DUP-SUP, and a counter term reduce to the documented normal forms.

Status note (2026-10-20): the term machine in `src/hvm5/eval.bend` passes the basic APP-LAM, Add, Tag/Match, and DUP-SUP cases. APP-SUP normalization inside a `Sup` child and the full DUP-LAM/different-label rules are still open. Do not claim a complete IC evaluator until `tests/eval.bend` covers them.

PHASE 0.3 — OFFICIAL CHECKED CORE IR

- Add `--emit-core` to the official Bend 2 frontend (or a pinned plugin).
- Define the stable `hvm5.core.v1` format (`spec/01e-core-ir.md`).
- Do not implement a second layout parser/typechecker in Bend 2.

Gate (target): the official compiler checks arbitrary Bend 2 and emits a deterministic core module for the supported contract fragment. Current `tools/emit-inet.ts` traverses the checked AST directly; the stable core file is not implemented.

PHASE 0.4 — SELF-HOSTED COMPILER BOOTSTRAP

- Keep `compiler/bootstrap/compile.ts` as bootstrap-only tooling.
- Stage B: `tools/emit-inet.ts` lowers the official checked AST to an inet package. It is a backend, not a second frontend.
- Stage C: freeze the stable `hvm5.core.v1` file and make `src/bendc.bend` lower it. Do not reimplement layout/typechecking.
- Stage D: conformance corpus against BendRT.

Gate (current): `tools/emit-inet.ts` compiles `counter.bend` through the official frontend and `src/hvm5/run.bend` reduces it to `Counter{2}`.

Gate (target): a native `bendc` binary compiles `counter.bend` to the same package hash as the bootstrap frontend, and the runtime reduces it to `Counter{2}`.

PHASE 0.5 — BEND 2 CONTRACT COMPILER

Current implementation:

- `compiler/bootstrap/compile.ts` — bootstrap source -> inet package.
- `tools/emit-inet.ts` — official checked-AST -> inet backend.
- `contracts/counter.bend` and `contracts/tally.bend` — checked sources.
- `src/bendc.bend` — in-process source parser, package model, deploy compile.
- `src/hvm5/parse.bend` — canonical inet package parser. It defines the `Package` type, so there is no separate `package.bend`.
- Tests: `tests/pipeline.bend`, `tests/deploy_compile.bend`, `tests/compiler_reject.bend`, `tests/compiler_names.bend`.

Gate: `counter.bend` compiles to a stable inet package; the reducer evaluates `transition(init, Inc{by:2})` to `Counter{2}`. The runtime never parses the Bend source.

PHASE 1B — TRUE INET GRAPH KERNEL (REQUIRED FOR "INET-NATIVE")

Status: not implemented. The v1 reference reduces `Term` data; there is no agent/wire graph module yet. The paths below are targets, not existing files.

- `src/hvm5/graph.bend` : agents, ports, wires, roots, spaces.
- `src/hvm5/rules.bend` : active pairs, annihilate/commute/erase/duplicate, labels, fuel, typed clash.
- `src/hvm5/lower.bend` : lower `Term` to a graph and quote a graph back.
- `src/hvm5/graph_laws.bend` : local diamond and fuel laws.
- Tests: `tests/graph.bend` .

Gate: `bend tests/graph.bend` prints `GRAPH-PASS` ; a lower→reduce→quote cycle is observationally equal to the term evaluator on the generated corpus. This is the gate that justifies the word "inet-native". Until it passes, the runtime is a homoiconic term interpreter on BendRT.

PHASE 2 — CONTRACT TERMS

Current implementation:

- `contracts/counter.bend` and `contracts/tally.bend` — checked sources.
- Deploy compiles source automatically through `src/bendc.bend` ; the official-frontend path is `tools/emit-inet.ts` .
- Transition terms are parsed by `src/hvm5/parse.bend` and reduced by `src/hvm5/eval.bend` .
- Tests: `tests/pipeline.bend` , `tests/deploy_compile.bend` , `tests/compiler_names.bend` .

Planned extraction (not done): a reusable `src/model/contract.bend` / `value.bend` pair. The directory does not exist yet; the model is consolidated in `src/sim.bend` .

Gate: a message is applied by the inet reducer, not by a host evaluator, and the result is `Counter{2}` / `Tally{7}` .

PHASE 3 — IDENTITY AND CAPABILITIES (MODEL)

Current implementation: the key/envelope, capability/token, epoch, and receipt model is part of the consolidated `src/sim.bend` ten-node story. It is a deterministic numeric model, not native Ed25519.

Planned extraction (not done): `src/runtime/key.bend` , `cap.bend` , `receipt.bend` . Do not create empty directories before those files exist.

Gate: forged/replayed/wrong-space envelopes rejected; a linear capability is consumed once.

PHASE 4 — SPACES AND AUTHORITY

Current implementation: space paths, authority rows, listener sets, partitions, and bridge denial are functions/data in `src/sim.bend` ; exercised by the `ISOLATED` / `BRIDGE-OK` checks in `tests/run.bend` .

Planned extraction (not done): `src/runtime/space.bend` , `partition.bend` .

Gate: a node cannot reduce outside its listen set; a bridge needs two approvals.

PHASE 5 — SYSTEM CONTRACT AND REGISTRY

Current implementation: genesis, authority, admission, sequences, nonces, package registration, and instance state are consolidated in `src/sim.bend` ; deploy compilation is `src/bendc.bend` (or the official-frontend backend).

Planned extraction (not done): `src/runtime/system.bend` , `registry.bend` , `instance.bend` .

Gate: deploy only by authorized owner; duplicate sequence rejected; package hash binds.

PHASE 6 — CLUSTER

Current implementation: `src/sim.bend` holds the ten nodes, three subspaces, three replicas, leader/quorum model, kill/heal, and catch-up. The full report is `tests/run.bend` (via `sim_report`). There is no separate `node.bend` or `cluster.bend` .

Gate: one node down, two commit; restarted node reaches the same digest.

PHASE 7 — FORK/TEST/MERGE

Current implementation: shadow copy, `Sup` record, replay, merge, abandon, and drain are consolidated in `src/sim.bend` ; the full gate is `tests/run.bend` .

Planned extraction (not done): `src/runtime/fork.bend` , `patch.bend` .

Gate: live branch serves during test; merge is atomic; abandon leaves the live root untouched.

PHASE 8 — CLI, BENCHMARKS, DOCUMENTATION

Current implementation:

- `src/deploy.bend` — deploy entrypoint;
- `src/sim.bend` — ten-node model;
- `src/hvm5/main.bend` and `src/hvm5/run.bend` — reducer entrypoints;
- `bench/bench.bend` — W3-style benchmark;
- `tests/run.bend` — full model gate;
- `PROOF.bend` — proof gate.

The multi-command `src/cli.bend` and `src/bench.bend` are not implemented; use the entrypoints above. If a CLI is extracted later, it belongs in `src/cli.bend` and must call the model rather than duplicate it.

Gate: `bend src/sim.bend` prints PASS; `bend bench/bench.bend` prints a digest and interaction count; `bend PROOF.bend` green.

4. CRITICAL PATH

```
references
  -> graph kernel
  -> contract fragment
  -> capability/membrane
  -> spaces
  -> system contract
  -> cluster
  -> fork/merge
  -> scenario + benchmark
```

Do not start the cluster before the pure kernel hash is deterministic. Do not start fork/merge before the capability epochs are enforced. Do not claim isolation before the partition test is adversarial.

5. BUDGETS

Area	Budget
spec + maps	1 unit
kernel + canonical	2 units
contract fragment	2 units
membrane + registry	2 units
cluster + failure	3 units
fork/merge	2 units
CLI + benchmark	1 unit
hardening	2 units

A unit is a focused implementation slice with its own tests, not a calendar estimate.

6. DONE

Done means the acceptance matrix in `../spec/11-acceptance.md` passes in one run, the proof gate is green, and the remaining gaps are explicitly listed in `03-risks.md` . It does not mean the native HVM5 performance target is met.

G.32 · bend-runtime/hvm5/version1/plan/01-slices.md

Slices

Each slice has one seam, one gate, and one rollback. A slice is not a calendar task; it is a reviewable unit of behavior.

ID	Slice	Depends on	Gate
S00	Freeze reference and v0 evidence	—	reference proof green
S01	Canonical graph + hash	S00	same graph, same hash
S02	Kernel erasure/annihilation	S01	A1
S03	Data duplication + fuel	S02	A1, A2
S04	Contract value + expression evaluator	S01	A5
S05	Contract package + validation	S04	total transition
S06	Ed25519 envelopes	S01	A4
S07	Capability membrane + linearity	S06	A4, X1–X5
S08	Space tree + authority	S01	A3
S09	Partitioned reducer	S03, S08	X6
S10	Registry + genesis system contract	S05, S07, S08	A6
S11	Deploy/graft/instance	S10	deploy receipt
S12	Send pipeline	S11	send receipt
S13	Node log + quorum	S12	A7
S14	Ten-node topology	S13	ten nodes start
S15	Failure + catch-up	S14	X9, failover
S16	Fork + SUP record	S12	A9
S17	Test replay + report	S16	A9
S18	Merge/abandon/drain	S17	A9, X10, X11
S19	Code-as-data patch	S05, S16	new package hash
S20	Isolation query + witness	S07, S08	A8
S21	CLI	S11–S20	all commands
S22	Benchmarks	S21	JSON evidence
S23	Full scenario	S14, S18, S20	A11
S24	Bend model + proof	S05	A12
S25	Hardening / gap review	all	09 checklist

STATUS LEGEND

- `TODO` — not started.
- `PARTIAL` — model or test exists, but a named gate is missing.
- `DONE` — gate passes and evidence is recorded.
- `DEFERRED` — explicitly outside v1.

NOTES

S03 and S09 are the two places where a wrong shortcut would invalidate the security story. Do not fake commutation, and do not let a node reduce a pair outside its listen set.

S18 is the only operation with a compare-and-swap on the visible root. It must be tested with a crash before and after the swap.

G.33 · bend-runtime/hvm5/version1/plan/02-gates.md

Gates

Run these from the repository root. A gate is green only with the expected output. No gate is waived because another gate is green.

G0 — EARLY-DRAFT REFERENCE

```
bend hvm5/reference/hvm5-mini/proof.bend
```

Expected: `All terms check.` This is a historical reference, not v1.

G1 — INET REDUCER

```
bend hvm5/version1/tests/eval.bend
```

Expected: `EVAL True .`

G2 — CHECKER

```
bend hvm5/version1/src/bendc.bend --check-only
bend hvm5/version1/src/sim.bend --check-only
bend hvm5/version1/tests/run.bend --check-only
```

Expected: `All terms check.`

G3 — V1 LAWS

```
bend hvm5/version1/PROOF.bend
```

Expected: `All terms check.`

G4 — CONTRACT → INET → REDUCER

```
bend hvm5/version1/contracts/counter.bend --check-only
bend hvm5/version1/src/deploy.bend hvm5/version1/contracts/counter.bend
bend hvm5/version1/tests/pipeline.bend
bend hvm5/version1/tests/deploy_compile.bend
```

Expected:

```
All terms check.
DEPLOY Counter{0} PKG <u32> DIGEST <u32>
PIPELINE done Counter{2} fuel=0
DEPLOY-COMPILE done Counter{2} fuel=0
```

The official-frontend backend can emit the same package:

```
bun hvm5/version1/tools/emit-inet.ts \
  hvm5/version1/contracts/counter.bend build/counter-official.inet
bend hvm5/version1/src/hvm5/run.bend build/counter-official.inet
```

Expected: `done Counter{2} fuel=0 .`

The renamed shape proves name extraction:

```
bun hvm5/version1/tools/emit-inet.ts \
  hvm5/version1/contracts/tally.bend build/tally-official.inet
bend hvm5/version1/src/hvm5/run.bend build/tally-official.inet
```

Expected: `done Tally{7} fuel=0` (init 5 + Inc 2).

The bootstrap frontend can also emit a package file:

```
bun hvm5/version1/compiler/bootstrap/compile.ts \
  hvm5/version1/contracts/counter.bend build/counter.inet
bend hvm5/version1/src/hvm5/run.bend build/counter.inet
```

Expected: `done Counter{2} fuel=0 .`

G5 — TEN-NODE SIMULATION

```
bend hvm5/version1/src/sim.bend
bend hvm5/version1/tests/run.bend
```

Expected: the final line is `PASS`, and the report has:

```
NODES 10
PUB Counter{4}
PRIV Counter{3}
N4 Counter{3}
ISOLATED True
PUB-OK True
PRIV-OK True
CATCHUP-OK True
REPLICAS-OK True
QUORUM-OK True
BRIDGE-OK True
CAP-OK True
DIGEST <u32>
PASS
```

G6 — DETERMINISTIC REPLAY

Run the simulation twice:

```
bend hvm5/version1/src/sim.bend
bend hvm5/version1/src/sim.bend
```

The output, including `DIGEST`, must be identical.

G7 — BENCHMARK

```
bend hvm5/version1/bench/bench.bend
```

Expected:

```
BENCH sends=100 ms=<positive>
STATE Counter{100}
```

It is reference evidence, not a native performance claim.

G8 — NEGATIVE/ADVERSARIAL CASES

```
bend hvm5/version1/tests/compiler_reject.bend
bend hvm5/version1/tests/compiler_names.bend
```

Expected: `REJECT True` and `RENAMED True`.

The simulation contains these checks:

- two public replicas down → `QUORUM-OK` (send denied, state unchanged);
- no public/private capability → `ISOLATED True`;
- bridge added → `BRIDGE-OK True` (isolation is false);
- killed node healed → `CATCHUP-OK`;
- fork candidate tested → `PUB Counter{4}` after merge.

The proof gate covers the closed topology and bridge laws.

G9 — NATIVE BENDRT

```
bend hvm5/version1/src/hvm5/run.bend -o build/run
./build/run build/counter.inet
bend hvm5/version1/src/sim.bend -o build/sim
./build/sim
```

Expected: same outputs as the interpreted runs.

G10 — DOCUMENTATION INDEX

`AGENTS.md` lists the docs index, the spec index, the plan index, the maps, and the commands. A new contributor can find every part without searching the tree.

GATE DISCIPLINE

- A red gate blocks the next slice.
- A test may be added, but a green test may not be deleted to make a red gate pass.
- A spec change requires a gate update.
- A benchmark number requires a recorded machine, commit, and digest.
- `bend PROOF.bend` green never excuses a red runtime gate.

G.34 · bend-runtime/hvm5/version1/plan/03-risks.md

Risks, gaps, and decisions

1. HIGH-SEVERITY GAPS

ID	Gap	Why it matters	v1 handling
R1	The Bend 2 interpreter is a reference normalizer, not a native optimizing HVM5	Performance claims would be false	Bend 2 + BendRT; native tuning is a separate gate
R2	Full Bend 2 → graph compiler is absent	Only the contract fragment is deployable	Sample contracts + validator; compiler is future
R3	Membrane tokens are a Bend model, not native `law` handles	A compromised runtime can forge a token	Opaque numeric tokens + membrane table; native handle future
R3b	Ed25519 is a deterministic signature model in the simulation	Key forgery is not cryptographically hard in the model	Host Ed25519 adapter is a separate gate
R4	No durable WAL/fsync	Crash recovery is modeled, not physically durable	JSON snapshot in CLI; WAL future

ID	Gap	Why it matters	v1 handling
R5	No Byzantine consensus	A lying leader can censor	Hash-quorum replication; BFT future
R6	No key rotation/multisig	Genesis key is a single point of governance failure	Documented; future
R7	Bridge revocation is membrane-only	A copied bridge record could outlive a revoke	Linear bridge default; test
R8	Different-label DUP/SUP and DUP-LAM are not in the v1 evaluator	Some optimal-sharing programs are stuck	Explicit `Stuck` boundary; full IC future
R16	The term interpreter is not a first-class inet graph kernel	"Inet-native" would be a false claim	Phase 1b graph kernel + lower/quote conformance gate
R17	BendRT is a flat evaluator, not an active-pair host	The host does not provide inet scheduling or wire identity	Object-level graph kernel in Bend 2; native host is a separate target
R18	The official-AST inet emitter covers only the first-order contract fragment	Larger Bend 2 programs reject at lowering	Stable core IR + broader lowering; reject, never miscompile

2. MEDIUM-SEVERITY GAPS

ID	Gap	Handling
R9	No TLS in the in-process CLI	Local only; native transport future
R10	No Durable Objects/D1	In-memory store; swap the backend
R11	No WASM Worker ABI	Keep `deploy/` v0 prototype, do not claim it
R12	No native effect declarations beyond the manifest	Transition path is pure; effects future
R13	No parallel scheduler	Sequential reducer; observational equivalence required
R14	No side-channel review	OS boundary; future audit
R15	No real network partition	Simulated `down` ; future fault injector

3. DECISIONS

D1 — ONE GRAPH REPRESENTATION

Accepted. Code, state, patch, and package are graphs. A source AST is a frontend artifact, not the storage format.

D2 — TYPED FRAGMENT FIRST

Accepted. v1 uses `ERA` , `CON` , `CTOR` , `DUP` , `SUP` with same-label annihilation and data duplication. Full different-label commutation is deferred with a stated boundary.

D3 — HASH-QUORUM, NOT VOTING

Accepted. Replicas must return the same hash. A different hash is divergence, not a vote.

D4 — SYSTEM CONTRACT IS CONTROL PLANE

Accepted. A system-contract outage stops new deploys, not running subspaces with quorum.

D5 — PRIVATE KEYS STAY IN THE CLIENT

Accepted. The runtime receives a signature, a public key, and canonical bytes. It never receives a private key. The simulation keyring is a test fixture.

D6 — NO AMBIENT PARENT AUTHORITY

Accepted. `root/public` does not automatically have authority over `root/public/acme`. A bridge is explicit.

D7 — RECEIPTS ARE FIRST-CLASS

Accepted. Every mutating operation has a receipt with old/new hashes and fuel. A successful call without a receipt is a bug.

4. OPEN QUESTIONS

1. Should the system contract be a normal Bend package or a hand-written genesis object? v1 uses a normal package plus a genesis state.
2. Should a bridge be linear by default? v1 defaults to linear.
3. How long is the drain window? v1 uses a configured number of operations, not wall time, so replay stays deterministic.
4. Should the registry store source or only compiled graphs? v1 stores both as content blobs; source is optional.
5. Can a fork change its space? No in v1; a cross-space move is a new deploy.

5. REVIEW RULE

A risk is closed only when:

- the spec section that claims the behavior is updated;
- an executable test covers the failure;
- the acceptance matrix references it;
- this file records the decision and the residual risk.

A risk is never closed by prose alone.

G.35 · `bend-runtime/hvm5/version1/plan/legacy/hvm5-bendrt-contract-runtime-plan-v0.md`

Inet-native HVM5 contract runtime — implementation plan

Normative specification: `docs/hvm5-inet-contract-runtime-spec.md`
Status: implementation plan only; no implementation is authorized by this document.

The normative specification defines layers, domain types, wire/SUP/DUP semantics, protocol state machines, invariants, and proof obligations. This file defines research, comparison, sequencing, budgets, and gates. The specification takes precedence over this plan.

Substrate: one live interaction net. Deploy grafts a subnet; invoke connects a wire (forming an active pair); a capability is a wire; cost is the interaction count; code, state, and snapshots are one kind of thing (graphs); agreement rides confluence. BendRT is the native host process (evaluator + membranes), not the isolation layer.

0. GOAL AND CURRENT BASELINE

GOAL

Build a Linux service, implemented in Bend 2 and compiled with BendRT, that hosts one live HVM5 net and can:

1. graft deployed packages as private subnets of the live net;
2. host many subnets isolated by disconnectivity (no wire, no interaction);
3. authenticate grafts and code replacements with a deployer's private key;
4. hand out wires as capabilities to external clients;
5. pass wires between subnets for internal sends;
6. persist subnets, wire bindings, nonces, and receipts as serialized graphs;
7. bound every reduction by interaction fuel plus graph-size limits;
8. make unauthorized cross-subnet access structurally impossible: interaction requires connection.

The first execution targets are:

1. native Linux BendRT for development and durable hosting;

2. a single-thread HVM5 WASM module inside a Cloudflare Worker, reached through the same deploy/send/listen protocol.

The WASM architecture follows Bend PR #866 / `AdrielSantana/metal-bending` : the Bend C runtime is compiled by Emscripten. The Worker is **not** the browser-page target; it uses the same runtime family with a Worker-safe ABI.

Browser-page threading, `SharedArrayBuffer` , `Window` , audio, and WebGPU remain separate targets. The Cloudflare Worker is a server-side WASM host and MUST NOT depend on DOM/window/pthread APIs.

The result is smart-contract-like execution. It is **not a blockchain** unless a separate consensus, replication, and finality layer is added.

CURRENT EVIDENCE

The current Bend repository contains `demos/pure_hvm5_mini/` :

- `main.bend:1-72` describes a port of `hvm5.c` ;
- `main.bend:82-200` models HVM5 templates, terms, cells, continuations, and evaluator state;
- `main.bend:954` contains the parser entry point;
- `main.bend:1388-1439` expands references from the evaluator book;
- `main.bend:1688-1775` contains the evaluator and driver;
- `main.bend:1876-1916` loads a book and runs `@main` ;
- `main.bend:333-339` hard-codes the sample HVM5 source in `program()` .

It runs as an ordinary Bend program and its `PROOF.bend` passes. It is a useful reference evaluator, not yet a deployment service, capability system, or Bend-to-HVM5 compiler. The current README still lists C, Metal, CUDA, and JavaScript targets, and states that there is no separate compilation or incremental build.

This workspace now vendors that reference as `hvm5/main.bend` , `hvm5/laws.bend` , and `hvm5/proof.bend` ; the proof gate passes. The web-WASM compiler from Bend PR #866 is vendored for build experiments. `deploy/flake.nix` pins the Emscripten toolchain and PR commit. The generated browser page is a real WASM conformance artifact; the Cloudflare Worker ABI remains a separate single-thread build and is not complete until its exports are defined.

1. SCOPE AND NON-GOALS

IN SCOPE

- HVM5 net reducer running inside a BendRT native executable;
- grafting packages as private subnets (per-package books as the interim mechanism toward true live-net grafting);
- wires as capabilities, including linear wires that cannot be duplicated and are revoked by severing;
- signed graft/replace envelopes;
- bounded reduction (fuel = stop reducing) plus graph-size admission;
- internal and external sends over explicitly passed wires;
- snapshots as serialized subgraphs, plus a WAL for membrane crash recovery;
- audited, narrow host effects behind membranes;
- native BendRT libraries as trusted host adapters only.

NON-GOALS FOR THE FIRST RELEASE

- blockchain consensus or multi-node replication;
- arbitrary native C/JS code uploaded by users;
- arbitrary IO from contract code;
- source-code confidentiality from the runtime operator;
- browser execution of the full native runtime;
- browser clients and browser-local contract execution;
- unrestricted hot patching of live code;
- a general-purpose Linux sandbox implemented only in Bend;
- a runtime facet type system (facets are separate wires);
- a separate state subsystem (state is a subgraph).

2. THE SEVEN-LEVEL BODY OF KNOWLEDGE

1. PRINCIPLES

- **Connectivity is authority:** a subnet interacts only across wires it holds.
- **No ambient authority:** there is nothing to look up — no global namespace, no shared book; hashes identify content, wires authorize.
- **Linearity where revocation matters:** a linear wire cannot be duplicated, so severing it revokes completely and provably.
- **Confluence:** any reduction order converges; independent replays agree structurally. Every calculus extension must preserve the diamond or say so.
- **Cost is computation:** each interaction is one countable step; fuel exhaustion is "stop reducing," not a separate accounting layer.
- **Homoiconicity:** code, state, and snapshots are one kind of thing (graphs); graft, serialize, and migrate are one kind of operation.
- **Atomicity:** failed reduction commits neither subgraph nor effects.
- **Authenticity:** graft/replace authority comes from signature verification at the host membrane.
- **Least authority:** subnets receive only explicitly passed wires.

1.1 INET CORE REFERENCE MODEL

The runtime must freeze this model before calling itself inet-native. The current `pure_hvm5_mini` program is a sequential evaluator of related terms; this section describes the graph semantics that the runtime must preserve.

```
Agent = kind + principal port + ordered auxiliary ports
Wire  = one edge joining two ports (or one external root)
Net   = agents + wires + named external roots
```

Required semantic vocabulary:

- **Principal/auxiliary ports:** only a principal-principal pair is an active redex; auxiliary wiring is carried through the local rule.
- **SUP:** a correlated superposition/branching agent. It is not an ordinary host-language copy and must preserve branch labels/correlation.
- **DUP:** a lazy duplication agent. It distributes a value through the graph according to the HVM5 duplication rules; a capability wire marked linear may not enter a `DUP` path.
- **ERA:** erasure. Dropping a graph must recursively release its reachable cells/agents and any owned linear wires.
- **Annihilation:** equal active agents connect corresponding auxiliary ports and remove the pair.
- **Commutation:** different active agents create the exact local replacement graph prescribed by the rule table; no global traversal is allowed.
- **Duplication/interaction distribution:** `DUP` / `SUP` rules must preserve sharing and labels, not serialize a shallow copy of a host object.
- **Confluence obligation:** every new pair rule, packed-number fast path, fusion rule, and effect boundary needs a local-confluence/diamond argument or must be excluded from the pure net.

The reference rule table must be copied from the selected HVM5 specification / `hvm5.c` implementation and represented in a machine-checkable test table. The plan must not silently replace `SUP` with a normal list, `DUP` with a host reference count, or a wire with a serializable integer: those substitutions destroy the inet semantics and the ocap claim.

2. MODELS

PACKAGE SUBNET

An immutable, content-addressed subnet grafted onto the live net — per-package books are the interim mechanism, true grafting the target (see §6). References are subnet-local wiring details, never cross-subnet authority tokens.

CONTRACT INSTANCE

```
Instance = package_hash + instance_id + state_subgraph + version + owner_identity
```

State roots are hashes of subgraphs.

CAPABILITY

```
Capability = wire (port) + linearity flag + target subnet
```

A linear wire must never pass through a duplicator, so severing revokes it completely. A shared wire may be passed freely; revocation then needs the proxy pattern. There is no facet type: an invoke wire and an update wire are simply different wires.

The `wire + linearity + target` struct is membrane-only state, never user data (spec §5.5). Any design showing a user-constructible capability struct is rejected the way `Cap{U32}` is.

SEND (REPLACES "TRANSACTION")

```
Send → connect wire → reduce to budget → snapshot subgraph → receipt
```

Invocation needs no dispatch: plugging a wire facing an agent's principal port creates an active pair, and reduction is the call.

HOST EFFECT

An operation outside the pure net, reachable only over a supplied wire and implemented by a trusted Bend effect adapter behind the membrane.

3. METHODS

1. canonicalize and hash a package (code identity);
2. verify a signed graft/replace envelope at the membrane;
3. parse HVM5 source or load a serialized subnet;
4. validate subnet-local wiring and its effect manifest;
5. graft the subnet onto the live net (books first, true graft second);
6. mint export wires (linear where revocation is required);
7. reduce received sends to budget;
8. pass wires between subnets by message;
9. snapshot the resulting subgraph and append a receipt atomically;
10. sever wires (or kill proxies) to revoke; rotate by re-grafting;
11. replace code only by signed redeploy plus an explicit migration send.

4. RULES

- No subnet can name, look up, or resolve another subnet's wires or entries: there is no namespace to query.
- A hash, string, index, or package name is never a capability.
- A linear wire must never pass through a duplicator; checker and runtime both enforce this.
- Every calculus extension (native ops, fusion, effects) must preserve the diamond property, or agreement is void — audited per extension.
- User packages cannot import arbitrary `.c` or `.js` effects.
- User packages cannot use the evaluator's `@unsafe` host escape.
- Host effects must be allowlisted in the package manifest and reachable only over an injected wire.
- Every send has interaction fuel plus graph-size, input, output, and nesting limits.
- Out-of-fuel, invalid, or failed reductions commit nothing.
- Graft and replace requests require a valid signature and fresh nonce/height.
- Invoke wires and update wires are distinct wires.
- A package upgrade is redeploy + migration; in-place mutation is forbidden.
- A wire token must not be logged, placed in URLs, or exposed in errors.
- The runtime must reject malformed, cyclic, oversized, and out-of-budget graphs.

Exceptions must be explicit and confined to the trusted runtime or test mode.

5. ARTIFACTS

- package archive/subnet snapshot;
- package manifest;
- signed graft envelope;
- signed replace envelope;
- contract instance record;
- export wires (linearity flagged);
- sever/revocation record;
- subgraph snapshot;
- append-only send/receipt record;
- proof and verifier report (including per-extension diamond audits);
- compatibility/version record;
- audit and benchmark report.

6. COMPONENTS

PACKAGE MANIFEST

```
format_version
package_hash
compiler_version
hvm5_version
export_wires
required_wire_types
effect_manifest
entry_abis (wire shapes)
state_schema_version
fuel_policy
memory_policy
```

GRAFT ENVELOPE

```
domain_separator
network/runtime_id
owner_public_key
package_hash
instance_nonce
initial_subgraph_hash
requested_wires
manifest_hash
signature
```

SEND REQUEST

```
wire_token (external) or wire (internal)
message + reply wire
canonical_input
fuel_limit
fuel_price
valid_after (sequence/height)
signature when required
```

RECEIPT

```
request_hash
instance_id
code_hash
previous_subgraph_hash
new_subgraph_hash
status
fuel_used (interactions)
effects/emissions
sequence_number
runtime_version
```

7. CANONICAL EXEMPLAR

Alice signs and grafts `counter-v1` with an invoke wire and a linear update wire held only by Alice. The runtime grafts a private subnet and returns the wires; only the public contract ID is published. Alice sends the invoke wire to Bob. Bob can send `increment` plus a reply wire, but cannot update code, read the private subgraph, or reach any other subnet —

no wire, no interaction. Alice signs `counter-v2` plus a migration; the runtime checks the expected old hash, runs the migration send transactionally, and returns a versioned receipt. Severing the linear update wire revokes update power completely: a linear wire has no copies.

3. CONTRACT LATTICE

GOAL

Observable success:

- a signed package can be deployed through an API;
- the package runs inside one BendRT-hosted HVM5 service;
- authorized external and internal calls succeed;
- unauthorized sends, forged wires, replayed replaces, over-budget sends, and cross-subnet references are rejected;
- state survives restart without violating receipt order;
- all gates pass on Linux.

Metrics for the first release:

- deterministic replay of every successful pure send (same net, same result, any reduction order);
- zero cross-subnet interactions without a wire in adversarial tests;
- bounded fuel and graph size for every send;
- crash recovery at every commit boundary;
- no private key material in the runtime process;
- every enabled calculus extension carries a passing diamond audit.

ANCHOR

- The `pure_hvm5_mini` evaluator is the initial semantic reference.
- HVM5 package code is data executed by the evaluator, not native C.
- The live net is the isolation substrate; the BendRT process is the host, not the boundary.
- Per-package books are the interim mechanism; true grafting is the target.
- Linear wires are the revocation story; shared wires need proxies.
- Native BendRT modules are trusted host adapters only.
- `PROOF.bend` passing is necessary but not a security proof of C effects.
- No ambient authority: there is no namespace to query.
- Code hashes are integrity identifiers, not permissions.
- The first release is a single-node service, not a consensus system.

MAP

Component	Owner	State	Forbidden dependency
HVM5 net reducer + grafter	Runtime core	live net, subnets, wires	network, filesystem
package linker/grafter	Runtime core	subnet snapshot/manifest	arbitrary host effects
wire membrane	Runtime core	wire↔token bindings, linearity flags	user-constructible wire tokens
send manager	Runtime core	state-root queue/sequence/reply wires	direct contract IO
persistence adapter	trusted effects	WAL/snapshots	contract-controlled paths
crypto adapter	trusted effects	public-key verification	private-key storage
API transport	trusted effects	sockets/request framing	evaluator internals
contract code	deployer	HVM5 graph/state	native C/JS, ambient refs
native modules	runtime operator	compiled code	untrusted deployment

PLACE

Capability × target	Allowed	Forbidden
Pure HVM5 package	grafted subnet, passed wires, bounded reduction	raw pointers, global book lookup
BendRT host	evaluator, broker, persistence adapters	treating tenant code as native plugins
Linux	user-space process, restricted service account	assuming BendRT is ring-0 code
Browser client	deferred after native release	part of the first release
Native module	trusted crypto/storage/network bridge	arbitrary tenant upload

BUDGET

Initial implementation budget:

- no full Bend-to-HVM5 compiler in the first vertical slice;
- HVM5 source or prebuilt book only;
- pure contracts only;
- one Linux process;
- one append-only local store;
- one signature algorithm;
- Ed25519 for Draft 0, with canonical domain-separated envelopes;
- no security claim for books-only isolation;
- no browser runtime;
- no browser client or browser-local evaluator;
- no consensus or multi-node deployment.

Every slice must state files, interfaces, tests, and rollback before editing.

GATE

No slice advances until its local gate and the full regression/proof gate pass. The final gate is a clean graft, send, replace, restart, recovery, and adversarial-isolation run on Linux, plus a passing per-extension diamond-preservation audit.

TRACE

Maintain these versioned files during implementation:

```
lattice/goal.md
lattice/invariants.md
lattice/owners.md
lattice/placement.md
lattice/seams.md
lattice/graph.md
lattice/slices/SLICE-NNN.md
lattice/checkpoints/CP-NNN.md
lattice/gaps.md
lattice/rejects.md
lattice/fitness.md
```

4. IMPLEMENTATION SLICES

Each slice is one admissible change. Do not batch slices. Slice numbers are stable IDs, not execution order: follow §13 and each slice's `Depends on`. Minimum lattice before SLICE-000 through SLICE-015: `invariants.md`, `seams.md`, and one `slices/SLICE-00x.md` each with files, gate, and rollback. The remaining lattice files grow from SLICE-020 onward.

SLICE-000 — FREEZE BASELINE

Goal: record the exact Bend/HVM5 source revision and current outputs.

Touches: source snapshot metadata, baseline commands, this plan.

Gate: `bend demos/pure_hvm5_mini/PROOF.bend` passes; sample output and unsafe warning are recorded.

Rollback: delete only baseline metadata.

SLICE-005 — FREEZE THE HVM5 RULE TABLE

Goal: select and version the exact HVM5 reference rule table for `SUP`, `DUP`, `ERA`, constructors, applications, packed values, reference expansion, fusion, and collection.

Touches: reference-rule artifact, rule-table tests, reference revision.

Gate: every enabled pair has a named rule, a differential test, and either a confluence argument or an explicit non-confluent boundary. No optimized path is enabled before its reference-equivalence test passes. The audit artifact is fixed-format: a pair-rule table (left/right kinds → replacement net, sharing preserved), a scheduler differential (at least two reduction orders converge), and either a local-confluence note or an explicit `NonConfluent{reason}` flag that strips agreement claims for graphs using that rule. Fusion, packed numbers, and effects start in the non-confluent bucket.

Rollback: disable the optimized rule; retain the reference rule.

SLICE-010 — EXTRACT THE EVALUATOR API

Goal: separate the semantic reference reducer from the service shell and define the pure net seam:

```
parse(source) -> OpenSubnet
reduce(net, root, fuel, limits) -> Result<Graph, NetError>
```

Touches: parser continuation, `book_run`, evaluator result types, tests.

Forbidden: networking, persistence, capabilities, compiler changes.

Gate: the original sample still produces the same normal form and interaction count; malformed source returns a structured error rather than printing only. The API MUST return structured graphs/errors, not print or perform host IO.

SLICE-015 — DEFINE THE FIRST-CLASS NET KERNEL SEAM

Goal: add the `Agent` / `Port` / `Wire` / `Root` model and an atomic local active-pair rewrite seam. The current Bend array/book evaluator remains the reference implementation until the live-net representation is equivalent.

Gate: a two-agent test can connect, reduce, disconnect, and collect without global lookup; disconnected subnets cannot interact; failed rewrites leave the old graph intact.

Forbidden: exposing host array indexes as ports; treating a book index as a capability; calling this the security boundary yet.

SLICE-020 — NATIVE FUEL: STOP REDUCING

Goal: make fuel native to the substrate: interaction budget plus heap-cell, graph-size, stack, input, output, and nesting limits in evaluator state. Exhaustion means "stop reducing," not a separate accounting layer.

Touches: `S`, `Out`, `run`, `drive`, allocation and collection paths.

Gate: finite programs complete; infinite/oversized programs return `OutOfFuel` or `OutOfMemory`; fuel is enforced on the fusion/copier paths, not just the main loop; failed sends expose no partial subgraph.

SLICE-030 — PRIVATE SUBNETS (BOOKS FIRST, GRAFT SECOND)

Goal: load each package into an independent private book with validated entry wires, and design the graft interface toward one live net. Books are the demo's mechanism and the interim step; wire-connected grafting is the target.

Touches: package loader, reference validation, manifest parser, graft interface stub.

Forbidden: resolving references through a global tenant book.

Gate: package A cannot resolve a definition from package B by index, name, or hash; package-local references remain valid after multiple packages load. This is the **book gate** (necessary, not sufficient). The **security gate** — no wire implies no redex even with all books loaded and hashes known — does not exist until graft plus wire (SLICE-015, SLICE-030, SLICE-040), and no isolation claim may cite this slice before then.

SLICE-040 — WIRE PRIMITIVE PLUS LINEARITY

Goal: commit the wire representation: node shape (a wire is connection, not data), linearity flag, serialization form, and the rule that linear wires never pass a duplicator — enforced by checker and runtime alike.

Gate: user code can receive, pass, and send on a wire but cannot construct, inspect, enumerate, or forge one from integers; linear wires provably never cross a duplicator (tested on every path that ever holds one).

Reject: ordinary public `Cap{U32}` constructors.

The checker MUST reject every path in which a `LinearWire` reaches `DUP`, `SUP`-based copying, a persistent shared receive, or a serializer. The runtime MUST repeat the check at graft and message boundaries.

SLICE-050 — WIRES-AS-FACETS; SEVERANCE PRIMARY, PROXY FALLBACK

Goal: separate rights by separate wires; revocation primarily by severing linear wires, proxy pattern only for shared wires. There is no runtime facet type and no runtime revoke op (a severed linear wire has no copies to take back; an already-shared non-linear wire cannot be taken back — Rholang rule).

Design: an invoke wire and an update wire are different unforgeable wires; update wires are linear by default. Revocability for shared wires is a forwarder process in contract code that consults a live flag; the runtime only needs to support the pattern, not implement it.

Gate: holding the invoke wire cannot update; the update wire cannot be minted by contract code; severing a linear wire leaves no residual copies (tested by attempted use-after-sever on every path that ever held it); killing a proxy stops future sends; direct shared wires are documented as non-revocable.

The persistent public receiver MUST be specified here: its continuation, state ownership, queue behavior, and fresh linear reply-wire behavior. A one-shot linear update wire MUST return a fresh update continuation after successful replacement.

SLICE-060 — IDENTITY AND SIGNATURES

Goal: verify signed graft/replace envelopes.

Depends on: the minimal crypto adapter from SLICE-120. The envelope format may be specified before it, but verification cannot land without it.

Design: the private key stays in a client wallet/CLI. The runtime stores only public keys and verifies Ed25519 signatures through a narrow crypto effect. Signed bytes use canonical encoding, domain separation, runtime/instance IDs, and a per-owner monotonic sequence. `valid_after` is not used in the single-node release.

Gate: invalid signatures, wrong domain, stale nonce, wrong previous hash, and replay are rejected. Canonical serialization has golden test vectors.

SLICE-070 — SENDS AS BOUNDED TRANSACTIONS

Goal: make a send a bounded transaction: connect the wire (active pair), reduce to budget, snapshot the subgraph, emit a receipt.

Design: active-pair invocation, snapshot commit, serialized per-instance execution initially. Locks: one mutex per state root, one global graft lock; graft on a disjoint root may proceed during a reduce, graft on a reducing root waits; no lock is held across a host-effect boundary. Reentrant sends enter a defined queue/savepoint policy; they do not mutate a second state root concurrently.

Gate: crash injection before/after every commit step gives either the old or new subgraph, never a mixed one.

SLICE-080 — PERSISTENCE AS SERIALIZED GRAPHS

Goal: persist subnets as serialized graphs (a snapshot *is* a subgraph), plus instance metadata, wire bindings, nonces, and receipts.

Initial implementation: append-only WAL for the membrane log plus snapshots through trusted Bend effects. No contract-controlled filesystem paths. Graph serialization must preserve sharing/dup structure (a dedicated canonical encoding slice, not naive term printing). The encoding MUST preserve sharing, `SUP` labels, `DUP`/cell state, roots, and package-local references. External handles are epoch-bound membrane records, never raw serialized wires. The host effect layer MUST provide the required fsync/atomic-replace/lock semantics.

Gate: stop/restart/recover/replay produces identical state and receipts.

SLICE-090 — EXTERNAL API: DEPLOY, SEND, LISTEN

Goal: expose exactly three operations: `deploy`, `send`, `listen` (Rholang-shaped; see §11). Grant, invoke, update, revoke, snapshot, and rollback are contract-language patterns, not runtime ops.

Initial transport: a narrow framed protocol over TCP or Unix sockets; add HTTP and JSON only after the binary protocol is tested. Public TCP MUST use TLS; otherwise the first release is Unix-socket-only.

Gate: API tests cover authentication, port leakage, replay, malformed inputs, fuel limits, and concurrent requests.

SLICE-100 — INTERNAL CONTRACT SENDS

Goal: pass a port from contract A to contract B by message; B invokes by sending on it (Rholang: passing a name delegates authority).

Rules: sends are serialized or explicitly scheduled; reentrancy and failure semantics are specified before implementation; ports are never looked up by package name; reply wires are one-shot and cannot be replayed.

Gate: A can reach B only after receiving B's port; B cannot reach A unless A's port was passed; failed nested sends have defined rollback behavior.

SLICE-110 — SIGNED REDEPLOY AND MIGRATION

Goal: owner-only versioned replacement via redeploy, not in-place mutation.

Design: immutable package versions, expected-old-hash check, explicit migration message accepted only on the owner-held update wire, transactional state handoff, fresh update continuation wire, update receipt. Draft 0 policy: existing public invoke wires stay attached to the old version; callers move via an explicit forwarder package. No implicit forwarding root.

Gate: non-owner redeploy fails; stale redeploy fails; migration failure preserves old code/subgraph; old versions remain addressable if policy allows.

SLICE-120 — TRUSTED NATIVE ADAPTERS

Goal: add crypto, storage, clock, and transport effects without granting them to contracts implicitly.

Gate: an effect is callable only through an injected wire and is absent from pure package manifests. Trusted code lives behind a **syntactic** seam (`@runtime` -style annotation or a `host/` module user packages cannot import), so the boundary is visible to the checker, not just a comment. ABI compatibility is pinned to the Bend version. Crypto, TLS/Unix transport, fsync/atomic replace, entropy, and clock behavior are explicit adapters. Effects are not included in the pure confluence claim.

SLICE-130 — BEND-TO-HVM5 COMPILER SUBSET

Goal: deploy ordinary Bend 2 source instead of handwritten HVM5 source.

Order: pure values → ADTs → pattern matching → recursion → closures/affinity → arrays → typed ABI. Leave IO/effects for later.

Gate: differential tests compare the lowered package with the Bend reference implementation and HVM5 reference evaluator on generated programs.

SLICE-140 — HARDENING AND LINUX RELEASE

Goal: run as a non-root service with defense in depth.

Use a dedicated user, restrictive filesystem, resource limits, seccomp/container policy, locked-down network effects, structured audit logs, and key-free runtime deployment. These are defense in depth; the HVM5 capability model remains the semantic boundary.

Gate: fuzzing, fault injection, package-isolation tests, race tests, replay tests, and an independent review of the trusted effects/evaluator.

SLICE-150 — HVM5 WASM WORKER TARGET

Goal: compile the pure HVM5 runtime to WASM and load it in the Cloudflare Worker through `WebAssembly.instantiate` . The Worker uses the same Bend package/graph semantics as native BendRT; TypeScript is only the Alchemy, HTTP, and WASM membrane.

Build: use the Emscripten route from Bend PR #866 / `metal-bending` :

```
Bend C runtime → emcc -std=gnu11 -O3 → hvm5.wasm
```

Cloudflare mode MUST use a worker-safe, single-thread ABI: no `PROXY_TO_PTHREAD`, `SharedArrayBuffer`, `Window`, audio, sockets, or WASI filesystem. Nix provides the pinned Emscripten toolchain; Alchemy bundles the resulting `.wasm` module.

The browser-page target (`bend file.bend -o file.html`) is optional and may add:

- `PROXY_TO_PTHREAD` and one browser worker per core;
- COOP/COEP or `coi-serviceworker` headers;
- `Window` canvas and input effects;
- WebGPU as a separate runtime, not a Metal port.

The Cloudflare Worker target is accepted only when its `.wasm` module executes the HVM5 selftest and API flows; the JS kernel mirror is not an accepted substitute.

5. VERIFICATION STRATEGY

BEND LAWS AND PROOFS

Add laws for:

- parser round trips;
- package-local reference closure;
- manifest canonicalization;
- linear wires never cross a duplicator;
- subnet grafting preserves package-local wiring;
- capability facet non-escalation;
- no capability construction from ordinary data;
- fuel never becoming negative;
- over-budget reduction never committing;
- version/hash/nonce update rules;
- deterministic state transition.

These prove the pure model. They do not by themselves prove the C runtime, cryptographic library, filesystem, network, or kernel.

CONFLUENCE/DIAMOND CHECKS (TESTS + AUDITS, NOT EQUALITY LAWS)

- the same net reduced under different schedules converges (differential order tests on generated graphs);
- every calculus extension (native ops, fusion paths, effects) re-passes the diamond audit, or agreement is void for graphs using it;
- severed linear wires stay severed under all schedules (no residual copy becomes reachable by reordering reduction).

ADVERSARIAL TESTS

- fabricate every public constructor and numeric token;
- try cross-subnet indexes, names, and hashes;
- pass wires between subnets and test intended delegation only;
- exhaust fuel, heap, stack, input, output, and nested-send budgets;
- replay and reorder signed graft/replace requests;
- crash at every WAL boundary;
- attempt reentrancy and stale-state updates;
- fuzz parser, manifest, signatures, and serialized graphs;
- compare HVM5 output with a reference implementation under varied schedules.

6. DECISIONS REQUIRED BEFORE IMPLEMENTATION

The normative Draft-0 decisions are in the specification §15. The following items remain explicit review decisions rather than hidden assumptions:

1. Is the first package format HVM5 source, HVM5 `Tpl` data, or both?
2. Which exact HVM5 rule-table revision is authoritative?
3. What is the final live-net graft representation after books-first mode?

4. Which wire payloads are allowed to contain `SharedWire` and `LinearWire`? **Decided (normative):** user HVM5 graphs MUST NOT construct, inspect, or serialize wires at all. The runtime represents wires as `law` handles (the `File` / `Socket` idiom), never `Data`. Linearity is affinity in the checker plus a runtime gate at graft/message boundaries — both, not either. Graphs are homoiconic; capabilities are extra-graph and cross the membrane only as epoch-bound records.
5. What is the serialized reply/queue/savepoint representation?
6. What is the state retention policy for replaced subgraphs?
7. Which host effects are admitted in the first pure-contract release?
8. What graph-size/fuel limits are required for the target workload?
9. Is the first deployment single-node only?

Recommended answers for the first vertical slice:

```
HVM5 source; Ed25519 verification; opaque bearer invocation wires;
pure-only contracts; serialized synchronous sends; explicit persistent
subgraphs; immutable version history; public code; interaction-count fuel;
single Linux node; books-first with a graft interface; update wires linear,
invoke wires shared unless revocation is required.
```

7. IMMEDIATE NEXT SLICE

Implement only `SLICE-000`, `SLICE-005`, `SLICE-010`, and `SLICE-015` first. Do not add HTTP, crypto, storage, or external handles until the exact rule table, the pure `reduce(OpenSubnet, root, fuel, limits)` seam, and the first-class net kernel seam exist.

The first useful milestone is therefore **not** "deploy Bend source." It is:

```
reduce(open_subnet, root_wire, fuel, limits) -> structured result
```

Once that seam is stable, deployment, capabilities, signatures, persistence, and the Bend-to-HVM5 compiler can be added as independently gated slices.

8. COMPARATIVE RESEARCH (REFERENCE APPENDIX — NOT ON THE CRITICAL PATH)

Sections 8–10 are research context for reviewers. Nothing in them gates SLICE-000 through SLICE-015; the normative path is spec plus §§2–7 and §13.

This section was added before implementation. Claims are tagged:

- **[Established]** directly stated or implemented by the cited project;
- **[Consensus]** a broadly accepted architectural interpretation;
- **[Design]** a proposed adaptation for this runtime, not a claim about the cited project.

SOURCES INSPECTED

- Bend: `bendlang/bend` (https://github.com/bendlang/bend), especially `demos/pure_hvm5_mini/main.bend` (https://github.com/bendlang/bend/blob/main/demos/pure_hvm5_mini/main.bend), `bend2/comp.ts`, `bend2/main.ts`, and `guide/`.
- Golem: task model (https://docs.golem.network/docs/creators/javascript/task-executor/task-model) and the historical `golemfactory/golem` README (https://github.com/golemfactory/golem).
- Unison: `unisonweb/unison` (https://github.com/unisonweb/unison) and The Big Idea (https://www.unison-lang.org/docs/the-big-idea/).
- Nix: `NixOS/nix` (https://github.com/NixOS/nix) and Nix flakes (https://nix.dev/concepts/flakes.html).
- Tezos: `tezos/tezos` (https://github.com/tezos/tezos), Michelson documentation (https://docs.tezos.com/smart-contracts/languages/michelson), and smart-contract documentation (https://docs.tezos.com/smart-contracts).
- Cairo: `starkware-libs/cairo` (https://github.com/starkware-libs/cairo).
- RISC Zero: `risc0/risc0` (https://github.com/risc0/risc0) and terminology (https://dev.risczero.com/terminology).
- Mirage/OCaml: `mirage/mirage` (https://github.com/mirage/mirage) and `mirage/irmin` (https://github.com/mirage/irmin).
- Bendoom: `eliesgalvira/bendoom` (https://github.com/eliesgalvira/bendoom) — largest verified Bend program; method reference (dual-backend tests, oracle discipline, law organization, reproducible native builds). Verified: no browser/WASM target exists there; JS lane is headless-logic only.
- Browser WASM reference: `AdrielSantana/metal-bending` (https://github.com/AdrielSantana/metal-bending), plus Bend PR #866 (https://github.com/bendlang/bend/pull/866). This is the actual WASM path: Bend C runtime → Emscripten (`-pthread -mtail-call`) → `.wasm`/JS/page, with browser workers and canvas. It is not Bendoom and it is not a Cloudflare Worker

target without removing browser/pthread/window APIs.

- RChain: `rchain/rchain` (https://github.com/rchain/rchain) and the primary docs at rholang.org (https://rholang.org/docs/rholang/): deploy/propose (deploy (https://rholang.org/docs/deploy-contract/), propose (https://rholang.org/docs/propose-block/)), unforgeable names (names (https://rholang.org/docs/names-processes/)), send/receive/contracts (send-receive (https://rholang.org/docs/send-receive/)), and cost accounting (https://rholang.org/docs/rholang-cost-accounting).

The projects solve different problems. Golem is a resource marketplace and task orchestrator; Nix is reproducible deployment; Unison is content-addressed code and typed durable computation; Tezos and Cairo are consensus/provability systems; RISC Zero proves execution; Mirage/Irmin combine a small deployment target with portable versioned state. None is a drop-in ocap HVM5 runtime.

9. SEVEN-LEVEL COMPARISON

RCHAIN/RHOLANG — THE CLOSEST MODEL, READ FIRST

1. **Principles:** all computation is message passing; names are channels; `new`-created names are private and **unforgeable** (not constructible from bits); passing a name delegates authority; no return, only sends plus reply channels. **[Established]**
2. **Models:** process, name/channel, tuplespace (resting messages), `contract` (= replicated receive), deploy, block, phlo.
3. **Methods:** write processes; `deploy` signed source with phlo limit/price; validator `propose`s it into a block; `send` on a channel; `for` / `contract` to receive persistently; read via `data-at-name`.
4. **Rules:** only a holder of a name can send/receive on it; deploys carry a key, phlo limit, and validity window; unused phlo is refunded; pattern-match cost is bounded by restricting the match language.
5. **Artifacts:** `.rho` source, signed deploy, `DeployId`, block, receipt, tuplespace contents.
6. **Components:** channel name, message, reply channel, phlo limit/price, private key, valid-after height, deploy ID.
7. **Instance:** `contract Counter(...)` deployed once; callers send messages plus a reply channel; state is the resting message the contract holds.

Take: the whole API philosophy — deploy + send + listen is sufficient; grant/invoke/update/revoke are language patterns, not runtime ops. Unforgeable names are the capability model to copy. Phlo limit/price/valid-after maps directly onto our fuel limit/price/replay guard. **Do not take:** blockchain consensus itself (Casper, validators, REV) — out of scope for the first release.

BELOW RHOLANG: THE NET ITSELF — THE INET-NATIVE SUBSTRATE

Rholang names/channels/tuplespace are still language-level machinery *around* the primitive. One level down, the interaction net already is the mechanism, and five more things dissolve:

1. **Wires, not names.** A Rholang name is resolved, looked up, Turf-stored. A wire needs none of that: connection *is* authority, physically. No registry, no lookup, no ambient namespace to secure — there is nowhere to look anything up. Deploy hands back a wire; send plugs into it.
2. **Active pairs, not entrypoints.** Invocation needs no dispatch table, ABI, or selector. Plugging a wire whose other end faces an agent's principal port *creates a redex*; reduction is invocation. Entry points are a consequence of graph shape, not a separate concept.
3. **Linearity, not proxies, for revocation.** Rholang cannot revoke a shared name — this is exactly why the plan fell back to proxy patterns. But a *linear* wire cannot be duplicated: the type discipline forbids it from passing through a duplicator. Severing a linear wire therefore revokes completely and provably — strictly stronger than Rholang, and native to affine/linear IC. **[Design — requires linearity enforcement on cap wires.]**
4. **Confluence, not replay protocol, for agreement.** Lafont's interaction combinators have the one-step diamond property: any reduction order reaches the same result. Validators replaying the same net agree *structurally*, without agreeing on scheduling first. RChain must engineer replay agreement; a confluent substrate gets it as a theorem — provided each extension (native numbers, dup labels, fusion, effects) is audited to preserve the diamond.
5. **Graph = code = state = snapshot (homoiconicity).** Deploy (graft a subgraph), persist (serialize it), migrate (graft its successor), and snapshot (copy it) are one operation on one kind of thing. No separate code/state/snapshot subsystems — RSpace-style tuplespace included, since a resting message is just a subgraph sitting on a wire.
6. **Interaction count is not a meter bolted on — it is the computation.** Each annihilation/commutation/erasure is one countable step. The `c` counter already threaded through `pure_hvm5_mini` is fuel in its native form; fuel exhaustion is "stop reducing," not a separate accounting layer.

What stays *outside* the net (host-level, still required): signature verification on deploy envelopes, mediation of the graft operation itself (who may extend the live net), admission bounds on graph size, the serialization format, and effect membranes. The net gives isolation, cost, and agreement; the host gives identity, admission, and the outside world.

Caveats, stated plainly: the current `pure_hvm5_mini` evaluator is a *sequential simulation* of IC semantics (whnf loop), not a parallel inet reducer — true concurrent active-pair reduction is HVM-lineage work, not this demo. Linearity on cap wires needs type-level support that public Bend `Data` constructors do not currently provide; it is a design requirement on the HVM5 level, not a property the demo has. And every calculus extension must preserve confluence or item 4 silently stops holding.

HVM5/BEND TARGET

1. **Principles:** interaction locality, affine/linear resource discipline, reduction as computation, reduction count as a natural cost signal, and wires/ports as the proposed authority boundary. **[Consensus/Design]**
2. **Models:** live net, grafted subnet (book as interim), template, cell, wire/port with linearity flag, active pair, interaction budget, serialized subgraph.
3. **Methods:** parse, hash, admit by signature, graft a private subnet, connect a wire (active pair), reduce to budget, sever to revoke, serialize the subgraph, and emit a receipt.
4. **Rules:** no lookup (nothing to look up), no forged or duplicated-linear wires, bounded reductions, no unapproved effects, subnet-local refs, diamond preserved, atomic snapshot commit.
5. **Artifacts:** HVM5 book/package, signed manifest, capability permit, transaction receipt, state snapshot, optional proof receipt.
6. **Components:** package hash, compiler/runtime version, entry ports, ABI, imports, wire shapes, fuel limit, state hash, nonce, signature, output.
7. **Instance:** one deployed counter graph with an owner update facet and a delegated invoke facet.

GOLEM NETWORK

1. **Principles:** rent distributed resources, split work into tasks, separate requestors from providers, and settle resource use economically. **[Established]**
2. **Models:** Yagna service, provider, requestor, job, task, worker interface, Docker-like image, proposal, activity, payment.
3. **Methods:** define requirements, select proposals, start an activity, upload inputs, run commands, download outputs, retry, and pay.
4. **Rules:** provider requirements and filters, image/volume contract, task boundaries, activity/payment lifecycle, provider selection policy.
5. **Artifacts:** image, task batch, command result, activity record, payment record, uploaded/downloaded data.
6. **Components:** command, volume, input/output, provider constraints, concurrency, activity ID, result status, payment token.
7. **Instance:** a requestor submits one image and a batch of independent jobs to providers.

Take: scheduler, activity lifecycle, provider/resource constraints, retry and receipt concepts. **Do not take as the contract security model:** Docker-like images and provider trust are not graph ocap isolation.

UNISON

1. **Principles:** immutable content-addressed definitions, names as metadata, typed code/data, and effects separated by abilities. **[Established]**
2. **Models:** codebase database, definition hash, name mapping, ability, typed durable value, project, deployment.
3. **Methods:** parse/typecheck/hash, cache, sync missing definitions, update names without mutating old definitions, persist values, deploy functions.
4. **Rules:** a definition hash identifies exact syntax/dependencies; code at a hash does not change; abilities control effect interpretation.
5. **Artifacts:** codebase entry, hash-addressed dependency tree, typed stored value, service/batch deployment.
6. **Components:** syntax tree, positional variables, dependency hashes, inferred type, ability set, name metadata, storage reference.
7. **Instance:** a function hash is synced to a host with missing dependencies, then invoked by a typed name/entrypoint.

Take: content-addressed package identity, dependency closure, immutable versions, typed storage, and a separate mutable name/registry layer. Unison names/hashes provide integrity and transport, not authority; our capability layer must remain separate.

NIX/NIXOS

1. **Principles:** purity, reproducibility, explicit dependencies, isolation of build inputs, and substitutable results. **[Established]**
2. **Models:** derivation, store path, closure, flake input/output, lock file, substitution/cache, system configuration.

3. **Methods:** evaluate, pin inputs, build in a controlled environment, hash dependencies, substitute a cached result, activate a configuration, roll back.
4. **Rules:** declared inputs, locked versions, reproducible build assumptions, no undeclared dependency, explicit system target. Flakes are still documented as experimental, so we should copy the idea rather than depend on the format.
5. **Artifacts:** store closure, binary/cache object, lock file, system generation, activation profile.
6. **Components:** source hash, dependency closure, platform, builder, outputs, environment, cache signature, generation number.
7. **Instance:** a locked package closure is built once and reused by hash on a Linux target.

Take: reproducible runtime image, lock file, signed content-addressed cache, generation/rollback, and explicit platform placement. Nix does not provide per-object runtime capabilities; it protects deployment/build provenance.

TEZOS/MICHELSON

1. **Principles:** deterministic replicated state transition, explicit storage, typed execution, bounded resource consumption, and formal verifiability. **[Established]**
2. **Models:** contract parameter, storage, entrypoint, stack, operation, address, amount, big-map, block context, gas.
3. **Methods:** originate, invoke an entrypoint, rewrite a typed stack, emit operations, serialize the new storage, charge gas, and commit a block state.
4. **Rules:** typed stack transitions; restricted access to own storage, call parameter, and specified context; gas limit; deterministic operation rules.
5. **Artifacts:** Michelson/Micheline contract, origination operation, call operation, storage value, operation receipt, proof/specification.
6. **Components:** parameter sum/entrypoints, storage type/value, amount, sender/source, calldata, gas limit/used, emitted operations, result.
7. **Instance:** one originated contract with `Increment` and `Decrement` entrypoints and serialized integer storage.

Take: the explicit `parameter/storage/entrypoint` ABI, pure state rewrite, gas-before-execution model, emitted-operation result, and contract-level views. This is the closest API shape to our first contract runtime. Tezos addresses and signatures are not automatically object capabilities.

CAIRO/STARKNET-STYLE PROVABLE EXECUTION

1. **Principles:** compile high-level code to a constrained intermediate form, execute deterministically, and prove the execution trace. **[Established]**
2. **Models:** Cairo source, Sierra, CASM, contract class, calldata, state diff, trace, proof, verifier.
3. **Methods:** compile to Sierra, compile to CASM, validate, execute, prove, verify, and apply state changes.
4. **Rules:** valid typed/verified intermediate representation, deterministic execution, declared program identity, resource/gas limits, verifier checks.
5. **Artifacts:** contract class, compiled class, execution trace/commitment, proof, public output, state update.
6. **Components:** program hash, entrypoint selector, calldata, memory/trace, public inputs/outputs, gas, proof commitment, state root.
7. **Instance:** a contract call whose compiled program and state transition can be verified independently of the executor.

Take: stable verified IR between source and runtime, program identity, and a future proof artifact for contract execution. Do not make zero-knowledge proving a prerequisite for local ocap execution; add it as an optional receipt backend after the deterministic state model is stable.

RISC ZERO

1. **Principles:** prove execution of a known program while exposing only the committed public output; separate host from guest. **[Established]**
2. **Models:** guest method, host, ELF, image ID, execution trace, journal, receipt claim, seal, composition, recursion.
3. **Methods:** compile a method, execute, generate a receipt, verify the image ID and seal, compose receipts, and reveal journal data.
4. **Rules:** the expected image ID/control root must match; the host cannot alter a proven guest execution; only committed journal data is public.
5. **Artifacts:** deterministic guest image, receipt, claim, journal, seal, composite receipt.
6. **Components:** program identity, input commitment, execution status, public output, memory claim, segment/continuation data, proof seal.

7. **Instance:** a private computation produces a receipt proving a public state transition.

Take: program identity, host/guest split, public journal, receipt schema, segmented execution, and recursive composition. This becomes an optional `ProofReceipt` for our contract receipt; it does not replace capabilities, persistence, or deployment authorization.

MIRAGEOS/IRMIN/OCAML

1. **Principles:** minimize the trusted operating environment, assemble only the components an application needs, and make state portable/versioned. **[Established]**
2. **Models:** functorized device/component graph, unikernel, target backend, Irmin repository, branch, commit, merge, snapshot.
3. **Methods:** describe components, configure, generate metadata, build a specialized unikernel, deploy under a hypervisor, snapshot, branch, merge, synchronize, and restore.
4. **Rules:** explicit component interfaces, no accidental full host OS, chosen backend/device set, store-specific consistency/merge policy.
5. **Artifacts:** unikernel image, configuration, store commit, branch/snapshot, deployment record.
6. **Components:** network/block/random devices, functors, key/value paths, contents, commit metadata, branch heads, merge policy.
7. **Instance:** a network unikernel backed by a branchable Irmin store that can be snapshotted and restored.

Take: explicit host boundary, minimal trusted components, immutable/branchable state, snapshots, and merge policies. MirageOS is a deployment/isolation boundary rather than an in-process ocap evaluator; Irmin's branching model is useful for contract state and rollback.

BENDOOM — THE CLOSEST ENGINEERING PRECEDENT, READ FOR METHOD NOT RUNTIME

Bendoom (Doom E1M1 in Bend 2, `eliesgalvira/bendoom`) is the largest verified Bend program in the wild: ~150 laws, dual-backend tests, Nix-reproducible native builds. Correction on first impression: it does **not** run as WASM in a browser — exhaustive search finds zero `.html/.js/.ts/.wasm` files. It runs native C/X11/ALSA; the JS lane (`bend -o .js` + bun) is used only for headless logic tests. That absence validates our SLICE-150 deferral more than any argument could: even the flagship game ships no browser target.

METAL BENDING — THE ACTUAL WASM PRECEDENT

`AdrielSantana/metal-bending` and Bend PR #866 are the source of truth for the WASM build, not Bendoom. The PR adds `bend file.bend -o file.html`:

```
Bend C runtime → emcc -std=gnu11 -O3 -pthread -mtail-call
  → file.html + file.js + file.wasm
```

The browser page uses `SharedArrayBuffer`, a Web Worker per core, Emscripten Atomics, tail calls, COOP/COEP headers, and the Bend `Window` effect. It keeps the same C runtime semantics and measures close to native. This is browser WASM, not JavaScript execution.

Cloudflare adaptation: Cloudflare Workers can import a `.wasm` module and instantiate it, but they are not browser pages: no DOM, `Window`, browser workers, or COOP/COEP pthread setup. Our Worker target therefore uses a single-thread Emscripten/WASM build with a small exported ABI and TypeScript only as the Alchemy/HTTP/membrane host. Browser `-pthread` mode remains a separate target.

1. **Principles:** the oracle (vanilla Doom/Chocolate Doom) is ground truth; expected values are computed *outside* the code under test; pure simulation plus a thin IO shell; native and JS lanes must agree byte-for-byte; everything pinned (bend-src tarball, flake.lock, WAD hashes). **[Established]**
2. **Models:** pure modules, `#|` -pinned stdout tests, LAWS (human states) + PROOF (AI fills), WAD/oracle fixtures, digest-keyed release cache.
3. **Methods:** write pure logic; pin expected outputs from the oracle; run both backends and diff; state closed sanity samples plus universal quantified laws; tag releases and publish checksummed bundles.
4. **Rules:** automated checks never open windows (headless Xvfb); frame hashes regenerate only after oracle-zero; checker constraints documented (`docs/bend.md`: helpers-before-callers, no mutual recursion, scrutinee order, Data-only lists, 255-width cap, `+` everywhere); tickets deleted when done.
5. **Artifacts:** pinned test outputs, frame hashes, portable tarballs + SHA256SUMS, proof log (`All terms check.`).
6. **Components:** test `.bend` mains, `#|` pins, oracle scripts, flake modules, digest-keyed launcher cache.
7. **Instance:** one test file passing both lanes plus oracle-zero frames.

Take (adopted): dual-backend print-and-diff tests — `net/test/self.bend` prints 15 PASS lines, verified byte-identical on native, bun, and node; closed-sanity plus universal law split for future LAWS work; oracle-first expected values (our oracle: HVM5 reference behavior + Rholang doc examples); digest-keyed versioned releases for the future native service; the checker-constraint catalog as the reference behind our G-08 finding. **Do not take:** rebuild-and-retag deployment (we need runtime grafting, which is strictly harder); browser pthread/window assumptions in the Cloudflare target. **Do take:** Bendoom's Nix pinning and Metal Bending's Emscripten compiler path.

10. SYNTHESIS: WHAT BELONGS IN OUR DESIGN

Requirement	Borrowed idea	Runtime decision
code identity	Unison hashes; Nix store paths; RISC Zero image IDs	`PackageHash` includes canonical HVM5 book, manifest, compiler/runtime version
dependency closure	Unison codebase; Nix closure	package contains all code refs or explicit immutable dependencies
authority	HVM5 wires (linear severable); explicit host/guest membrane	opaque wires, never hashes or integer refs
cost	HVM5 interaction count natively; Tezos gas; zkVM cycles	fuel = stop reducing (pure net only; host-effect charging comes after SLICE-120 and stays out of the diamond claim)
entry ABI	Rholang contract shape; Tezos parameter/storage/entrypoints	persistent receive on export wires; explicit subgraph transition
state	subgraph snapshots; Tezos storage; Irmin branches	immutable subgraph roots, WAL, optional branches
deploy	RChain deploy; Golem image/activity; Nix generation	signed package/instance manifest and versioned activation
execution proof	Bend laws; Tezos formal specs; RISC Zero receipts	pure Bend laws first; optional HVM5 execution receipt later
scheduling	Golem tasks/providers	later multi-worker scheduler; first release serialized per instance
rollback	Nix generations; Irmin branches	previous code/subgraph roots retained and addressable
effects	Unison abilities; Mirage explicit devices	effect manifest + injected capability + narrow Bend foreign
distribution	Golem requestor/provider; Unison dependency sync	later package/cache service; no consensus in first release

11. DOMAIN TYPE AND API COMPARISON

The following are the names to use in the Bend design. They are deliberately separate: a hash identifies content, an identity authorizes actions, and a capability grants runtime authority.

Domain type	Meaning	Analogues	First API surface
`PackageHash`	immutable code/manifest identity	Unison hash, Nix store path, RISC Zero image ID	— (fetch by hash is not first release; deploy uploads bytes)
`Identity`	public-key principal	Tezos source/signer, Golem requestor	signature verification
`Package`	closed HVM5 book + ABI + effect manifest	Cairo class, Golem image, Unison closure	`deploy(package, signature)`
`InstanceId`	one deployed stateful instance	Tezos address, Nix generation	created by `DEPLOY`; no separate `instance.create` op
`StateRoot`	immutable state snapshot	Tezos storage, Irmin commit	internal only; exposed via `LISTEN`, never `state.load/commit`

Domain type	Meaning	Analogues	First API surface
`Entrypoint`	typed persistent receive on an export wire	Rholang `contract`, Tezos entrypoint, RISC Zero method	`send(port, msg + reply_port)`
`CapRef`	unforgeable wire (linear flag); possession is authority	Rholang name, inet wire	passed as a message, never constructed
`Facet`	dissolved: a facet is a separate wire	— (no runtime facet type)	distinct invoke vs update wires
`Fuel`	interaction budget	Rholang phlo, Tezos gas, HVM5 interactions, zkVM cycles	`limits.fuel` + price
`Receipt`	authenticated execution result	RChain DeployId/receipt, Tezos operation receipt, RISC Zero receipt	`send -> receipt`
`Branch`	alternate subgraph/history	Irmin branch, Nix generation	copy-on-graft/serialize
`Effect`	host action outside pure graph	Unison ability, Mirage device	`effect(cap, request)`

PROPOSED API — RHOLANG-STYLE: DEPLOY + SEND + LISTEN

The previous draft had seven runtime operations (`deploy/invoke/grant/update/revoke/snapshot/rollback`). Per the RChain model below, that is overcomplicated. The runtime needs **three** operations; everything else is a pattern in the contract language itself.

RChain reference (primary docs at rholang.org):

- A deploy is one signed envelope: Rholang source plus `--phlo-limit`, `--phlo-price`, `--private-key`, and `--valid-after-block-number`. It returns a `DeployId`; a validator later bundles it into a block (`propose`). Reads are `data-at-name` / `listenForDataAtName` on a channel.
- Rholang has two kinds of values: **processes and names**. A name is a channel. `new x in {...}` creates a private, **unforgeable** name: it cannot be constructed from bits, only received from someone who holds it. Passing a name delegates authority — that *is* the grant operation.
- There is **no return**: `contract C(args) = {...}` is sugar for a replicated receive (`for(args <= C)`). Callers send a message and a reply channel; the contract sends the answer back. Invocation *is* sending.
- State *is* messages resting in the tuplespace until a `for` consumes them. No snapshot/rollback op: versioning is just new messages.
- Update is redeploy + explicit migration, authenticated by the deployer's key. Revocation of an already-shared name is impossible by construction — exactly like object capabilities — so revocability is built as a proxy/forwarder pattern *in user code*, not as a runtime op.
- Cost accounting charges per reduction/match (`phlo`), with refunds for unused budget — the same shape as interaction-count fuel.

Mapping to HVM5 nets (one-to-one):

Rholang	HVM5 runtime
unforgeable name/channel	wire/port into a private subnet
`new x`	fresh port minted at deploy
send `!(...)`	invoke = deliver a message on a port
`for(... <- x)` / `contract`	persistent entry loop on an export port
passing a name	grant = send the port to another package
redeploy + migrate	update = new deploy + explicit state handoff
`data-at-name`	read resting messages/state on a port
phlo limit/price	interaction fuel limit (+ price when multi-tenant)

The runtime API is therefore:

```
deploy(
  package_bytes,     # HVM5 source or book
  manifest,          # exports, effect allowlist, ABI
  owner_signature,   # private-key auth; key never leaves the client
  fuel_limit,        # interactions
  fuel_price,        # settlement price when multi-tenant
  valid_after        # sequence/height guard against replay + stale deploy
) -> DeployReceipt { package_hash, deploy_id, exported_wires }

send(
  wire,              # the capability itself — possession is authority
  message,           # input + reply wire for the answer (no built-in return)
  fuel_limit
) -> SendReceipt { deploy_id, fuel_used, status }

listen(wire, depth) -> normal form / attached subgraph readable at that root
(depth bounds the walk; this is not a tuplespace query language)
```

Deliberately **not** runtime operations:

- `grant` → send the port to another package. Nothing to implement.
- `invoke` → `send` on the entry port with a reply port. Nothing separate.
- `update` → `deploy` of a new version + a migration message the old code accepts only on its owner-held update port. Owner check = signature on the deploy envelope plus possession of that port.
- `revoke` → proxy/forwarder pattern in contract code (a port that consults a live flag before forwarding). The runtime only needs to support the pattern, not implement it.
- `snapshot` / `rollback` → state is messages; keep old messages and you have history. The WAL already gives crash recovery.
- `Facet` as a runtime type → dissolved: a facet *is* a separate port. An invoke port and an update port are different unforgeable names; no facet-checking logic in the broker.

The external representation of a port is a high-entropy bearer token or a connection-bound handle. It must not be a package name, public hash, array index, URL query parameter, or user-constructible HVM5 constructor. The internal representation is a runtime port/table entry. A signed deploy proves who may create a package; it does not replace the port.

12. PROOF AND LAW PLAN AFTER COMPARISON

The existing Bend `LAWS.bend` / `PROOF.bend` gate proves the small demonstration program. It does **not** yet prove HVM5 isolation or cryptographic security. The new implementation must keep these proof classes separate:

PURE EVALUATOR LAWS

- parser/source round-trip;
- template/book loading preserves definitions;
- reduction result agrees with the reference evaluator;
- interaction counter increments exactly once per defined reduction;
- fuel decreases monotonically and cannot wrap;
- out-of-fuel execution has no committed state;
- package-local reference closure.

CAPABILITY-MODEL LAWS

- an invoke port cannot authorize update;
- a port/target mismatch is rejected;
- a killed proxy rejects future sends;
- receiving a port never grants a different port;
- a port is never produced from ordinary package data;
- a package cannot enumerate another package's book;
- internal sends require an explicitly passed port.

TRANSACTION/STATE LAWS

- successful send produces exactly one new subgraph root;
- failed send preserves the old root;

- update requires the expected old hash and fresh nonce;
- migration is atomic;
- receipt fields match the committed transition;
- replayed requests do not apply twice.

TRUSTED-BOUNDARY TESTS, NOT BEND PROOFS

Crypto, C effects, file durability, TCP framing, native memory safety, and Linux permissions cannot be claimed proven merely because `PROOF.bend says All terms check.` They require golden vectors, fault injection, fuzzing, source review, and defense-in-depth. If later desired, the pure capability machine can be formalized in Lean and the HVM5 execution trace can receive a RISC Zero-like receipt.

13. REVISED IMPLEMENTATION ORDER

The comparison changes the order from "add an API early" to "freeze the rule table and net seam before building the service":

1. **SLICE-000:** freeze the HVM5 demo and proof baseline.
2. **SLICE-005:** freeze the exact `SUP` / `DUP` / `ERA` and optimization rule table.
3. **SLICE-010:** expose the pure `OpenSubnet` / `reduce` seam without IO.
4. **SLICE-015:** define the first-class agent/port/wire kernel seam.
5. **SLICE-020:** native fuel (stop reducing) with checked interaction and memory limits.
6. **SLICE-030:** private package books plus the true graft interface.
7. **SLICE-040:** wire primitive plus linearity; reject forgeable representations.
8. **SLICE-050:** persistent receive, shared invoke wires, linear update/reply wires.
9. **SLICE-070:** sends as bounded transactions (active pair + snapshot).
10. **SLICE-080:** canonical graph persistence + membrane WAL.
11. **SLICE-120:** trusted adapters first (crypto, storage, clock, transport — SLICE-060 depends on the crypto adapter).
12. **SLICE-060:** signed graft envelopes and replay protection.
13. **SLICE-090:** local framed API (`deploy/send/listen`), then HTTP adapter.
14. **SLICE-100:** internal wire-passing sends and deterministic scheduling.
15. **SLICE-110:** immutable versions and explicit migration sends.
16. **SLICE-130:** Bend-to-HVM5 compiler subset.
17. **SLICE-140:** Linux hardening and release gates (OS boundary primary until the evaluator is verified; per-extension diamond audits).
18. **SLICE-150:** Emscripten/WASM Worker target; browser-page pthread mode is optional and separate.

No implementation slice should begin until the corresponding laws, test cases, owner, placement, budget, rollback, and gate are recorded in `lattice/` .

14. NATIVE + WASM ACCEPTANCE TARGETS

The first demonstrable system is:

```
Linux client
      signed deploy/invoke/update requests
    ▼
native BendRT executable
    ├─ HVM5 net reducer written in Bend 2 (one live net)
    ├─ grafted private subnets (books first)
    ├─ wire membrane with linear severance
    ├─ native fuel (stop reducing) plus graph-size admission
    ├─ subgraph snapshots + WAL
    └─ narrow audited Linux effects
```

The Cloudflare target is the same pure runtime compiled through the Emscripten architecture from PR #866, without browser page glue:

```
Nix-pinned Bend web compiler + emcc
  → hvm5-worker.wasm
  → Cloudflare Worker WebAssembly.instantiate
  → TypeScript membrane / deploy-send-listen adapter
```

The Worker MUST NOT run the TypeScript kernel mirror as its contract engine. That mirror remains a parity/reference test only. The `.wasm` module MUST own rule reduction, grafting, wires, fuel, and contract state. TypeScript owns only HTTP, external handle epochs, Cloudflare storage bindings, and Alchemy deploy.

The first release is successful only when both native and Worker targets can deploy two packages, prevent cross-book/capability access, invoke them externally and internally, reject unauthorized updates, recover state after restart, and pass the same pure conformance vectors. Browser-page support is not a release criterion; Cloudflare WASM support is.

G.36 · `bend-runtime/hvm5/version1/plan/README.md`

HVM5 v1 plan

- `00-plan.md` — mission, phases, critical path, budgets.
- `01-slices.md` — slice table with dependencies and gates.
- `02-gates.md` — exact commands and expected outputs.
- `03-risks.md` — gaps, decisions, and residual risks.
- `legacy/` — the Draft 0 plan, kept for history.

The plan is subordinate to `../spec/`. If they conflict, fix the plan.

G.37 · `bend-runtime/hvm5/version1/bench/results/w3-2026-10-20.md`

W3 — repeated public sends

Date: 2026-10-20 Compiler: bend bend 2.0.25 Host: Linux 6.18.33.2-microsoft-standard-WSL2 x86_64

```
BENCH sends=100 ms=21
STATE Counter{100}
```

G.38 · `bend-runtime/hvm5/version1/compiler/README.md`

Compiler

The target is a self-hosted Bend 2 → inet compiler. **Deploy invokes it automatically; it is not a user-facing emit option.**

```
src/bendc.bend       Bend 2 source
  -> bend bendc.bend -o bendc
  -> deploy(contract.bend) -> compile_for_deploy -> contract.inet
```

`compiler/bootstrap/compile.ts` is **bootstrap only**. It exists to freeze the output format and test the runtime before the self-hosted stages land. It is not the final compiler and must not be presented as one.

Stages:

Stage	File	Status
A	`bootstrap/compile.ts`	implemented for the counter subset
B	`tools/emit-inet.ts` official checked-AST backend	working for the first-order contract fragment
B2	`src/bendc.bend` lowering/serializer from core IR	source parser for the v1 counter shape (bootstrap)

Stage	File	Status
C	`src/bendc.bend` general lowering	must consume the official checked core IR; do not reimplement the Bend 2 frontend
D	conformance corpus vs BendRT	planned

The compiler output is a postfix inet package. See `../spec/01b-contract-compiler.md` and `../spec/01d-self-hosted-compiler.md`.

Do not add a second runtime here. The compiler is build-time; the reducer is `../src/hvm5/eval.bend` and is compiled by BendRT.

G.39 · `bend-runtime/hvm5/version1/contracts/README.md`

Contracts

A contract is written in Bend 2 and compiled to an HVM5 inet package at deploy time.

```
counter.bend
  -> Bend 2 frontend (parse + check)
  -> core IR
  -> HVM5 inet term (Lam/App/Dup/Sup/Ctr/Mat/Num/Op)
  -> package hash
  -> signed deploy
  -> graft into a subspace
  -> reducer executes the inet
```

The source is **not** interpreted by the runtime. The runtime never parses `counter.bend` at send time. It receives the compiled inet package.

PACKAGE SHAPE

```
ContractPackage {
  format: "hvm5.package.v1",
  name, version, space,
  sourceHash,
  inetCode,          # the lowered transition term
  initCode,          # the lowered init term
  stateType, msgType,
  exports: [invoke, update],
  laws
}
```

V1 TARGET SUBSET

The first compiler slice covers:

- `type ... is Data` with first-order fields;
- `def` with parameters and a return type;
- `match` on constructors;
- constructor values;
- `Nat` literals and `Nat.add`, `Nat.sub`, `Nat.is_le`;
- `let` of a value;
- no effects, clocks, randomness, or foreign calls in `transition`.

The compiler inserts `Era` for unused binders and `Dup` for reused binders, so the lowered term preserves Bend 2's affinity.

COUNTER AND TALLY

`tally.bend` is the same contract shape with different names. It proves the compiler reads names from the checked source instead of assuming `Counter/value/by`.

COUNTER

`counter.bend` is the smallest complete contract. Its lowered shape is:

```
transition = λs. λm.
  Match(s, "Counter", ["value"],
    Match(m, "Inc", ["by"],
      Ctr("Counter", [Add(Var "value", Var "by")]),
      Match(m, "Reset", [],
        Ctr("Counter", [Num 0]),
        Stuck)),
    Stuck)
```

This is the exact term the inet reducer consumes.

G.40 · bend-runtime/hvm5/version1/src/README.md

`src` layout

This directory contains only files that exist and are used. There are no placeholder `model/` or `runtime/` directories.

ACTUAL FILES

```
src/
├─ README.md
├─ bendc.bend Bend 2 source parser, package model, deploy compile, digest
├─ deploy.bend deploy entrypoint; compiles source automatically
├─ sim.bend consolidated ten-node system/space/instance/fork model
└─ hvm5/
   ├─ term.bend homoiconic Term/Def/Program data
   ├─ parse.bend postfix inet package parser
   ├─ eval.bend CEK-style inet reducer
   ├─ show.bend canonical term display
   ├─ hash.bend structural digest
   ├─ main.bend small reducer entrypoint
   └─ run.bend package runner
```

WHY `model/` AND `runtime/` ARE ABSENT

The v1 reference slice consolidated the contract model and the ten-node model into `src/sim.bend`; the deploy compiler and package model live in `src/bendc.bend`. Empty `model/` and `runtime/` directories were removed because they advertised a module split that did not exist.

A future extraction into `model/` and `runtime/` is allowed, but it must be a real move: create the files, update imports and tests, and delete the consolidated code. Do not add empty placeholder directories.

DEPENDENCY DIRECTION

```
official Bend 2 frontend -> tools/emit-inet.ts -> inet package
src/bendc.bend (bootstrap) --------------------> inet package
inet package -> hvm5/parse -> hvm5/eval
hvm5/term -> hvm5/eval
hvm5/term -> hvm5/show -> hvm5/hash
src/sim.bend -> hvm5/* + bendc.bend
src/deploy.bend -> sim.bend + bendc.bend
tests/* -> src/*
bench/bench.bend -> sim.bend + hvm5/term
PROOF.bend -> sim.bend + LAWS.bend
```

Import rules:

1. `hvm5/*` imports Base and other `hvm5` modules.
2. `bendc.bend` imports `hvm5/*`.
3. `sim.bend` imports `hvm5/*` and `bendc.bend`.
4. `deploy.bend` imports `sim.bend` and `bendc.bend`.
5. Tests may import any of the above.
6. No runtime module imports TypeScript, Node, or a JavaScript host.
7. `tools/emit-inet.ts` is build-time backend tooling, not a runtime.

Host effects stay at the executable boundary (`deploy.bend` , `run.bend` , `main.bend` , `bench/bench.bend` , tests). The inet reducer itself is pure.

G.41 · bend-runtime/hvm5/version1/legacy/net-v0/readme.md

Net layer map

All implementation filenames are lowercase, short, and underscore-free. `laws.bend` and `proof.bend` are the local proof gate.

```
net/
  core/             L3 pure net kernel models
    wire.bend       wire-data model, fuel, receiver machine
    rule.bend       toy rule classifier; HVM5 table pending
  model/            L4 contract-side models
    reg.bend        public-name → model-wire registry
    code.bend       version/update/state model
    box.bend        box deploy/migrate with stay-policy
    send.bend       full send pipeline → receipt
    mem.bend        epoch-resolution membrane model
  host/             L0/L1 effect-boundary smoke tests
    codec.bend      model text codec; not graph encoding
    wal.bend        File-effect WAL roundtrip smoke test
    tcp.bend        TCP effect smoke test (one bounded request)
  test/             cross-backend executable checks
    self.bend        dual-backend model self-check
  laws.bend    pure model laws
  proof.bend   proof gate for laws.bend
```

Dependency direction:

```
core/wire,core/rule → model/reg,model/code,model/box,model/send,model/mem
  → host/codec → host/wal,host/tcp → test/self
       laws/proof import the model layers only

`deploy/` (TypeScript, Alchemy):
  src/kernel.ts  pure port of net semantics, exact string parity
  src/store.ts   membrane store (deploy/send/listen/update/sever)
  src/app.ts     route handlers → JSON (no framework)
  src/worker.ts  Cloudflare Worker fetch entry
  alchemy.run.ts Alchemy stack (bend-game shape)
  test/          bun:test parity + API flows (27 tests)
```

The current files intentionally expose model constructors so their laws can execute. They are not the production opaque-wire implementation. The real runtime must replace model `Wire` with a law-declared handle and keep tokens in the membrane only.

G.42 · bend-runtime/hvm5/version1/legacy/lattice-v0/fitness.md

FITNESS — gates that must stay green

No wrapper scripts. Each line below is run directly, in order:

- `bend PROOF.bend` → `All terms check.` (demo app laws).
- `bend net/proof.bend` → `All terms check.` (pure model laws only).
- `bend demos/pure_hvm5_mini/PROOF.bend` (reference) → `All terms check.`
- Dual-backend equivalence (bendoom pattern): `net/test/self.bend` prints byte-identical PASS lines on native, `bend -o .js` + bun, and node.
- Execution tests: `net/host/wal.bend` → `WAL-PASS` exit 0; `tcp.bend` server + client → `served` + `TCP-PASS` exit 0/0.
- Worker tests: `bun test` in `deploy/` → all pass (kernel parity with the Bend selftest literals + full API flows through the worker fetch handler).
- WASM page build: `bun run build:page` in `deploy/` → `artifacts/hvm5/hvm5.wasm` . This is the PR #866 browser-page artifact, not yet the Cloudflare `hvm5-worker.wasm` ABI.
- These are model/effect smoke tests, not proof of opaque wires, live grafting, signatures, crash durability, or continuous service behavior.
- No slice lands with a red gate; no spec weakening to pass.
- Adversarial suite (post-040): forged wires, cross-subnet refs, duplication of linear wires, use-after-sever, fuel exhaustion on copier/fusion/erase paths, WAL crash injection, replay, reentrancy, stale epochs.

G.43 · bend-runtime/hvm5/version1/legacy/lattice-v0/gaps.md

GAPS — open questions blocking slices

- <input disabled="" type="checkbox"> G-01 Exact authoritative HVM5 rule-table revision (blocks 005).
- <input disabled="" type="checkbox"> G-02 True-graft representation after books-first (blocks 030 security gate).
- <input disabled="" type="checkbox"> G-03 Canonical graph encoding preserving SUP/DUP sharing (blocks 080).
- <input disabled="" type="checkbox"> G-04 Persistent-receiver queue/savepoint policy (blocks 050/100).
- <input disabled="" type="checkbox"> G-05 Effect request/reply ABI + retry semantics (blocks 120).
- <input disabled="" type="checkbox"> G-06 Key rotation/recovery story (blocks 060 release).
- <input disabled="" type="checkbox"> G-07 Parallel reducer existence (post-v1; sequential accepted per criterion).
- <input disabled="" type="checkbox"> G-08 Safe-Bend search-by-value wall (found in CP-001): a recursive loop can only branch on structural patterns of its shrinking argument — a helper returning the branch condition creates a banned mutual-recursion cycle. Consequences: registry/lookups are Nat-position structural; U32 array scans with computed control need @unsafe or host primitives. Net internals that need search MUST be structural (lists/terms/Nat) or live behind the membrane.
- <input disabled="" type="checkbox"> G-09 Worker WASM ABI (found in CP-006): PR #866 builds a browser page (`.html/.js/.wasm`) with Emscripten, but the Cloudflare Worker needs a single-thread module export (`hvm5_deploy/send/listen`) without page glue, Window, pthread proxying, or WASI filesystem.
- <input disabled="" type="checkbox"> G-10 HVM5 dynamic package input (found in CP-006): the copied HVM5 demo has hard-coded `program()` and prints through IO; it must accept package bytes/graph through the Worker ABI and retain state between requests.

Resolved: wire representation (`law` handles + affinity, spec §15); replace policy (stay + explicit forwarder); fuel=interactions; Ed25519; per-owner sequence (no block height); locks (per-root mutex + graft lock).

G.44 · bend-runtime/hvm5/version1/legacy/lattice-v0/goal.md

GOAL — inet-native HVM5 contract runtime

Normative spec: `docs/hvm5-inet-contract-runtime-spec.md` . Implementation plan: `docs/hvm5-bendrt-contract-runtime-plan.md` .

DONE DEFINITION (OBSERVABLES)

1. `bend net/proof.bend` prints `All terms check.` (wire/fuel/machine laws).
2. A signed HVM5 package grafts as a private subnet of one live net.
3. The deployer receives wires: shared invoke wire, linear update wire.
4. A public caller sends on the invoke wire with a reply wire and gets an answer.
5. Contract A sends its wire to contract B; B sends back. No other path exists.
6. No wire implies no interaction — demonstrated adversarially, not asserted.
7. Owner-only redeploy + migration; stale/non-owner redeploys rejected.
8. Restart preserves committed subgraphs and rejects stale epochs.
9. Every send is fuel-bounded; over-budget sends commit nothing.

NON-GOALS (V1)

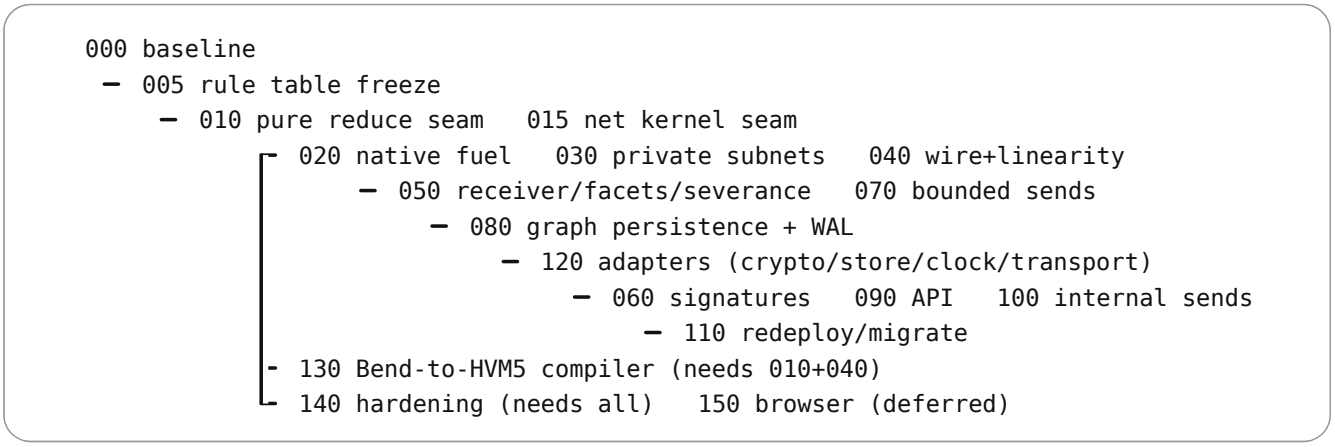
Consensus, replication, finality, monetary settlement, browser runtime, confidentiality from the operator, arbitrary native tenant code.

METRICS

- Deterministic replay: same net + same sends → same subgraph, any schedule.
- Zero cross-subnet interactions without a wire (adversarial suite, not unit tests).
- Fuel covers copier, fusion, erase, match, allocation, serialization paths.
- Crash recovery at every WAL boundary.
- No private key material in the runtime process.

G.45 · `bend-runtime/hvm5/version1/legacy/lattice-v0/graph.md`

GRAPH — slice dependency graph



Numbers are stable IDs. Order follows this graph, not numeric order. Critical path: 000 → 005 → 010 → 015 → 040 → 050 → 070.

G.46 · `bend-runtime/hvm5/version1/legacy/lattice-v0/invariants.md`

ANCHOR — invariants that never change

1. **Connection is authority.** No wire ⇒ no active pair ⇒ no interaction.
2. **No ambient authority.** No global namespace, shared book, or lookup exists.
3. **Hashes identify, wires authorize.** Never the reverse.
4. **Linear wires are uncopyable.** Checker and runtime enforce the same rule; severing leaves no reachable copy.
5. **Fuel is stop-reducing.** One interaction = one countable step, on every path including copier/fusion/erase/match/allocation/serialization.
6. **Failed sends expose nothing.** No committed subgraph, no partial state.
7. **Graphs are homoiconic; capabilities are extra-graph.** Wires never serialize raw; only epoch-bound membrane records cross boundaries.
8. **Confluence or explicit boundary.** Every rule extension carries a diamond argument or a `NonConfluent{reason}` flag that voids agreement claims.
9. **`PROOF.bend` green is necessary, never sufficient** for C effects, crypto, disk, network, or Linux security.
10. **Books are prototype, not boundary.** Book-index isolation tests are necessary but never advertised as the security gate.

G.47 · `bend-runtime/hvm5/version1/legacy/lattice-v0/owners.md`

MAP — component → owner → state → forbidden deps

Component	Owner	Owned state	Forbidden deps
net reducer + grafter	runtime core	live net, subnets, wires	network, filesystem
package linker/grafter	runtime core	subnet snapshot, manifest	arbitrary host effects
wire membrane	runtime core	wire↔token bindings, linearity flags, epochs	user-constructible wire tokens
send manager	runtime core	state-root queue, sequence, reply wires	direct contract IO
persistence adapter	trusted effects	WAL, snapshots	contract-controlled paths
crypto adapter	trusted effects	pubkey verification (Ed25519)	private-key storage

Component	Owner	Owned state	Forbidden deps
API transport	trusted effects	sockets/framing	evaluator internals
contract code	deployer	HVM5 graph/state	native C/JS, ambient refs
native modules	operator	compiled adapters	untrusted deployment

One owner per component. Cross-owner access only through the seams in `seams.md`.

G.48 · bend-runtime/hvm5/version1/legacy/lattice-v0/placement.md

PLACE — capability × target matrix

Capability × target	Allowed	Forbidden	Reason
Pure HVM5 package × net	grafted subnet, passed wires, bounded reduction	raw pointers, global book lookup	affinity + membrane is the boundary
BendRT host × service	reducer, membrane, persistence adapters	tenant code as native plugins	native code escapes the net model
Linux × process	user-space, restricted account, seccomp	ring-0 assumptions, root runtime	OS is primary boundary until audit
TCP client × API	deploy/send/listen over TLS or Unix socket	plaintext bearer tokens	passive theft
Native module × host	crypto/store/clock/transport via injected wire	implicit availability to contracts	ambient authority
Browser × runtime	deferred (client only, post-native)	secrets, host tools in page	page is hostile territory

Kernel imports no platform; all IO via wires. Browser holds no secrets.

G.49 · bend-runtime/hvm5/version1/legacy/lattice-v0/rejects.md

REJECTS — infeasible/out-of-scope with evidence

- R-01 Native C/JS tenant upload in one process: breaks net isolation (shared address space, ambient syscalls). Use HVM5 graphs or processes.
- R-02 Runtime `GRANT` / `REVOKE` ops: dissolved — grant is wire-passing, revoke is sever/proxy (Rholang rule).
- R-03 Runtime facet type: dissolved — facets are separate wires.
- R-04 `state.load/commit` API ops: internal only; exposed via LISTEN.
- R-05 Block-height validity: no chain; per-owner sequence instead.
- R-06 Gas-as-money in v1: fuel accounting only, no settlement layer.
- R-07 Browser runtime in v1: deferred to SLICE-150.

G.50 · bend-runtime/hvm5/version1/legacy/lattice-v0/seams.md

SEAMS — allowed crossings between components

1. `parse(source) -> OpenSubnet` — pure, no IO (SLICE-010).
2. `reduce(net, root, fuel, limits) -> Result<Graph, NetError>` — pure, no IO.
3. Graft seam: `graft(verified_subnet, supplied_wires) -> ExportRoots | GraftError` — atomic; failure leaves no reachable partial subnet.
4. Wire seam: `connect(wire, message) -> active pair ; sever(wire)` for linear.
5. Membrane seam: external token ↔ live root + epoch; tokens never enter graphs.

6. Persistence seam: `snapshot(subgraph) -> bytes` ; `restore(bytes, epoch)` ; canonical encoding only, no raw wires/pointers.
7. Crypto seam: `verify(pubkey, bytes, sig) -> Bool` ; no private keys cross it.
8. Effect seam: `request(effect_wire, graph) -> graph` via linear reply wire; effects excluded from the confluence claim.

Every seam crossing is typed, budgeted, and logged. No other crossings exist.

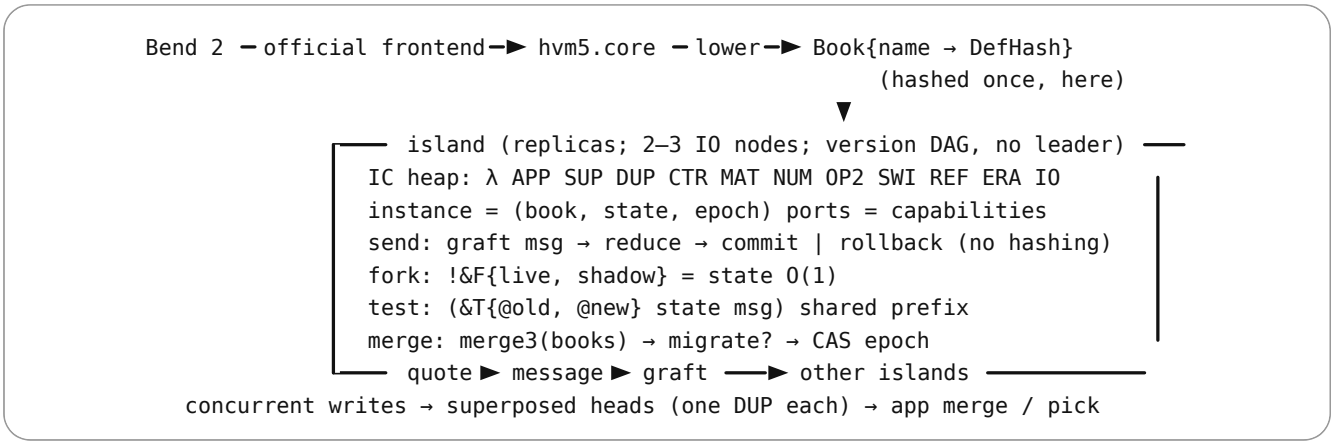
G.51 · `bend-runtime/hvm5/version2/README.md`

HVM5 v2 — forkable Bend 2 on islands of interaction nets

Thesis. Code is an immutable, content-addressed book of interaction-calculus terms. Then:

- forking a running program is one `DUP` node (O(1));
- testing a candidate against the live version runs both in one reduction, superposed with `&T{old, new}` , and shares every step before the first changed definition;
- merging is a three-way merge of definition hashes, followed by an atomic epoch swap.

Programs are written in Bend 2. They are checked by the official frontend and lowered by our backend. The kernel that reduces them is written in Bend 2 and compiled by BendRT. A cluster is a set of **islands**: local IC heaps, replicated on every node that holds a Replicate cap, with no leader, talking only through capability ports.



READ IN THIS ORDER

#	Document	Owns
—	`REVIEW.md`	honest review of v1, every correction, what not to claim
0	`spec/00-overview.md`	the five objects, five operations, invariants, vocabulary
1	`spec/01-calculus.md`	IC syntax, all rules, labels, fuel, confluence, readback, soundness boundary
2	`spec/02-runtime.md`	heap kernel in Bend 2 on BendRT: layout, machine, rollback, IO, parallelism
3	`spec/03-compiler.md`	`--emit-core` , lowering, linear matches, numbers, fragment, ABI
4	`spec/04-code-as-data.md`	books, fork, superposed-book testing, merge3, promote, rollback
5	`spec/05-islands.md`	leaderless version DAG, IO nodes, caps, subscriptions, failure
6	`spec/06-formats.md`	`.ic text` , `Serial` , core bytes, log/checkpoint records, hash domains
7	`spec/07-trust.md`	TCB, invariants, threats, what proofs do and do not cover
—	`plan/00-plan.md`	milestones M1–M6, gates, risks, definition of done
—	`maps/00-map.md`	concept → spec → module, module dependency graph

STATUS (2026-10-23)

Milestone	State	Evidence
M0 v1 frozen	done	`../version1`
M1 IC kernel	done (sequential)	`src/kernel/*` ; `tests/kernel_rules.bend` 28/28; K2 Church c2·c2 = c4; K3 shown (same label gives 1, not 4)
M2 compiler	done for the fragment	`tools/emit-core.ts` (pinned patch, build-time only) → `src/compiler/{core,lower,book,hash}.bend` ; SHA-256 FIPS vectors pass
M3 code as data	done on one island	`tests/e2e_vault.bend` **PASS**: live branches Alice+Bob superposed, Carol rejected (Conflict), `merge_all` = hand merge (state and BookHash), broken branch auto-dropped
M4 membrane	checks, untested	`src/membrane/cap.bend`
M5 islands	in-process mesh done (DAG and world-label stores); TCP not started	`tests/e2e_cluster.bend` **PASS** (D1–D5): any arrival order → same heads + StateHash; IO node killed, nobody blocks; revive from a peer log; commutative book never forks; merge = hand oracle; `tests/e2e_worlds.bend` **PASS**: the same gates on one superposed state, plus per-pair independence (24 orders) and sequencer takeover
M6 benchmarks	not started	`bench/` empty; `LAWS.bend` / `PROOF.bend` not written

Islands have **no leader and no consensus** (`spec/05`). Each message names the versions it was written against. Concurrent conflicting messages become superposed heads, each one DUP on its parent. The application resolves them with the book's `merge` , and proven `law commutes` books never fork. 2–3 IO nodes per island write and do IO. Every replica reduces everything, and killing an IO node blocks nobody.

Upstream fact, stated plainly: **official Bend 2 has no interaction-net backend today.** v2 needs exactly one upstream addition (`bend --emit-core` , pinned as a patch in `tools/emit-core.ts`). Everything after the core file is ours.

G.52 · `bend-runtime/hvm5/version2/REVIEW.md`

Review of the HVM5 draft (v1)

Written from the perspective of an interaction-calculus runtime designer (HVM lineage). The question is: does the draft get us to a cluster of partitioned inets running forkable, mergeable Bend 2 code? Every statement is either true of the v1 code today or labelled as a correction.

1. VERDICT

The goal is right, and it is unusual in a good way. Code as immutable terms, updates as term transformations, and an optimal confluent reducer underneath is a combination nobody ships today.

The draft has the right *nouns* and the wrong *machine*. Four of its central mechanisms do not work as written: the reducer, the shared net, fork, and hashing. Several documents describe a system that does not exist. Fixing this is not a rewrite of the idea. It is replacing the parts that fight IC with parts that use IC.

Area	v1 reality	Grade
Goal	coherent, novel	A
Honesty sections (`11-acceptance`)	mostly honest	B+
Compiler	first-order, coupled to compiler internals	C (prototype)
Reducer	tree + name substitution; DUP-LAM impossible	**F as an inet**
Partitioned net	cross-partition active pairs	**wrong model**

Area	v1 reality	Grade
Fork/merge	`SUP` on live root / record swap	**wrong primitive**
Hashing	per-op graph hash (spec), FNV-32 of `show` (code)	**wrong place**
Capabilities	booleans	placeholder
Cluster	one process, pure model	placeholder
Doc structure	18 normative files, overlapping	C

2. KEEP

1. **The deployed artifact is a term, not source.** The runtime never parses Bend.
2. **One frontend.** The official Bend 2 checker is the language authority.
3. **Kernel in Bend 2, executed by BendRT.** BendRT is a fine host as long as the net lives in an explicit heap (`Array<U64>`), not in Bend's own values.
4. **Hashes identify; capabilities authorize.** Never conflate them.
5. **Capabilities never enter terms.**
6. **Reject, never miscompile.**
7. **Failed operations change nothing.**

3. CORRECTIONS

C1 — THE REDUCER MUST BE A HEAP, NOT A TREE

`eval.bend` substitutes by binder name and relies on "globally unique names". The first duplication of a lambda breaks that invariant, because both copies carry the same names. That is why DUP-LAM and commuting labels are listed as "unsupported". They cannot be supported in this representation without deep copies, and deep copies throw away lazy, shared duplication, the only reason to use IC.

Do instead: an HVM4-style kernel. Nodes live in a linear `Array<U64>` . A term is a 64-bit tagged pointer. Variables are heap locations, so substitution is one write. All IC rules, including DUP-LAM and DUP-SUP with different labels. → `spec/01-calculus.md` , `spec/02-runtime.md` .

C2 — NO WIRE MAY CROSS A MACHINE

"One logical net with bridges between partitions" means some active pairs have agents on two machines. One interaction there costs a network round trip, about $10^4\times$ a local interaction. Worse, the v1 rule "pairs outside the listen set stay in the net" means a pair spanning two spaces is never reduced by anyone.

Do instead: islands. All wires are local. Islands exchange *values*: quote (read back a closed normal form), ship, graft. An island is an actor whose behaviour is an IC term. → `spec/05-islands.md` .

C3 — FORK IS NOT `SUP` ON THE LIVE ROOT

Putting the live root under `&L{old, new}` makes every live send run in two worlds and makes the live answer depend on a collapse policy. That is not a fork.

Do instead:

- **fork** = `!&F{live, shadow} = state` . One node. Copying happens lazily, only where a side rewrites. The book is immutable and shared.
- **test** = run a *superposed book*: every definition whose hash differs becomes `@d = &T{@d_old, @d_new}` , and unchanged definitions stay single. One reduction serves both versions. Everything computed before control reaches a changed definition is computed **once**.
- **merge** = three-way merge of `Name → DefHash` , an optional `migrate` , then a compare-and-swap of `(book, state, epoch)` .

→ `spec/04-code-as-data.md` .

Precision about the sharing claim: sharing covers the common prefix of the computation, the inputs, and unchanged state. Two *independently written* code paths that happen to compute the same thing are not shared. Report the measured `sharedRatio` , never a promised one.

C4 — NEVER HASH A LIVE NET

A net in mid-reduction has no canonical form. Heap addresses and redex order are scheduling artifacts, and canonical labelling of graphs is expensive. Per-op hashing is also exactly what made the local node slow.

Do instead:

- hash **code once**, at compile time, per definition (Merkle DAG, Unison-style);
- hash **state only at checkpoints**, after readback to a closed normal form, which has a canonical serialization;
- **never hash during reduction.** Replicas agree because of the ordered log and confluence, and checkpoints confirm it every K sends.

→ `spec/05-islands.md` §5, `spec/06-formats.md`.

C5 — "MERGE" IN V1 IS PROMOTION

Replacing live code with a candidate is a promotion. Merging two people's forks needs structure that survives compilation, and reduced nets lose definition boundaries.

Do instead: books are `Map<Name, DefHash>`. `merge3(base, a, b)` per name. A conflict means the same name changed differently on both sides, and it is resolved by writing a new definition in Bend 2. Type changes go through `migrate`. → `spec/04-code-as-data.md` §6.

C6 — USE STRONG CONFLUENCE; STATE ITS LIMITS

IC is strongly confluent: the one-step relation has the diamond property, so all complete reductions have the same result and the same length. Use this:

- results do not depend on the scheduler;
- **fuel is deterministic** if it is defined on a canonical strategy (lazy sends) or on complete reduction (readback);
- replicas need no per-step comparison.

Limits that must be written down: it holds for the pure rules only. `IO`, host nondeterminism and `Stuck` redexes are boundaries.

C7 — IC IS NOT THE Λ-CALCULUS; SAY WHERE THEY DIFFER

Optimal reduction without brackets is incorrect for some terms: a lambda that is duplicated and then meets its own duplicator under the same label. The draft never mentions this. **Do instead:** a label discipline. Compiler-inserted dups on lambda-typed values get fresh labels per expansion. User-written `Sup` / `Dup` are banned in transitions. The runtime's fork and test labels come from a reserved range. First-order contract code is always safe. → `spec/01-calculus.md` §7.

C8 — BEND 2 → INET IS NOT AN UPSTREAM FEATURE

`tools/emit-inet.ts` walks compiler-internal HOAS (`Book` / `HTerm`). It will break on upstream refactors and covers only the first-order fragment. **Do instead:** one small upstream addition, `bend --emit-core`, producing an erased first-order core with linearized matches. Delete `emit-inet.ts` the day it lands. → `spec/03-compiler.md`.

C9 — NUMBERS

Unary `Nat` in IC would be catastrophically slow. **Do instead:** `NUM` / `OP2` / `SWI` on U32. `Nat.sub` saturates (as in Bend 2). Overflow on add/mul is `Stuck`, not a wrap, because silent wrap in contract code is a security bug.

C10 — DOCUMENT STRUCTURE

There are 18 normative files and the same concept appears in three places. **Do instead:** v2 has one calculus file, one runtime, one compiler, one code-as-data, one cluster, one formats and one trust file, and every gate has a status.

4. STOP CLAIMING

- "Bend 2 compiles to interaction nets." Upstream it does not.
- "The runtime is inet-native." Not until kernel gates K1–K4 pass.
- "One logical net across the cluster." Islands and messages.
- "Fork = DUP, merge = SUP collapse." Fork is `DUP` of state, test is `SUP` of books, merge is CAS.
- "Every operation is hash-quorum committed." Only checkpoints.
- "Isolation is proven." Only for the cap-graph model; not BendRT, the OS or the network.

5. WHY THIS DESIGN IS WORTH BUILDING

System	Has	Lacks
Unison	content-addressed defs, structural code merge	live instances, optimal sharing
Erlang/OTP	hot swap, old/new draining	code as data, deterministic replay
Ethereum	replicated deterministic state machines	cheap fork/test, code merge
HVM/IC	optimal sharing, confluence	distribution, code identity, authority
E / Spritely	object capabilities	optimal reduction, code merge

v2 combines all five properties. The one capability nobody else has is **testing a candidate against live code in a single shared reduction**. Measure it first (benchmark B5). If it does not pay off, fork/merge still works without it.

G.53 · bend-runtime/hvm5/version2/spec/00-overview.md

00 — Overview

Status: normative, HVM5 v2.

1. FIVE OBJECTS

Object	Definition	Mutable?
Term	an interaction-calculus term in an island heap (`01`)	consumed by reduction
Book	`Map<Name, DefHash>` plus a manifest. Each def is a closed term template	never
Instance	`(book: BookHash, state: Term, epoch: Nat, seq: Nat)`	only by sends and promotes
Island	a set of nodes, each a full replica (own heap) of its documents; 2–3 are IO nodes (`05`)	—
Capability	membrane record `(port, rights, mode, epoch)` behind a random token	revocable

2. FIVE OPERATIONS

```
deploy(book, init)        state := normalize(@init)                      new instance
send(cap, msg)            state', reply := (@transition state msg)       atomic, fuel-bounded
fork(inst)                !&F{state, shadow} = state                     one node
test(fork, msgs)          (@transition_T state msg) on superposed book T shared prefix
promote(inst, cand, e)    CAS epoch e; state := (@migrate state)?        atomic
```

`merge3` (three-way book merge) and `abandon` (erase the shadow) complete the set. All of them are term operations. None of them edits running code in place.

3. LAYERS

```
L6  tools      bend --emit-core · hvm5 CLI · signer
L5  registry   DefHash → Serial blobs · names → BookHash · authority rows
L4  islands    version DAG · writer chains · IO nodes · replicas · subscriptions (no leader)
L3  membrane   caps · rights · linearity · epochs · fuel & heap budgets
L2  instance   init · transition · migrate · fork · test · promote
L1  kernel     IC heap & rules · readback · graft · rollback   (Bend 2 on BendRT)
L0  host       BendRT binary · OS · TCP · disk · Ed25519 · BLAKE3
```

L0–L1 are trusted. L3 is the security boundary. Contract code is untrusted and fuel-bounded. L4–L6 are clients of L3.

4. INVARIANTS

1. **Local wires.** No heap pointer refers outside its island.
2. **No ambient authority.** A term can reach only its book and its cap handles.

3. **Hashes identify, capabilities authorize.**
4. **Books are immutable.** A new version is a new book.
5. **No hashing on the reduction path.** Only compile, checkpoint and records.
6. **Confluent core.** Pure results are scheduler-independent (`01` §6).
7. **Atomicity.** A send or promote commits fully or not at all.
8. **Linear authority is not copyable.**
9. **Declared boundaries.** `IO` , host numbers and the soundness boundary of IC (`01` §7) are explicit.

5. NON-GOALS

Consensus of any kind: v2 has no leader and no total order (`05` §2). Tokens or money. Cross-island atomic transactions (use sagas). Confidentiality from island operators. Lowering every Bend 2 feature (the fragment is listed in `03` §5). GPU (later; the kernel layout is compatible).

6. VOCABULARY

Word	Meaning
redex	an active pair: an eliminator meeting a value (`APP` /λ, `DUP` /x, `MAT` /#K, …)
interaction	one rule application; the unit of fuel
label	24-bit tag on `SUP` / `DUP` ; equal labels annihilate, different labels commute
quote / readback	heap term → canonical `Serial` of its normal form
graft	`Serial` → fresh heap term
superposed book	book where changed defs are `&T{old, new}`
projection π_i^T	readback picking side `i` of every `&T`
epoch	instance generation; incremented by promote
version	one DUP on a parent state; id `V{writer, seq}` (`05` §2)
heads	versions no other version names as parent: the live superposition
checkpoint	canonical heads + StateHash; compaction = restore(take)

G.54 · bend-runtime/hvm5/version2/spec/01-calculus.md

01 — The calculus

Status: normative, HVM5 v2. The interaction calculus (IC) with typed data, numbers, references and an IO boundary. Notation follows HVM4.

1. SYNTAX

```
t ::= x                       variable (affine: occurs at most once)
    | λx. t                   lambda
    | (t t)                   application
    | &L{t, t}                superposition, label L
    | !&L{x, y} = t; t        duplication, label L, binds x and y
    | *                       eraser
    | #K{t, …}                constructor K (arity from the book)
    | ~t {#K₁: t; …; #Kₙ: t}  match; arm i is a λ over the fields of Kᵢ
    | n                       U32 number
    | (⊛ t t)                 binary op ⊛ ∈ {+ -ˢ × / % = ≠ < ≤ & | ^ << >>}
    | ?t {0: t; +: t}         number switch; `+` arm receives n-1
    | @f                      reference to book entry f
    | %io(op, t, λr. t)       IO request; the continuation receives the result
```

Scope is global. A variable may occur outside the body of the λ that binds it. This is IC, not λ-calculus. It is what lets DUP-LAM split a lambda into two without copying its body. Every variable is used at most once. Sharing is explicit `DUP` .

2. RULES

`x ← v` means "substitute v for x" (one heap write).

```
APP-LAM   (λx.f a)              → x ← a; f
APP-SUP   (&L{f,g} a)           → !&L{a₀,a₁} = a; &L{(f a₀), (g a₁)}
APP-ERA   (* a)                 → erase a; *

DUP-LAM   !&L{p,q} = λx.f       → p ← λx₀.g₀; q ← λx₁.g₁; x ← &L{x₀,x₁}; !&L{g₀,g₁} = f
DUP-SUP   !&L{p,q} = &L{a,b}    → p ← a; q ← b                                  (annihilate)
DUP-SUP'  !&L{p,q} = &M{a,b}    → p ← &M{a₀,b₀}; q ← &M{a₁,b₁};
                                    !&L{a₀,a₁} = a; !&L{b₀,b₁} = b                (commute, L≠M)
DUP-CTR   !&L{p,q} = #K{a…}     → p ← #K{a₀…}; q ← #K{a₁…}; ∀i !&L{aᵢ₀,aᵢ₁} = aᵢ
DUP-NUM   !&L{p,q} = n          → p ← n; q ← n
DUP-ERA   !&L{p,q} = *          → p ← *; q ← *

MAT-CTR   ~#Kᵢ{a…}{…}           → (armᵢ a…); erase other arms
MAT-SUP   ~&L{a,b}{arms}        → !&L{r₀,r₁} = arms; &L{~a{r₀}, ~b{r₁}}
MAT-ERA   ~*{arms}              → erase arms; *

SWI-NUM   ?n{z; s}              → n = 0 ? (z; erase s) : ((s n-1); erase z)
SWI-SUP   ?&L{a,b}{z; s}        → as MAT-SUP
SWI-ERA   ?*{z; s}              → erase z, s; *

OP2-NUM   (⊛ n m)               → n ⊛ m                (see §5 for overflow)
OP2-SUP₀  (⊛ &L{a,b} m)         → !&L{m₀,m₁} = m; &L{(⊛ a m₀), (⊛ b m₁)}
OP2-SUP₁  (⊛ n &L{a,b})         → &L{(⊛ n a), (⊛ n b)}  (n is a number; copying is free)
OP2-ERA   (⊛ * m) / (⊛ n *)     → erase other; *

REF       @f                    → instantiate(book[f])  (labels per §7)
IO        %io(op, v, k)         → Yield(op, v, k)       (stops reduction, see runtime §8)

erase λx.f  → x ← *; erase f      erase &L{a,b} → erase a; erase b
erase #K{…} → erase fields        erase n       → ·
```

Typed clash. `APP` on a number or constructor, `MAT` on a λ or a wrong-type constructor, `OP2` on a non-number: the redex is left in place and reduction returns `Stuck{loc}`. A checked program never produces one. Grafted foreign data can.

3. INTERACTION COUNT

Every rule above, including each erasure step and each `REF`, is one interaction. Fuel is interactions.

4. NORMAL FORMS AND READBACK

A term is **normal** when it contains no redex. A normal closed term is a tree of λ, `#K`, numbers, `*`, residual `&L{…}` and residual `!&L` bound to variables.

`quote(t)` normalizes `t` and serializes it (`06` §2): binders as de Bruijn levels, labels renumbered from 0 in order of first occurrence. Two α-equivalent normal forms have identical bytes.

Projection `πᵢᴸ(t)`: readback that replaces each `&L{a,b}` by side `i`, and each `!&L{x₀,x₁} = v` by `xᵢ ← v`, `x₁₋ᵢ ← *`. This is how superposed test results are separated (`04` §5).

5. NUMBERS

U32. `+`, `×` and `<<` that overflow give `Stuck`; contracts must not wrap silently. `-ˢ` saturates at 0 (Bend 2 `Nat.sub`). `/` and `%` by zero give `Stuck`. Comparisons return `1` / `0`. The compiler wraps them into `Bool` constructors.

6. CONFLUENCE AND DETERMINISM

Theorem (strong confluence of IC). If `t → u` and `t → v` with `u ≠ v`, there is `w` with `u → w` and `v → w`, each in one step. Hence all complete reductions of `t` reach the same normal form in the same number of steps.

v2 extensions (`NUM`, `OP2`, `SWI`, `CTR`, `MAT`, `REF`) are local and non-overlapping, so the diamond property is preserved. This is proof obligation **L1** in `LAWS.bend`.

Consequences used by the system:

Use	Requires
results independent of scheduler	pure fragment (no `IO` pending, no `Stuck`)
fuel identical on all replicas	the canonical strategy (`02` §4) for sends; complete reduction for readback
parallel readback/checkpoint quote	complete reduction (count is order-invariant)

7. SOUNDNESS BOUNDARY: IC IS NOT THE Λ-CALCULUS

Optimal reduction without brackets gives wrong results when a lambda copied by a duplicator with label `L` later meets a `DUP` or `SUP` of the *same* label that came from a different duplication. The `&L{x₀,x₁}` created by DUP-LAM annihilates with the wrong partner.

Label discipline (normative):

1. **Compiler dups** (inserted for variables used ≥2 times) on values whose type is first-order data get a static label per `(DefHash, site)`. That is safe: DUP-CTR/NUM never create SUPs.
2. **Compiler dups on λ-typed values** get a **fresh label per REF expansion** from a per-island counter. Labels are renumbered from 0 at every checkpoint compaction. If the counter would overflow 24 bits, the kernel forces a checkpoint first.
3. **User `Sup` / `Dup` syntax** is rejected in contract books. Superposition is a runtime tool here, not a user feature.
4. **Runtime labels** come in two classes:
   - *rdata* `0xF00000–0xF7FFFF`: dups of data made by the runtime (fork `F` / `G`, version DUPs, the pair split `D`);
   - *rworld* `0xF80000–0xFFFFFF`: worlds (live-branch and test labels `T`).

   Counters: fresh = register 7, rdata = 8, rworld = 21 (`02` §2).
5. **Erase shortcut.** `!&L{x, *} = v → x ← v` is applied only for data-only labels (static, rdata). For fresh and world labels, `v` may hold a SUP of the same label that must still annihilate, so the dup stays and the erased side becomes garbage.
6. **DUP-REF.** A dup of a `REF` copies the reference (`!&L{a,b} = @f → a ← @f, b ← @f`).

Bend 2's kinds guarantee that multi-use binders are Data, hence first-order, so rule 1 covers every compiler dup.

Under 1–6, no unintended annihilation can occur in contract code. (Obligation **L2**.)

8. LAWS (`LAWS.BEND`)

Id	Law
L1	local diamond for every rule pair in §2
L2	label discipline ⇒ no cross-annihilation
L3	`quote` is canonical: α-equivalent normal forms give equal bytes
L4	π_0^T(run(superpose(A,B,T), x)) = run(A, x) and π_1^T(…) = run(B, x)
L5	π_0^f of a forked state equals the original state
L6	`merge3` table (`04` §6) is commutative in its two branches

G.55 · bend-runtime/hvm5/version2/spec/02-runtime.md

02 — Runtime: the IC kernel in Bend 2 on BendRT

Status: normative, HVM5 v2.

1. HOST MODEL

BendRT compiles Bend 2 to a native program with a flat evaluator. It knows nothing about active pairs. The kernel is an ordinary Bend 2 program that owns an explicit heap and implements `01` on it. Bend 2 `Array` is single-owner and updated in place, which is exactly what a heap needs.

2. POINTER AND HEAP

```
Ptr  = two U32 words: hi = sub:1 | tag:7 | lab:24, lo = loc   (Bend 2 has no U64)
Heap = Array<U32>          single owner, threaded through the kernel
```

Tag	Nodes at `loc`	Notes
`VAR`	—	`loc` is a binder slot. If the slot holds a `SUB`, follow it
`SUB`	value	marks a filled slot (substitution done)
`ERA`	—	
`LAM`	[body]	the binder slot is `loc+0`, the body is stored in it until substituted
`APP`	[fn, arg]	
`SUP`	[a, b]	`lab` = label
`DP0` / `DP1`	[value] shared	the two projections of one `DUP`; `lab` = label
`CTR`	[f_1 … f_n]	`lab` = constructor id; arity from the book table
`MAT`	[value, arm_1 … arm_n]	`lab` = type id
`SWI`	[value, z, s]	
`NUM`	—	value inline in `loc`
`OP2`	[a, b]	`lab` = operator
`REF`	—	`loc` = def index into the island's book table
`IOR`	[op, arg, k]	IO request

Layout: registers in words 0..31 (7 fresh label, 8 rdata label, 21 world label, 22 its tx copy, …), book table at 16, trash at 65552, nodes from 66576.

Allocation is a bump pointer only. **There are no free lists.** The write log grows down from the top, and rollback is bit-identical. Garbage (erased sides, pruned versions) is reclaimed by compaction at checkpoints: quote every live instance, reset the heap, graft back (`island/checkpoint.bend`, compact = restore(take)). There is no GC.

3. BOOK TABLE

Each island keeps `DefIndex → (DefHash, Template)`. A template is the def's `Serial` pre-decoded into a node array with relative locations. `instantiate` copies it to fresh heap space, relocates pointers and assigns labels per `01` §7. One `REF` expansion is one interaction, plus copying work proportional to the template size. That copying is not counted as fuel, but it is counted against the heap budget.

4. MACHINE

- `whnf(heap, p)`: an explicit stack of eliminator frames (`APP`, `MAT`, `SWI`, `OP2`, `DP0/DP1`). Pop a frame when the head is a value and apply the rule. This is the **canonical strategy**: leftmost-outermost, call-by-need through `DUP`. Its interaction count defines fuel for sends.
- `normalize(heap, p)`: `whnf`, then recurse into children. Used by `quote`. Complete reduction, so the count is order-invariant and may run in parallel (§9).
- A send reduces `(@transition state msg)` to WHNF, then forces the `Pair{state', reply}` constructor to WHNF. State may stay lazy between sends. Reply is normalized (it leaves the island).

5. BUDGETS

Per send: `fuel` (interactions) and `heap` (nodes allocated). Exceeding either gives `OutOfFuel` or `OutOfHeap`, and the send rolls back.

6. TRANSACTIONS (ROLLBACK WITHOUT COPYING)

Every heap write in a send appends `(loc, old)` to a **write log**. The allocation pointer is saved at start. `Done` discards the log. `Stuck`, `OutOfFuel`, `OutOfHeap` or `Denied` replays the log in reverse and resets the pointer. Cost is O(writes). The pre-send heap is restored bit for bit (gate K6).

7. QUOTE AND GRAFT

- `quote(p)` : `normalize` , then a pre-order walk emitting `06` §2 bytes. It fails if the term contains `IOR` or `Stuck` .
- `graft(bytes)` : decode, allocate, link, and return the root pointer. The decoder is total: malformed bytes give `Denied` .
- Messages crossing islands also carry `REF` s by **DefHash**. Before grafting, the receiver resolves missing hashes from the registry (`05` §3).

8. IO

When reduction hits `IOR(op, v, k)` , the machine returns `Yield{op, v, k}` . The send rolls back and returns NeedIO (4) with the op in the receipt's `note` . Only an **IO node** (the envelope's writer, `05` §3) asks the host and records the answer in the envelope's `ios` , then runs again. Replicas apply the envelope with the recorded answers and never call the host. Transitions without `IOR` are marked `pure` by the compiler, and only pure transitions can be SUP-tested.

9. PARALLELISM

`normalize` forks children with disjoint subheaps (BendRT `fork`), merging allocation through per-worker arenas. Strong confluence gives the same normal form and the same count. The sequential kernel ships first. The parallel scheduler is enabled only after gate K4 passes on the corpus.

10. MODULES

```
src/kernel/ptr.bend      tags, encode/decode
src/kernel/heap.bend     registers, bump alloc, write log, budgets, label counters
src/kernel/book.bend     book table, instantiate, label assignment
src/kernel/rules.bend    every rule in 01 §2
src/kernel/whnf.bend     canonical strategy, fuel
src/kernel/norm.bend     normalize (sequential, then parallel)
src/kernel/quote.bend    quote / graft / projection πᵢᴸ
```

11. GATES

Gate	Check
K1	each rule in `01` §2 has a unit test
K2	DUP-LAM corpus (Church numerals `2²²` , `λf.λx.f (f x)` duplicated) matches HVM4 normal forms
K3	label discipline: a corpus that breaks with static lambda labels passes with fresh ones
K4	fuel: canonical strategy is deterministic; complete reduction gives the same count sequential vs parallel
K5	quote canonical: α-variants give equal bytes
K6	rollback: failed send leaves a bit-identical heap
K7	conformance: corpus results equal BendRT running the same Bend 2 source
K8	native build reports interactions/s on B1; target within 3× of HVM4 C single-thread (measured, not assumed)

G.56 · `bend-runtime/hvm5/version2/spec/03-compiler.md`

03 — Compiler: Bend 2 → IC

Status: normative, HVM5 v2.

1. BASELINE (TRUE TODAY)

Official Bend 2 compiles to BendRT (C/JS). It has **no** inet backend. HVM5 owns a backend and must not own a frontend.

```
contract.bend
  ─ official frontend: layout · modules · dependent types · quantities · totality · laws
      ─ bend --emit-core → hvm5.core (only upstream change)
          ─ src/compiler/lower.bend → Book (ours, Bend 2)
```

`../version1/tools/emit-inet.ts` (which reads compiler internals) stays as a migration aid. Delete it when `--emit-core` lands (gate C7).

2. CORE (`HVM5.CORE`)

Erased, first-order, checked. Byte format in `06` §3.

```
Module = { format, compiler, types: [Type], defs: [Def], exports }
Type   = { name, ctors: [{ name, arity }] }       ctor id = canonical index
Def    = { name, arity, pure: Bool, body: Expr }
Expr   = Var(i) | Ref(name) | Lam(Expr) | App(Expr, Expr)
       | Ctr(type, ctor, [Expr])
       | Mat(type, Expr, [Arm]) , Arm = { fields: n, body: Expr }   all ctors, in order
       | Num(u32) | Op2(op, Expr, Expr) | Swi(Expr, Expr, Expr)
       | Let(Expr, Expr) | Era | Io(op, Expr, Expr)
```

The emitter (upstream side) must:

1. emit only checked modules, rejecting holes, open laws and `@unsafe` in exported transitions;
2. erase types, proofs, `&0` arguments, `Eq` / `Rfl` / `Rwt` evidence;
3. compile nested and dependent matches to one-level total `Mat` s;
4. map `Nat` to `Num` / `Op2` / `Swi` (§4) and `String` to lists of U32;
5. be deterministic: same source and version give the same bytes.

3. LOWERING (OURS)

`lower.bend : Module → Book`.

1. **Linearize matches.** Free variables of a `Mat` are abstracted into every arm and passed after the match: `~x{#A: λa. f(a, y); #B: g(y)}` becomes `(~x{#A: λa. λy. f(a,y); #B: λy. g(y)} y)`. A variable used in several arms is therefore *not* duplicated. This is the standard HVM technique and the biggest constant-factor win in contract code.
2. **Affinity.** After step 1, count occurrences. Zero uses become `*` (erase). Two or more uses build a balanced tree of `DUP` s with labels per `01` §7: static for first-order types, fresh-per-expansion for λ-typed values. The core carries a `ho` bit per binder, set by the emitter from the checked type.
3. **Constructors and matches.** `Ctr` becomes `#K{…}` with a numeric id and `Mat` becomes `~`, both with ids from `Type`.
4. **References.** `Ref(name)` becomes `@index`. The book maps index → DefHash.
5. **Let.** A linear let substitutes directly. A non-linear let goes through the affinity rule.
6. **Reject.** `Sup` / `Dup` written in source (`01` §7.3), arrays, maps and FFI. Each rejection carries the source span.

COUNTER, LOWERED

```
@init       = #Counter{0}
@transition = λs λm ~s{
  #Counter: λv ~m{
    #Inc:   λby λv #Counter{(+ v by)}
    #Reset: λv  (erase v) #Counter{0}
  } v
}
```

(`v` is passed through the inner match by linearization. The `Reset` arm erases it.)

4. NUMBERS

Bend 2	Core	Semantics
`Nat.add`, `Nat.mul`	`Op2(+)`, `Op2(×)`	overflow → `Stuck`
`Nat.sub`	`Op2(-ˢ)`	saturating, as Bend 2

Bend 2	Core	Semantics
`Nat.div` , `Nat.mod`	`Op2(/)` , `Op2(%)`	by zero → `Stuck`
`Nat.eq/lt/le`	`Op2` → `Bool` ctor	
`match n: 0n / 1n+p`	`Swi`	
`n` literal	`Num`	must fit U32 or compile error

5. SUPPORTED FRAGMENT

Feature	v2
ADTs of any arity, nested/dependent match (after erasure)	yes
recursion, mutual recursion	yes
higher-order functions, closures	yes, with λ-label discipline
`Nat` , `Bool` , strings as lists	yes
laws, proofs, `Eq` , rewrites	erased
declared IO ops	yes, marks the def impure
user `Sup` / `Dup`	rejected in contract books
`Array` , `Map` , in-place update	rejected (v3: linear vector nodes)
`@unsafe` , FFI	rejected in exports

6. CONTENT ADDRESSING (WHERE CODE IS HASHED)

```
DefHash  = H("hvm5.def.2\0"  ‖ Serial(template) ‖ DefHash(ref₁) ‖ … )
SCCHash  = H("hvm5.scc.2\0"  ‖ templates of the strongly connected component)   for mutual recursion
BookHash = H("hvm5.book.2\0" ‖ sorted (name, DefHash) ‖ manifest)
```

Computed once, at compile time or registration. Renaming does not change a `DefHash` . `H` = BLAKE3 by default (SHA-256 for interop), fixed per cluster.

7. CONTRACT ABI

```
type State …
type Msg …
type Reply …                                  # optional
def init() -> State
def transition(s: State, m: Msg) -> Pair(State, Reply)    # or -> State
def view(s: State) -> View                    # optional: what subscribers see
def migrate(old: OldState) -> State           # optional: required when State changes
def merge(base: State, a: State, b: State) -> State   # optional: resolves concurrent versions (05 §2.4)
law commutes:                                 # optional, must be proven
  for s: State
  for x: Msg
  for y: Msg
  {next(next(s, x), y) == next(next(s, y), x) : State}
```

`emit-core` emits `merge` as an ordinary def. A filled `law commutes` (checked by the official frontend's `book_valid`) is recorded as the marker def `~commutes` (arity 0, body ERA). Being a def, it is covered by the BookHash. A `law commutes` with no proof is an error. `def independent(x: Msg, y: Msg) -> Bool` is admitted only with a filled `law independent_ok` (`05` §2.7); the frontend rejects a false claim (`contracts/kv.bend`).

Open bug. `Bool.pick(U32, c, y, x)` inside `merge` read back as erased fields (`Kv{*, *}`). The same code with an explicit `match` is correct (`contracts/kv.bend` uses that). The lowering or the erase rule for `Bool.pick` needs a kernel test. Example: `contracts/flags.bend` (a proven grow-only flag set) and `contracts/vault/merge.bend` (delta merge `a + b −`

base).

Manifest: `{ state, msg, reply, view?, entry: {init, transition, migrate?}, pure }` , plus the name-addressed optional defs `merge` and `~commutes` .

8. GATES

Gate	Check
C1	`--emit-core` (upstream or pinned patch) gives deterministic bytes
C2	`lower` covers every `Expr`
C3	corpus (counter, vault with `Nat.sub` , order book with 4-field ctors, list ops, closures, nested match) equals BendRT results
C4	each unsupported feature gives an error with a span
C5	same source twice gives the same `BookHash`
C6	rename a def: same `DefHash`
C7	`emit-inet.ts` deleted

G.57 · `bend-runtime/hvm5/version2/spec/04-code-as-data.md`

04 — Code as data: fork, test, merge

Status: normative, HVM5 v2.

1. TWO LEVELS

Level	Representation	Used for
Book	`Map<Name, DefHash>` + manifest	identity, diff, merge, patches, rollback
Term	IC term in a heap	running, forking state, superposed testing

Merging happens on books. A reduced net no longer has definition boundaries.

Homoiconic loop. Each def's core is available as a Bend 2 value of type `Core.Expr` (`06` §3). A Bend 2 program holding the `Registry` right can read it, transform it (refactor, instrument, optimize, patch), and submit the result, which the registry lowers and hashes. Code is ordinary data for code.

```
def add_fee(t: Core.Expr) -> Core.Expr   # an ordinary Bend 2 function over code
```

2. INSTANCE

```
Instance = { id, book: BookHash, state: Ptr, epoch, seq, forks: [ForkId] }
```

3. FORK — ONE NODE

```
fork(inst) -> Fork
  F = fresh runtime label                      (reserved range, 01 §7)
  !&F{s₀, s₁} = inst.state
  inst.state  = s₀                             live continues on s₀
  Fork{ base: (inst.book, inst.seq), book: inst.book, state: s₁, label: F }
```

Cost: one `DUP` node. A side copies a subterm only when its own reduction forces `DP0` / `DP1` through it. State the live side never rewrites is shared forever. The fork is a private instance with its own ports, held by the forker.

4. EDIT

```
patch = { set: [(name, DefHash)], remove: [name], migrate?: DefHash }
fork.book := apply(fork.book, patch)            new BookHash; old book untouched
```

5. TEST — SUPERPOSED BOOK

Given base book `A` and candidate `B`, define for a fresh test label `T`:

```
superpose(A, B, T)[n] =
  A[n]                        if A[n] == B[n]          (same DefHash: one copy)
  &T{A[n], B[n]}              if both exist and differ
  B[n] / A[n]                 if only one side has n   (reachable only from that side)
```

Run one message on both versions at once:

```
r = normalize((@transition state msg))          under superpose(A, B, T)
old = π₀ᵀ(r)      new = π₁ᵀ(r)                  01 §4
```

Why this is the point of using IC:

- control flows through unchanged definitions **once**;
- at the first reference to a changed def, `APP-SUP` / `MAT-SUP` split the computation, and `DUP` copies only the continuation that depends on the result;
- inputs (`msg`, the state) are duplicated lazily, so untouched parts are never copied.

Exact claim (law L4): `π₀ᵀ(r) = run(A)` and `π₁ᵀ(r) = run(B)`. The amount of sharing is a measured quantity, not a promise:

```
sharedRatio = 1 − fuel(r) / (fuel(run A) + fuel(run B))
```

N candidates. Nest labels `T₁…T_k` to test 2^k variant combinations of k changed defs in one run. Combinatorial A/B with shared work.

Replay test.

```
test(fork, inputs) → Report{ diffs: [(i, old, new)], fuel, sharedRatio, laws }
  s := fork.state
  for m in inputs: r := (@transition s m) on superpose; record π₀, π₁; s := π₁ᵀ(state part)
```

Rules:

- Only `pure` transitions can be SUP-tested. Impure ones are tested by plain replay against recorded IO results.
- If `State` changed, test `&T{(t_A s m), (t_B (@migrate s) m)}`.
- `laws` runs the book's declared `LAWS` defs on the candidate.

6. MERGE — THREE-WAY ON BOOKS

```
merge3(O, A, B)[n]:
  a == b        → a
  a == o        → b
  b == o        → a
  otherwise     → Conflict(n, o, a, b)
manifest (State/Msg types): same rule; a State change needs migrate in the result
```

Equality is `DefHash` equality: exact, O(1). Resolve a conflict by writing a new Bend 2 def. Commutative in `A` / `B` (law L6).

7. PROMOTE — ATOMIC

```
promote(inst, cand, mode, expectedEpoch):
  require inst.epoch == expectedEpoch                    CAS
  require Update capability (linear), consumed
  mode = swap   (default, Erlang semantics):
           state' = cand.migrate ? normalize((@migrate inst.state)) : inst.state
  mode = replay (strict): state' = fold cand.transition over log[fork.base.seq+1 .. now]
                          starting from the fork-time state
  inst := { book: cand.book, state: state', epoch: epoch+1 }
  log:   Promote{inst, oldBook, newBook, epoch, mode}
  caps:  old epoch → Draining (deadline) → Dead
```

Sends ordered before the promote entry used the old book. Sends after it use the new one. A send tagged with a stale epoch is rejected (`StaleEpoch`). Nothing is half applied.

Rollback = promote back to the old book (which is still in the registry), with a reverse migrate if the type changed.

8. ABANDON

`erase` the shadow s_1. The `DUP` then resolves to the live side, so no live work is lost. Drop the fork's ports. The live instance is unchanged (gate F7).

9. EXAMPLE: FIXING A VAULT, WHILE BOB ALSO EDITS IT

```
type State is Data:
  Vault{balance: Nat, owner: Nat}
type Msg is Data:
  Deposit{amount: Nat}
  Withdraw{amount: Nat, who: Nat}

def transition(s: State, m: Msg) -> State:     # book A (live): no owner check
  match s:
    case Vault{balance, owner}:
      match m:
        case Deposit{amount}:     Vault{Nat.add(balance, amount), owner}
        case Withdraw{amount, who}: Vault{Nat.sub(balance, amount), owner}
```

Alice forks and edits only `transition` :

```
        case Withdraw{amount, who}:
          match Nat.eq(who, owner):
            case True{}:  Vault{Nat.sub(balance, amount), owner}
            case False{}: Vault{balance, owner}
```

```
hvm5 fork    vault#1                      → f1  (1 node)
hvm5 edit    f1 --set transition=#9c1…    → book A'
hvm5 test    f1 --replay last:1000        → diffs: 3 foreign Withdraws; sharedRatio measured
```

Bob forks the same base, adds `def fee(a: Nat) -> Nat` and changes the `Deposit` arm. That also changes `transition` , so his book is `B` .

```
hvm5 merge3 base=A a=A' b=B
  fee         → take B      (only Bob)
  transition  → Conflict    (both changed)
hvm5 edit    m --set transition=#d40…     # Alice writes the combined transition in Bend 2
hvm5 test    m --replay last:1000
hvm5 promote vault#1 m --epoch 4          → epoch 5; old caps draining
```

10. GATES

Gate	Check
F1	fork allocates O(1) nodes, independent of state size
F2	untouched state subtrees are never copied (allocation counter)
F3	L4: projections of a superposed run equal separate runs (corpus)

Gate	Check
F4	`sharedRatio` reported on B5; > 0 on the prefix-heavy corpus
F5	`merge3` table and type-conflict tests; L6
F6	promote CAS: stale epoch rejected; swap and replay modes; no lost live sends
F7	abandon leaves the live heap observationally unchanged
F8	rollback to the old book reproduces old behaviour

G.58 · `bend-runtime/hvm5/version2/spec/05-islands.md`

05 — Islands: the leaderless version DAG, IO nodes, capabilities

Status: normative, HVM5 v2. Supersedes the leader-ordered log (`attic/05-islands-leader.md`).

1. WHY ISLANDS

A local interaction costs nanoseconds. A network round trip costs 10^4–10^5 times more. So:

- **every pointer is island-local.** Wires never leave a node;
- nodes exchange **envelopes** (messages as values), never heaps.

```
Island = { id, book table, documents, membrane, nodes }
Node   = { full replica: own heap + version DAG per document }   (island/cluster.bend)
```

Any node holding a `Replicate` cap is a replica and reduces everything. 2–3 of them are **IO nodes** (writers). Reduction needs no node in particular: confluence (`01` §6) makes it the same everywhere. Only IO is tied to a node.

2. NO LEADER, NO CONSENSUS: ORDER LIVES IN THE DATA

Confluence makes one reduction deterministic. It does not make different messages commute. v2 never agrees on a total order. Instead it keeps the **partial order** messages carry themselves:

```
Env{ w, s, parents: [Vid], body }       Vid = V{w, s}   (writer, its own seq)
body = BSend{msg, ios} | BMerge{base} | BPick
```

Relation of A and B	How it is handled	Agreement
B's parents include A (causal)	B waits for A; it applies on A's version	none
concurrent, book proves `law commutes`	applied in arrival order to the one head	none
concurrent, may conflict	both become versions: superposed heads	none, the application resolves later

2.1 VERSIONS ARE DUPS

Each new version is **one DUP node** on its parent's state, the O(1) fork:

```
send   parent p:       !&D{p', v} = state(p);  v ← (@transition v msg)
merge  [a, b], base o:  v ← (@merge o' a' b')        lazy; forced by the next send or read
pick   [x, …]:          v ← x'
```

The parent stays, so a concurrent sibling forks from the same state, and siblings share every subterm neither has forced. The **heads** are the versions no other version names as a parent: a live superposition. A failed send (Stuck, fuel, heap) still makes a version, equal to its parent, so children can attach deterministically.

2.2 WHY REPLICAS AGREE WITHOUT TALKING

A version's state depends only on its parent's state and its body. The heads are "every version not named as a parent". So two replicas holding the **same set** of envelopes have the same heads with the same states, whatever order the envelopes arrived in, duplicates included (gate D1). The rules:

- **causal buffer:** an envelope whose parents (and merge base) are not all known waits. Each application drains the buffer again;
- **dedup:** a known `V{w, s}` is dropped (retries, gossip echoes);
- **canonical order:** heads are kept sorted by `(w, s)`.

No hashing happens on this path.

2.3 COMMUTATIVE BOOKS

If the book carries a proven `law commutes` (`03` §7), envelopes apply in arrival order to a single version: no fork, no buffer (gate D3). The proof is checked by the official Bend 2 frontend at compile time. The runtime trusts the `~commutes` marker exactly as far as it trusts the compiler (`07`).

2.4 RESOLVING

The application decides, at a time it chooses, by writing an envelope:

- `BMerge{base}` with parents `[a, b]` runs the book's own `merge(base, a, b)`, the state-level twin of `merge3` for code (`04` §6). Without a `merge` def it is Denied (5) and behaves as a pick of `a`;
- `BPick` with parents `[x, …]` keeps `x`'s state (one DUP).

Both are ordinary versions, so concurrent resolutions are themselves just more heads.

2.5 FAST PATH: AN OPTIONAL SEQUENCER

A sequencer (as in `bend-game`) is just one writer whose envelopes everyone builds on. While it is up there is one head. When it is down or slow, other writers keep writing: their envelopes become concurrent heads, and nothing waits. A replacement sequencer (any node with the cap) resolves the open heads first. Split brain is harmless: two sequencers make two heads. The sequencer stamp itself is not implemented yet (§9).

2.6 WORLD LABELS: THE INET-NATIVE STORE (`ISLAND/WORLDS.BEND`)

The same delivery rules (§2.2) and the same heads (§2.3), with one superposed state per document instead of one DUP-forked instance per version:

```
message v:  !&Lv{s0, s1} = S;  S ← &Lv{ ~fst(@transition s0 m), s1 }
merge v:    S ← &Lv{ @merge(π_o s0, π_l s0, π_r s0), s1 }
pick v:     S ← &Lv{ π_l s0, s1 }
```

- Each message is applied **once**, at delivery, lazily, in every world at once: one world label, one DUP, one SUP and three app nodes. Side 0 of `Lv` is "v happened", side 1 is "v did not".
- A version is a set of messages, i.e. a world. Reading it installs one collapse dup per label (in the set → 0, else 1) and reduces only that world. Labels of independent messages commute, so worlds that never diverge share their work.
- The host keeps only what the net cannot know: dedup, the causal buffer, which sets are heads, and vid → label.
- `settle` (one head left, at an acknowledged frontier) collapses every label permanently and materializes the state.
- Limits: pure books only (a lazily applied message cannot replay recorded IO; IO books use the DAG store); reads are rolled back, so until `settle` a read redoes the lazy work of its world.

Both stores pass the same gates (`tests/e2e_cluster.bend` DAG, `tests/e2e_worlds.bend` worlds) with identical heads and StateHashes.

2.7 PER-PAIR INDEPENDENCE

`def independent(x, y) -> Bool` plus a proven `law independent_ok` (`Indep(independent(x, y), s, x, y)`, the swap law when True; `03` §7). A head is a consistent set: downward closed, and every two concurrent messages in it independent. A message joins every head whose extra messages it commutes with, and seeds a new set grown greedily (canonical order) from concurrent independent messages. Heads contained in others are dropped. Gate D8 checks all 24 arrival orders of a 4-message mix. The greedy step is not proven to give every maximal consistent set in general.

3. IO NODES

Only a writer performs IO. It runs its own send on a scratch version. Each `NeedIO` (status 4, the receipt's `note` = op) asks the host, and the answer is appended to the envelope's `ios`. Replicas apply the envelope with the recorded answers and never call the host (`02` §8). The reference host is deterministic (clock for op 0, a keyed random word otherwise). A real node substitutes the OS clock and RNG at the host boundary.

If an IO node dies, its chain ends and nobody waits for it: there is no quorum and no lease (gate D2). A client resumes through another writer, naming the last version it saw. IO with external side effects (e.g. an email) is at-least-once if a writer dies between performing and publishing.

4. CAPABILITIES (`MEMBRANE/CAP.BEND`)

```
Cap  = { token, holder: Who{island, id}, island, inst, rights, linear, epoch, state, parent }
Right = Send 0 | Subscribe 1 | Fork 2 | Update 3 | Grant 4 | Replicate 5 | Registry 6 | Io(op) 100+op
state = live 0 | draining 1 | dead 2 | used 3
```

- Tokens are `H(tok.2 ‖ island secret ‖ counter)`. Terms hold only handles.
- `grant` attenuates (rights ⊆ parent's) and records the parent. `revoke` kills a cap and its descendants.
- `epoch_bump` sets older caps to draining; `refresh` trades them for current ones; `drain_end` kills the rest.
- `isolated(X, Y)` searches the cap graph, counting nodes that hold `Replicate` as inside an island. It returns None, or the caps of a witness path.

5. READING AND SUBSCRIPTIONS

`heads(doc)` returns every head with its state, each read in a rolled-back transaction (reads never change heap layout or later fuel). `observe` returns the heads only to a holder of a `Subscribe` cap for the instance (gate D5). The owner's `view` def (`03` §7) is the projection for subscribers. Gossip/streaming transport is outside the kernel: anything that eventually delivers every envelope to every live node works.

6. HASHING (ALL OFF THE REDUCTION PATH)

Event	Hashed
compile / register	`DefHash`, `BookHash` (`03` §6)
send / reduction / delivery	**never**
audit / checkpoint	`StateHash = H(dag.2 ‖ (w, s, canonical state)*)` over the heads; labels and instance ids canonicalized away

7. MEMORY

Every version is pinned while a writer may still name it. `prune` forgets every non-head version; its heap cells become garbage reclaimed by compaction (`island/checkpoint.bend`: restore(take)). An envelope naming a pruned version waits forever, so prune only at a frontier every writer has acknowledged.

Versions and the seen set are persistent maps (`Map<&2, …>`, O(log n)); each version's history is a shared map plus a shared cons list. Only concurrent messages walk history, at most `WINDOW` (64) items per head. Throughput is not measured yet (B7).

8. FAILURE AND RECOVERY

Event	Behaviour
an IO node dies	its chain ends; every other node continues (D2)
replicas partitioned	each side keeps writing; healing = exchanging envelopes; overlapping writes are heads
a fresh / restarted replica	takes any peer's envelope log, in any order (`Cl.revive`)
divergence (bug, bit flip)	caught when audits compare `StateHash`

9. GATES

Gate	Check	State
D1	same envelopes, any arrival order (duplicates, child before parent) → same heads and StateHash	`tests/e2e_cluster.bend` **PASS**
D2	kill an IO node: nobody blocks; a revived replica catches up from a peer's log	PASS
D3	`law commutes` book: concurrent writers, two orders, one head, no fork	PASS
D4	`BMerge` with the book's `merge` equals the hand-merged oracle (vault: 100+50−30+5)	PASS
D5	observe with Subscribe cap ok, Send-only cap denied	PASS
D6	TCP nodes, one replica per process (`net/`)	not started
D7	sequencer fast path and takeover (`Cl.sequence` ; a writer merges open heads, then sequences on)	PASS (both stores)
D8	per-pair independence: 24 arrival orders → same two heads; merge of the heads	PASS (both stores)
W1	worlds: one superposed state, one label per message	PASS
W2	worlds: `settle` collapses every label, state unchanged, sequencing continues	PASS

G.59 · `bend-runtime/hvm5/version2/spec/06-formats.md`

06 — Formats

Status: normative, HVM5 v2. Every byte format and hash domain in one place.

1. `.ic` TEXT (HUMAN-READABLE, HOMOICONIC)

The text form of templates, quotes and debugging output. Grammar = `01` §1.

```
book  ::= def*
def   ::= "@" name "=" term
name  ::= [A-Za-z_][A-Za-z0-9_.]*
ctor  ::= "#" name "{" (term ("," term)*)? "}"
match ::= "~" term "{" ("#" name ":" term)* "}"
```

`parse ∘ print = id` on normal forms (gate X1).

2. SERIAL (CANONICAL BYTES OF A CLOSED NORMAL FORM)

Pre-order. One tag byte per node. Integers are LEB128.

Tag	Node	Payload
0x00	ERA	—
0x01	VAR	de Bruijn level
0x02	LAM	body
0x03	APP	fn, arg (only inside templates; normal forms have none at head)
0x04	SUP	label, a, b
0x05	DUP	label, value, body (binds 2 levels)
0x06	CTR	type id, ctor id, fields…
0x07	MAT	type id, value, arms…

Tag	Node	Payload
0x08	NUM	u32
0x09	OP2	op, a, b
0x0A	SWI	value, z, s
0x0B	REF	DefHash (32 bytes) in quotes/messages; def index in templates
0x0C	CAP	handle index

Canonicality: labels are renumbered 0,1,… in order of first occurrence, and binder levels are assigned in pre-order. `IOR` and `Stuck` are not serializable.

3. HVM5.CORE (COMPILER → BACKEND)

```
magic "HVMC" · u16 version=2 · compiler string
types : vec{ name, vec{ ctor name, arity } }          sorted by name
defs  : vec{ name, arity, pure:u8, ho-bits, Expr }    sorted by name
exports: { init, transition, migrate?, view?, state, msg, reply }
Expr  : tag byte + fields, same tag style as §2 plus 0x10 Let, 0x11 Io
```

Also exposed to Bend 2 as `type Core.Expr is Data: …` for the homoiconic loop (`04` §1).

4. RECORDS

```
Msg        { from: IslandId, token: 32B, nonce: u64, fuel: u64, heap: u64, replyTo?: token, bytes: Serial }
Env        { w: u32, s: u32, parents: [Vid], body }          Vid = (w, s)   (05 §2)
             body = BSend{msg: Serial, ios: [u32]} | BMerge{base: Vid} | BPick
LogEntry   (legacy, island/log.bend: per-op records; not on the v2 ordering path)
Checkpoint { island, seq, epoch, states: sorted [(instance, StateHash)],
             books: sorted [(instance, BookHash)], prev: CheckpointHash, sigs: [(node, sig)] }
Event      { heads: sorted [(Vid, view bytes)], StateHash }
Receipt    { status (0 Done 1 Stuck 2 OutOfFuel 3 OutOfHeap 4 NeedIO 5 Denied 6 StaleEpoch
              7 NoSuchInstance 8 Conflict 9 TypeChange 10 NoSuchBranch 11 Unavailable),
             fuel, reply, shadow: [reply per branch], note (NeedIO: op; branches dropped: 1) }
```

5. HASH DOMAINS

`H` is SHA-256 in pure Bend 2 (`src/crypto/sha256.bend`, FIPS vectors pass). BLAKE3 is optional and not implemented.

Domain prefix	Hashes
`hvm5.def.2\0`	def template + referenced DefHashes
`hvm5.scc.2\0`	mutually recursive group
`hvm5.book.2\0`	sorted (name, DefHash) + manifest
`hvm5.state.2\0`	`Serial` of an instance's normalized state
`hvm5.dag.2\0`	heads: (w, s, canonical state)* (05 §6)
`hvm5.log.2\0`	legacy log entries
`hvm5.ckpt.2\0`	checkpoint record without sigs
`hvm5.msg.2\0`	message bytes (for signatures and dedup)

No hash is ever accepted as a capability.

G.60 · `bend-runtime/hvm5/version2/spec/07-trust.md`

07 — Trust, invariants, threats

Status: normative, HVM5 v2.

1. TRUSTED COMPUTING BASE

Component	Trusted for
BendRT + kernel	rules, fuel and heap accounting, rollback, label discipline
membrane	tokens, attenuation, linearity, epochs, IO rights
version DAG	causal buffer, dedup, heads (no ordering layer to trust: `05` §2)
host crypto	SHA-256 (pure Bend); Ed25519 at the host boundary, not implemented
`law commutes`	checked by the official frontend; the runtime trusts the `~commutes` marker
official Bend 2 frontend + `lower`	type soundness of admitted books

Untrusted: contract code, messages, clients, subscribers, other islands, grafted bytes.

2. INVARIANTS

1. Terms cannot construct tokens. Handles index only the holder's own table.
2. Attenuation only: passed rights ⊆ sender's rights.
3. Linear caps are used at most once.
4. Every send is bounded by fuel and heap budgets.
5. Sends and promotes are atomic (write log; one log entry).
6. Epochs are monotone. Stale-epoch operations are rejected.
7. `Stuck` is safe. It rolls back and never crashes the island.
8. Equal `DefHash` means identical code.
9. The only non-pure inputs are logged IO results.
10. No user-visible superposition in contract books (`01` §7.3).

3. THREATS

Threat	Mitigation	Residual
exponential DUP / fuel bomb	fuel + heap budgets	—
label collision to annihilate foreign SUPs	label discipline, reserved runtime range	proof obligation L2
malicious `Serial`	total decoder; structural type check before graft; `Stuck` → rollback	compiler bugs
code-carrying message abuse	receiver budgets; DefHash fetch from the registry only	—
replay	dedup by `(writer, seq)`	a writer reusing a seq for different content: first arrival wins per replica (needs signed envelopes)
stale epoch after promote	CAS	—
reordering network / client	order is the parents in the envelope; arrival order is irrelevant	—
false `commutes`	frontend proof check	compiler bugs

Threat	Mitigation	Residual
version pinning (memory)	`prune` at an acknowledged frontier	unbounded until pruned
divergent replica	StateHash audits over canonical heads	until the next audit
subscriber leaks	`view` projection limits exposure	copying cannot be prevented
operator reads heap	out of scope	needs TEE
cross-domain hash reuse	domain prefixes (`06` §5)	—

4. PROOFS

`LAWS.bend` states L1–L6 (`01` §8). `PROOF.bend` must be green before any release. It covers the rules, the label discipline, canonical quoting, projection correctness, merge3 and the cap-graph isolation model. It does **not** cover BendRT's generated C, the OS, the network or cryptography. Necessary, not sufficient.

G.61 · `bend-runtime/hvm5/version2/spec/attic/05-islands-leader.md`

05 — Islands: partitioning, replication, capabilities, subscriptions

> **Superseded** (2026-10-23) by `../05-islands.md` : the leader-ordered log was replaced by the leaderless version DAG. Kept for the record; not normative.

Status: normative, HVM5 v2.

1. WHY ISLANDS

A local interaction costs nanoseconds. A network round trip costs about 10^4–10^5 times more. A wire that crosses machines turns optimal reduction into network-bound reduction and makes scheduling global. Therefore:

- **every pointer is island-local;**
- islands exchange **values** (quoted normal forms), never wires;
- an island is an actor whose behaviour is an IC term.

```
Island = { id, heap, bookTable, instances, ports, membrane, log, checkpoint, replicas[3] }
```

An instance lives in exactly one island. Moving it means quote, graft in the target, redirect its ports, and erase it in the source, all as one log entry in each island (a two-phase saga).

2. PORTS AND CAPABILITIES

```
Port  = { instance, entry: Name }                    which exported def receives
Cap   = { token: 32 random bytes, port, rights, mode: Shared | Linear, epoch }
Right = Send | Subscribe | Fork | Update | Grant | Registry | Io(op)
```

- Tokens exist only in the membrane of the island that owns the port.
- Inside terms, a capability is a **handle**: `#Cap{n}` , an index into the holding instance's own cap table. A forged index refers to the holder's own table, so it can never reach a cap the holder does not have.
- Passing a handle in a message: the membrane mints a fresh token for the receiver with rights ⊆ the sender's (attenuation only).
- `Linear` caps are consumed by use. `Update` is always linear.
- Promote increments the epoch. Old-epoch caps go Draining, then Dead.

3. CROSS-ISLAND SEND

```
sender island X receiver island Y
  quote(msg) → bytes (+ DefHashes it references)
  membrane: cap valid? rights ∋ Send? budget?
  ─ Msg{from, token, bytes, fuel, heap, replyTo?, nonce} ─►
                                                  leader: dedup nonce, append log, seq
                                                  type-check bytes against port's Msg type
                                                  fetch missing DefHashes from registry
                                                  replicas: graft → reduce → commit | rollback
  ◄─ Reply{seq, bytes} or Receipt{seq, status, fuel} ─
```

- **Code can travel.** A message may contain closures. Its `REF`s are DefHashes, so the receiver fetches them, content-addressed like Unison. The receiver's fuel and heap budgets bound what they can do.
- **Structural check before graft.** First-order shape against the port's `Msg` type. A mismatch gives `Denied` with no reduction.
- Laziness and sharing do not cross islands. That cost is paid only at the boundary.

4. REPLICATION AND ORDERING

- Three replicas per island. The leader is the lowest live id (Raft by default; the log is pluggable, e.g. BFT).
- The leader orders sends. Replicas apply them in log order. A receipt is returned after 2 of 3 have durably appended the entry.
- **Determinism without comparison.** Same checkpoint, same ordered sends and same recorded IO give the same normal forms and the same fuel on every replica (`01` §6). Nothing is compared per send.

5. HASHING IS OFF THE REDUCTION PATH

Event	Hashed	What
compile / register	once	`DefHash`, `BookHash` (`03` §6)
send / reduction	**never**	—
log entry	small	`EntryHash = H(log.2 ‖ prev ‖ entry)`; the entry is the message, never the heap
checkpoint every K sends or T s	yes	`StateHash = H(state.2 ‖ quote(state))` for instances touched since the last one
fork / promote	no new hashing	records reference existing BookHashes

Checkpoint protocol.

```
at seq c:
  each replica: quote touched instances (parallel normalize), hash
  sign Checkpoint{island, c, epoch, Map<instance, StateHash>, prev}
  2-of-3 equal → commit; compact heap (graft quotes back; labels renumbered)
  mismatch     → minority replica = CatchingUp; refetch checkpoint; replay
```

K trades verification latency for throughput. A reducer never hashes. A divergent replica is caught within K sends. Strict islands may set K = 1. Benchmark B3 publishes the cost curve.

6. SUBSCRIPTIONS

A holder of a `Subscribe` cap streams a port's events:

```
Subscribe{token, fromSeq} → Event{seq, epoch, kind, bytes, entryHash, leaderSig}
kind ∈ Reply | View | Promote
```

- **The owner chooses what is visible.** `View` events are `quote((@view state))`, where `view` is a def in the book (`03` §7). Only replicas receive raw state. A subscriber never gets more than `view` exposes.
- **Only the leader publishes.** Events are signed. A subscriber may relay only with `Grant`.
- **Verification without state hashing.** Check the signature, the epoch, `seq` continuity and the `entryHash` chain. Check checkpoint signatures when stronger assurance is needed.
- **Epoch change** ends the subscription. Resubscribe with a new-epoch cap from the last seq.
- **Revocation** stops future delivery. Delivered bytes cannot be recalled. Confidentiality against a subscriber is encryption and trust, not capabilities.

Subscriber ≠ replica. A replica holds the heap and reduces. A subscriber holds a stream of views. A gateway holds caps and nothing else.

7. FAILURE AND RECOVERY

Event	Behaviour
one replica down	2/2 quorum; no downtime
leader down	election; term and epoch increment; clients retry with the same nonce
two down	island read-only: last checkpoint + committed log
rejoin	fetch the latest checkpoint (quotes + sigs), graft, replay the tail, verify the next checkpoint, then serve

8. REFERENCE TOPOLOGY

Island	Replicas	Contents
`sys`	n0 n1 n2	system contract: authority rows, registry pointers, island directory
`pub`	n3 n4 n5	public instances
`priv`	n6 n7 n8	private instances
—	n9	gateway: caps only, no heap

`isolated(A, B)` runs over the membrane cap graph across islands. It returns `True`, or a witness path of caps.

9. GATES

Gate	Check
N1	10 real processes over TCP; elections; log replication
N2	cross-island send/reply; code-carrying message fetches defs by hash
N3	kill a replica under load: zero failed sends
N4	rejoin from checkpoint + tail replay; next checkpoint matches
N5	injected divergence is detected at the next checkpoint
N6	subscribe without a cap → denied; stale epoch → denied; view-only never sees raw state
N7	B3: throughput at K = 1, 100, 10 000
N8	`isolated(pub, priv)` = True; after a grant, the witness path is returned

G.62 · bend-runtime/hvm5/version2/maps/00-map.md

Map

CONCEPT → SPEC → MODULE

Concept	Spec	Module
IC terms and rules	01 §1–2	src/kernel/rules.bend
labels, soundness boundary	01 §7	src/kernel/book.bend , src/compiler/lower.bend
fuel, canonical strategy	01 §3, 02 §4	src/kernel/whnf.bend
heap, write log, budgets	02 §2, §5–6	src/kernel/heap.bend
quote, graft, projection	01 §4, 02 §7	src/kernel/quote.bend
--emit-core , core	03 §2, 06 §3	upstream patch, src/compiler/core.bend
lowering	03 §3	src/compiler/lower.bend
DefHash, BookHash	03 §6, 06 §5	src/compiler/hash.bend
instance, send, live branches	00 §2, 04 §5	src/instance/{island,branch}.bend
fork, test, merge3, promote	04	src/instance/{fork,test,merge,promote}.bend
caps, rights, epochs	05 §2	src/membrane/cap.bend
IO boundary	02 §8, 05 §3	writers in src/island/version.bend (author , with_io)
version DAG: envelopes, causal buffer, dedup, heads, merge/pick, commutes, prune	05 §2, §7	src/island/version.bend
world-label store: one superposed state, label per message, settle	05 §2.6	src/island/worlds.bend , src/island/cluster_worlds.bend
per-pair independence	05 §2.7, 03 §7	src/island/{version,worlds}.bend , contracts/kv.bend
leaderless mesh: nodes, kill, revive, observe	05 §3–5, §8	src/island/cluster.bend
checkpoints, compaction	02 §2, 05 §6	src/island/checkpoint.bend
legacy op records (not the ordering path)	06 §4	src/island/log.bend
leader-based draft (attic)	attic/05-islands-leader.md	src/island/attic/cluster_leader.bend
merge / law commutes ABI	03 §7	tools/emit-core.ts , contracts/{flags,vault/merge}.bend
TCP and disk	05 §9 D6	net/ (host boundary, not started)

MODULE DEPENDENCIES

```
kernel/ptr → kernel/heap → kernel/book → kernel/rules → kernel/whnf → kernel/norm → kernel/quote
compiler/core → compiler/lower → compiler/hash            (lower emits kernel templates)
instance/* → kernel/*, compiler/hash
membrane/* → instance/*
island/* → membrane/*, instance/*, kernel/quote
net/ (host) → island/*
```

No module imports TypeScript or JavaScript. The only host code is BendRT itself, the `net/` adapter and crypto, all at the boundary.

V1 → V2

v1 artifact	v2 fate
`src/hvm5/eval.bend` (tree/name evaluator)	replaced by `kernel/`; kept as v1 evidence
postfix `.inet` package	replaced by `Serial` + `.ic` text
`src/bendc.bend` (shape parser)	retired; `--emit-core` + `lower`
`tools/emit-inet.ts`	migration aid until C1; then moved to an attic
`src/sim.bend`	replaced by real islands; its scenarios become N-gate tests
`PROOF.bend` / `LAWS.bend`	rewritten for L1–L6

G.63 · `bend-runtime/hvm5/version2/plan/00-plan.md`

HVM5 v2 plan

Order is fixed: **kernel → compiler → code as data → membrane → islands → benchmarks.** A milestone ends when its gates are green on real code. Pure simulations are allowed only as `LAWS.bend` models, never as evidence for a gate.

M1 — IC KERNEL (CRITICAL PATH)

Spec: `01`, `02`. Files: `src/kernel/{ptr,heap,book,rules,whnf,norm,quote}.bend`, `tests/kernel/*.bend`, `bench/b1.bend`.

1. `ptr` + `heap` + write log; unit tests.
2. Rules APP/DUP/MAT/SWI/OP2/ERA. Then DUP-LAM and DUP-SUP'.
3. `whnf` canonical strategy with fuel; `normalize`.
4. `quote` / `graft` / $\pi_{i^{\llcorner}}$; `.ic` printer and parser.
5. HVM4 cross-check corpus (Church numerals, duplicated lambdas, Scott lists).

Exit: K1–K8. **Nothing downstream is claimed until K2 (DUP-LAM) passes.**

M2 — COMPILER

Spec: `03`. Files: `src/compiler/{core,lower,hash}.bend`, upstream patch `bend --emit-core`.

1. Pin a Bend 2 version and write the `--emit-core` patch (offered upstream).
2. `core.bend` decoder → `Core.Expr`.
3. `lower.bend`: linear matches, affinity/labels, ids, refs.
4. `hash.bend`: DefHash/SCC/BookHash (BLAKE3 through the host boundary).
5. Corpus C3 against BendRT.

Exit: C1–C7. Move `../version1/tools/emit-inet.ts` to an attic (files are never deleted).

M3 — CODE AS DATA

Spec: `04`. Files: `src/instance/{book,instance,fork,test,merge,promote}.bend`.

Exit: F1–F8, plus the vault and Bob demo from `04` §9 running end to end on one island.

M4 — MEMBRANE

Spec: `05` §2, `07`. Files: `src/membrane/{cap,budget,io}.bend`.

Exit: invariants 1–10 of `07` tested by an adversarial corpus.

M5 — ISLANDS

Spec: `05`. Files: `src/island/{version,cluster,checkpoint}.bend`, `net/` host adapter (TCP gossip + disk, at the host boundary).

No leader and no consensus. Envelopes `(writer, seq, parents)`, a causal buffer, dedup, a version DAG with one DUP per version, superposed heads, resolution by the book's `merge` or a pick, and a commutative path for books with a proven `law commutes`.

1. `version.bend + cluster.bend` (in-process mesh). **Written, checks.**
2. `tests/e2e_cluster.bend`, gates D1–D5. **PASS.**
3. `net/`: one replica per process, gossip over TCP (D6).
4. Sequencer fast path and takeover (D7); per-pair commutation (D8).

Exit: D1–D8.

M6 — BENCHMARKS

Id	Workload	Metric
B1	kernel micro (DUP-LAM, Church 2^{2^2}, list sort)	interactions/s vs HVM4 C
B2	counter sends, one island	sends/s, p50, p99
B3	vault sends, 3 replicas, K = 1/100/10 000	cost of hashing
B4	fork cost vs state size (1 KB … 1 GB)	flat = O(1)
B5	superposed test vs two runs	sharedRatio
B6	cross-island ping-pong	latency
B7	envelopes/s per replica, 2–3 IO nodes, commuting vs conflicting	throughput, heads, memory

RISKS

Risk	Effect	Mitigation
BendRT array ops too slow for a heap kernel	low interactions/s	measure in M1 week 1; if > 3× off HVM4, run the same spec on HVM4's C kernel via FFI and keep the Bend 2 kernel as the executable spec
`--emit-core` not accepted upstream	coupling	keep a pinned patch; the core format is ours
superposed testing shares little in practice	weaker headline	B5 decides; fork/merge/promote do not depend on it
λ-label discipline too restrictive	some higher-order code rejected	first-order contracts are unaffected; widen with a proof
24-bit label exhaustion	forced checkpoints	per-checkpoint renumbering
many concurrent heads on non-commuting state	work and memory grow with concurrency	`commutes`, per-pair independence (D8), sequencer fast path (D7), `prune` at frontiers
version lookup is a list	O(versions) per delivery	a map keyed by `(w, s)`; measure first (B7)

DEFINITION OF DONE (FINAL HVM5)

1. A Bend 2 contract compiles via `--emit-core` → `lower`, without any second frontend.
2. It runs on the Bend 2 IC kernel (BendRT native) with all rules and deterministic fuel.
3. Fork is O(1). Superposed testing produces projections equal to separate runs. merge3 and promote are atomic with draining.

4. Ten processes, three islands × 3 replicas: failover with zero failed sends, catch-up from checkpoint, capability-gated subscriptions, no hashing on the reduction path.
5. `PROOF.bend` green. B1–B6 published with hardware and commit.

G.64 · bend-runtime/hvm5/reference/hvm5-mini/main.bend

HVM5 in Bend

==========

A port of hvm5.c: an Interaction Calculus evaluator with the same

syntax, semantics and interaction counting as the C reference.

Term ::=

| x variable

| &{} eraser

| &L{a,b} superposition

| ! &L{a,b} = v; k duplication

| λx.f lambda

| λ- f drop lambda

| (f x) application

| () unit

| λ() f unit eliminator

| #0{x} left constructor

| #1{x} right constructor

| λ{#0:f; #1:g} constructor eliminator

| #(a,b) pair

| λ! f pair eliminator

| @ref reference

| @ref(a,b,..) reference applied to arguments

Terms are Bend constructors, so term memory is Bend's own: erasing a

term is dropping it. A run of #0/#1 is one word: 24 bits and their

count, outermost bit lowest, as hvm5.c packs them; a run over () is

the word alone (Won), so a bit chain costs no heap at all.

The env is an Array of cells indexed by binder id. A lambda's cell

holds its argument (Sub) once applied; a dup's cell holds the value

until a side asks (Sha, or Haf when the other side is dead), then the twin's copy. A cell read by its variable goes to the free list: the cell stores the old head (Num), and the words fh and nx (the head, the next fresh id) ride the loop. A dropped term is walked so its cells follow (hvm5.c's collect): an erased argument, a loser branch, a dead dup side's copy. The book is an Array of templates (Tpl, the parser's output) by definition index, the parser numbering names and binders as it meets them. A reference moves its template out of the book, copies it in place and writes it back: the copier (inst) only reads the template, so the compiler lends it and no node is rebuilt. A binder's cell hands its occurrence to the copy and a fresh cell takes the binder. Under fusion, hvm5.c's L_RFO loop, the copier runs the head of the template against the pending arguments instead of copying it: a lambda or a drop takes an argument, a unit or pair eliminator takes a unit or a pair, a constructor eliminator takes a bit and goes on with the branch it selects, an application pushes its argument's copy; a root eliminator meeting an argument that is no bit waits for its whnf (K_DISP). Only what the arguments cannot decide gets copied.

The evaluator is one loop over the env, the book, a mode, a stack and the registers. The mode is a term to evaluate (Go), a whnf to deliver to the top frame (Kn) or a stuck term unwinding the stack (Sk: hvm5.c dies here). The stack is hvm5.c's: a pending argument, an eliminator waiting for its scrutinee, a dup cell waiting for its value, a deferred @ref. Strong normalization runs in the same loop: an NfK frame normalizes the children of the whnf it receives, and a frame per node shape rebuilds it. Every helper answers the next state; the loop matches the mode and the top frame.

The parser is a continuation monad over the parser state, in do-notation: a primitive is a def over the continuation and the state. An error prints and stops: nothing after it runs. A form that looks ahead before it reads more terms resumes as a phase (Ph) of term, the one recursive parser.

Names and labels are codes, as in hvm5.c: six bits per character of the display alphabet, first character highest. The program to run is the src constant, since a Bend binary takes no arguments; the output is hvm5 -s's. A literal holds λ as one code point (955); the output writes it as its two UTF-8 bytes.

```
import Base
```

Types

=====

Tpl: a template, the parser's output: the syntax with its binders as ids. Dup/Dux (both sides live, one side live) exist here only: a copy turns them into a cell.

```
type Tpl is Data: TVar{+i: U32} TCo0{+l: U32, +i: U32} TCo1{+l: U32, +i: U32} TEra{} TSup{+l: U32, a: Tpl, b: Tpl} TDup{+a: U32, +b: U32, v: Tpl, k: Tpl} TDux{+a: U32, v: Tpl, k: Tpl} TLam{+i: U32, f: Tpl} TDrp{f: Tpl} TApp{f: Tpl, x: Tpl} TOne{} TUse{f: Tpl} TPak{+p: U32, t: Tpl} TWon{+p: U32} TTup{a: Tpl, b: Tpl} TGet{f: Tpl} TMat{f: Tpl, g: Tpl} TRef{+n: U32}
```

Term: a runtime term. Abs and Nam appear in normal forms only: a lambda shown with its display name.

```
type Term is Data: Var{i: U32} Co0{+l: U32, i: U32} Co1{+l: U32, i: U32} Era{} Sup{+l: U32, a: Term, b: Term} Lam{+i: U32, f: Term} Drp{f: Term} Cal{f: Term, x: Term} One{} Use{f: Term} Pak{+p: U32, t: Term} Won{+p: U32} Tup{a: Term, b: Term} Get{f: Term} Mat{f: Term, g: Term} Ref{+n: U32} Abs{+n: U32, f: Term} Nam{+n: U32}
```

Cell: an env slot: a substitution, a dup value not yet copied (both sides live, or the twin dead), a free-list link, nothing, a dup side dropped while its value is under evaluation.

```
type Cell is Data: Sub{t: Term} Sha{v: Term} Haf{v: Term} Num{+n: U32} Nul{} Ded{}
```

Sh1, Sh2: the shape of a node with one child, with two: what a dup copies, an eliminator a sup splits, or inst builds over the copied children.

```
type Sh1 is Data: SUse{} SGet{} SDrp{} SLam{+n: U32} SPak{+p: U32}
```

type Sh2 is Data: STup{} SMat{} SApp{} SSup{+l: U32}

Kont: the stack. Arg: a pending argument. Usk, Mtk, Gtk: an

eliminator waiting for its scrutinee. Dpk: dup cell i (label l)

waiting for its value, side sd asking, fl: is the twin alive. Rfk:

a deferred eliminator @ref. NfK: the whnf arriving gets normalized

deeper. The N-frames rebuild a node from its normalized children.

type Kont is Data: Arg{x: Term, k: Kont} Usk{f: Term, k: Kont} Mtk{f: Term, g: Term, k: Kont} Gtk{f: Term, k: Kont} Dpk{+l: U32, +i: U32, +sd: U32, +fl: U32, k: Kont} Rfk{+r: U32, k: Kont} NfK{k: Kont} NSupA{+l: U32, b: Term, k: Kont} NSupB{+l: U32, a: Term, k: Kont} NTupA{b: Term, k: Kont} NTupB{a: Term, k: Kont} NMatA{g: Term, k: Kont} NMatB{f: Term, k: Kont} NPak{+p: U32, k: Kont} NUse{k: Kont} NGet{k: Kont} NDrp{k: Kont} NAbs{+n: U32, k: Kont} End{}

Mode: a term to evaluate, a whnf to deliver, a stuck term unwinding

the stack.

type Mode is Data: Go{t: Term} Kn{w: Term} Sk{w: Term}

S: the machine: the env, the book, the mode, the stack, the

interaction count, the free-list head, the next fresh id.

type S is Type: S{env: Array<Cell>, book: Array<Tpl>, m: Mode, k: Kont, +c: U32, +fh: U32, +nx: U32}

Out: what the loop answers: the normal form and the interaction

count, or a request to its driver: expand definition r, or resolve

a deferred eliminator @r with the whnf w of its scrutinee (K_DISP).

type Out is Type: Fin{env: Array<Cell>, book: Array<Tpl>, t: Term, +c: U32} Exp{env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +r: U32} Dsp{env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +r: U32, w: Term}

FK: the frames of a collect walk: the sibling still to free. FS: the

walk's state; Fr: its answer, the env and the free-list head.

type FK is Data: FEnd{} FNxt{t: Term, k: FK}

type FS is Type: FS{env: Array<Cell>, t: Term, k: FK, +fh: U32}

type Fr is Type: Fr{env: Array<Cell>, +fh: U32}

Al: fresh cells taken: the env, their ids, the allocator words.

type Al is Type: Al{env: Array<Cell>, +n: U32, +fh: U32, +nx: U32}

type Al2 is Type: Al2{env: Array<Cell>, +n: U32, +p: U32, +fh: U32, +nx: U32}

type Al3 is Type: Al3{env: Array<Cell>, +n: U32, +p: U32, +r: U32, +fh: U32, +nx: U32}

Fz: whether a copy fuses leading binders with pending arguments:

at the root of a reference (a Mat root defers instead), under its

fused binders, or not at all.

type Fz is Data: FRoot{+r: U32} FLam{} FOff{}

Ins: a copy: the env, the instance, the stack, the interaction

count, the allocator words.

type Ins is Type: Ins{env: Array<Cell>, t: Term, k: Kont, +c: U32, +fh: U32, +nx: U32}

IM: the copier's phase over the template node in hand and its

sibling: copy it; copy it under a one-child shape; copy it and its

sibling under a two-child shape; its sibling, the left copy done;

copy it under a fresh binder cell; copy it as a dup's value and take

a cell, then its body; copy it once a dropped argument is freed;

done (a deferred root).

type IM is Type: IGo{env: Array<Cell>, +fh: U32, +nx: U32, fz: Fz, k: Kont, +c: U32} IUn{sh: Sh1, env: Array<Cell>, +fh: U32, +nx: U32, k: Kont, +c: U32} ITw{sh: Sh2, env: Array<Cell>, +fh: U32, +nx: U32, k: Kont, +c: U32} ILt{sh: Sh2, r: Ins} IAl{al: Al, +i: U32, k: Kont, +c: U32} IDv{+a: U32, +b: U32, env: Array<Cell>, +fh: U32, +nx: U32, k: Kont, +c: U32} IDq{+a: U32, +b: U32, r: Ins} IXv{+a: U32, env: Array<Cell>, +fh: U32, +nx: U32, k: Kont, +c: U32} IXq{+a: U32, r: Ins} IFr{fr: Fr, +nx: U32, k: Kont, +c: U32} IGr{env: Array<Cell>, +fh: U32, +nx: U32, k: Kont, +c: U32} IAg{env: Array<Cell>, +fh: U32, +nx: U32, k: Kont, +c: U32} IAr{r: Ins} IEnd{ins: Ins}

Hc: the syntax form the first two characters of the source open;

HCtr{b} is #0 (b false) or #1.

type Hc is Data: HEra{} HSup{} HDup{} HTup{} HCtr{b: Bool} HRef{} HOne{} HApp{} HMat{} HDrp{} HGet{} HUse{} HLam{} HOth{}

Ws: the source head as whitespace: a blank, a comment, or neither.

type Ws is Data: Sp{} Cm{} No{}

Occ: how a binder occurs: a variable, a dup side (label, cell).

type Occ is Data: OVar{+i: U32} OCo0{+l: U32, +i: U32} OCo1{+l: U32, +i: U32}

Bind: a scoped binder: its name, its occurrence, its use count.

type Bind is Data: Bind{+nm: String, +oc: Occ, +u: U32}

Names: the definition names met so far, each with its index; the

next index.

type Names is Data: Names{m: Map<&2, U32>, +n: U32}

Ent: a parsed definition: its index and its template.

type Ent is Data: Ent{ix: U32, t: Tpl}

St: the parser state: the head form, the source (blanks skipped),

the scope, the next binder id, the names.

type St is Data: St{+h: Hc, src: String, sc: List<&2, Bind>, +vc: U32, ns: Names}

Dp: a dup's sides: both dead, one live (its cell), both live.

type Dp is Data: Dead{} Half{+j: U32} Both{+i: U32, +j: U32}

Ph: the parser phase: a term; a term whose head form is known; a

reference with or without arguments; a call after "(", closed or

not; a call after an argument, with a comma or not; a dup's body.

type Ph is Data: Trm{} PAt{h: Hc} PArg{op: Bool, f: Tpl} PCls{cl: Bool, f: Tpl} PSep{m: Bool, f: Tpl} PDup{d: Dp, v: Tpl}

Constants

==========

the display-name alphabet, hvm5.c's: a name code is base 64 over it

def alphabet() -> String: "_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789$"

"λ" as the two bytes the output writes: the one place a code stands

for a character, since a Char literal is the code point (955) and

IO.print writes one byte per Char

def lam() -> String: SCon{Chr{206}, SCon{Chr{187}, ""}}

the program: hvm5 -s of this file prints the same

def program() -> String: "// Booleans are bit runs over (); @not flips one.\n" ++ "@not = λ{#0: λ- #1{()}; #1: λ- #0{()}};\n" ++ "// Church numeral 2: the function is shared by a dup.\n" ++ "@two = λf. ! &F{f0, f1} = f; λx. (f0 (f1 x));\n" ++ "// 2^2 flips of a bit, applied to both bits at once.\n" ++ "@main = ! &A{f, g} = @two(@two(@not)); &S{(f #0{()}), (g #1{()})};\n"

Pick

====

def pick(-A: Type, z: Bool, a: A, b: A) -> A: match z: case True{}: a case False{}: b

Chr

===

def chr_code(ch: Char) -> U32: Chr{x} = ch x

is code c the character ch

def chr_is(c: U32, ch: Char) -> Bool: U32.is_eq(c, chr_code(ch))

Str

===

the code at position n (0 past the end)

def str_at(s: String, n: Nat) -> U32: match s n: case SCon{Chr{c}, t} 0n: c case SCon{h, t} 1n+p: str_at(t, p) case SNil{} _: 0

the string past n characters

def str_drop(s: String, n: Nat) -> String: match s n: case SCon{h, t} 1n+p: str_drop(t, p) case u _: u

the position of code c in s, or the length of s; z: is the head c

def str_find(s: String, z: Bool, +c: U32, +i: U32) -> U32: match s z: case _ True{}: i case SCon{h, +t} False{}: str_find(t, U32.is_eq(str_at(t, 0n), c), c, (i + 1 : U32)) case SNil{} False{}: i

the digit of code c in the alphabet; 64 when it is not a digit

def alpha(+c: U32) -> U32: +a = alphabet() str_find(a, U32.is_eq(str_at(a, 0n), c), c, 0)

a name character: a digit, or ”-” so a dead dup side reads as a name

def chr_name(+c: U32) -> Bool: U32.is_lt(alpha(c), 64) || chr_is(c, '-')

a name’s code: six bits per character, first character highest

def name_code(s: String, +acc: U32) -> U32: match s: case SCon{Chr{c}, t}: name_code(t, ((acc << 6n : U32) + alpha(c) : U32)) case SNil{}: acc

def ws_new(sp: Bool, cm: Bool) -> Ws: match sp cm: case True{} _: Sp{} case _ True{}: Cm{} case _ _: No{}

the source head as whitespace: a blank or ”~”, or ”//”

def str_ws(s: String) -> Ws: match s: case SCon{Chr{+c}, t}: ws_new(U32.is_le(c, chr_code(' ')) || chr_is(c, '~'), chr_is(c, '/') && chr_is(str_at(t, 0n), '/')) case SNil{}: No{}

the source head inside a comment: a newline ends it

def str_cm(s: String) -> Ws: match s: case SCon{Chr{c}, t}: +nl = chr_is(c, '\n') ws_new(nl, Bool.not(nl)) case SNil{}: No{}

blanks and comments dropped from the head; z classifies the head

def str_skip(s: String, z: Ws) -> String: match s z: case SCon{c, +t} Sp{}: str_skip(t, str_ws(t)) case SCon{c, +t} Cm{}: str_skip(t, str_cm(t)) case u _: u

def str_word.go(c: Char, p: String & String) -> String & String: (w, u) = p (SCon{c, w}, u)

the word at the head, and the rest; z: is the head a name character

def str_word(s: String, z: Bool) -> String & String: match s z: case SCon{c, +t} True{}: str_word.go(c, str_word(t, chr_name(str_at(t, 0n)))) case u _: ("", u)

Spec

====

What LAWS.bend measures: the word and the rest re-joined (the lexer

loses nothing), and the ”{” of a string (the printer’s braces).

def word_cat(p: String & String) -> String: (w, u) = p w ++ u

def opens.go(brace: Bool, n: Nat) -> Nat: match brace: case True{}: 1n+n case False{}: n

how many "{" a string holds

def opens(s: String) -> Nat: match s: case SNil{}: 0n case SCon{Chr{c}, t}: opens.go(chr_is(c, '{'), opens(t))

Hc

==

def hc_hash(+d: U32) -> Hc: pick(Hc, chr_is(d, '('), HTup{}, HCtr{chr_is(d, '1')})

def hc_lam(+d: U32) -> Hc: pick(Hc, chr_is(d, '{'), HMat{}, pick(Hc, chr_is(d, '-'), HDrp{}, pick(Hc, chr_is(d, '!'), HGet{}, pick(Hc, chr_is(d, '('), HUse{}, HLam{}))))

the form opened by the two characters c, d

def hc_new(+c: U32, +d: U32) -> Hc: pick(Hc, chr_is(c, '&'), pick(Hc, chr_is(d, '{'), HEra{}, HSup{}), pick(Hc, chr_is(c, '!'), HDup{}, pick(Hc, chr_is(c, '#'), hc_hash(d), pick(Hc, chr_is(c, '@'), HRef{}, pick(Hc, chr_is(c, '('), pick(Hc, chr_is(d, ')'), HOne{}, HApp{}), pick(Hc, chr_is(c, 'λ'), hc_lam(d), H0th{})))))))

St

==

a state over a source: blanks skipped, the head classified

def st_new(+src: String, sc: List<&2, Bind>, vc: U32, ns: Names) -> St: +s = str_skip(src, str_ws(src)) St{hc_new(str_at(s, 0n), str_at(s, 1n)), s, sc, vc, ns}

Parser monad

==========

a parser of an A: it takes the continuation and the state; the last

continuation runs the program

def P(-A: Type) -> Type: (A -> St -> IO(Unit)) -> St -> IO(Unit)

def P.pure(-A: Type, x: A) -> P(A): k => s => k(x, s)

def P.bind(-A: Type, -B: Type, m: P(A), f: A -> P(B)) -> P(B): k => s => m(a => f(a)(k), s)

an error: it prints, and nothing after it runs

def fail(-A: Type, msg: String) -> P(A): k => s => IO.print(msg)

the form at the head

def peek(k: Hc -> St -> IO(Unit), s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s k(h, St{h, src, sc, vc, ns})

n characters consumed

def drop(n: Nat, k: Unit -> St -> IO(Unit), s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s k(Unit{}, st_new(str_drop(src, n), sc, vc, ns))

def name.go(sc: List<&2, Bind>, vc: U32, ns: Names, k: String -> St -> IO(Unit), p: String & String) -> IO(Unit): (w, u) = p k(w, st_new(u, sc, vc, ns))

the name at the head

def name(k: String -> St -> IO(Unit), s: St) -> IO(Unit): St{h, +src, sc, vc, ns} = s name.go(sc, vc, ns, k, str_word(src, chr_name(str_at(src, 0n))))

the name at the head, as a code

```
def label() -> P(U32): do P<U32>: w : String <- name return name_code(w, 0)

def opt.go(hit: Bool, k: Bool -> St -> IO(Unit), s: St, t: String) -> IO(Unit): match hit: case True{}: St{h, src, sc, vc, ns} = s k(True{}, st_new(t, sc, vc, ns)) case False{}: k(False{}, s)
```

an optional character: consumed when present; the answer says which

```
def opt(ch: Char, k: Bool -> St -> IO(Unit), s: St) -> IO(Unit): St{h, +src, sc, vc, ns} = s opt.go(chr_is(str_at(src, 0n), ch), k, St{h, src, sc, vc, ns}, str_drop(src, 1n))
```

an optional character, consumed when present

```
def skip(ch: Char) -> P(Unit): do P<Unit>: hit : Bool <- opt(ch) return Unit{}

def sym.go(hit: Bool, ch: Char) -> P(Unit): match hit: case True{}: P.pure(Unit, Unit{}) case False{}: fail(Unit, "parse error: expected '" ++ SCon{ch, "'"})
```

a required character

```
def sym(+ch: Char) -> P(Unit): do P<Unit>: hit : Bool <- opt(ch) sym.go(hit, ch)
```

the closing brace of a form that opened one

```
def close(br: Bool) -> P(Unit): match br: case True{}: sym('}') case False{}: P.pure(Unit, Unit{})
```

a variable binder pushed onto the scope: its id

```
def bindv(x: String, k: U32 -> St -> IO(Unit), s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s k(vc, St{h, src, Bind{x, OVar{vc}, 0} <> sc, (vc + 1 : U32), ns})

def bindc.go( da: Bool, db: Bool, +l: U32, a: String, b: String, k: Dp -> St -> IO(Unit), s: St ) -> IO(Unit): match da db: case True{} True{}: k(Dead{}, s) case True{} False{}: St{h, src, sc, vc, ns} = s k(Half{vc}, St{h, src, Bind{b, OCo1{l, vc}, 0} <> sc, (vc + 1 : U32), ns}) case False{} True{}: St{h, src, sc, vc, ns} = s k(Half{vc}, St{h, src, Bind{a, OCo0{l, vc}, 0} <> sc, (vc + 1 : U32), ns}) case False{} False{}: St{h, src, sc, vc, ns} = s k(Both{vc, (vc + 1 : U32)}, St{h, src, Bind{b, OCo1{l, (vc + 1 : U32)}, 0} <> (Bind{a, OCo0{l, vc}, 0} <> sc), (vc + 2 : U32), ns})
```

a dup's live sides pushed onto the scope ("-" is a dead side)

```
def bindc(l: U32, +a: String, +b: String, k: Dp -> St -> IO(Unit), s: St) -> IO(Unit): bindc.go(String.eq(a, "-"), String.eq(b, "-"), l, a, b, k, s)

def pop.go(ok: Bool, nm: String, u: U32) -> P(Unit): match ok: case True{}: P.pure(Unit, Unit{}) case False{}: fail(Unit, "non-linear variable: " ++ nm ++ " used " ++ U32.show(u) ++ " times")
```

the innermost binder popped: it was used once

```
def pop(k: Unit -> St -> IO(Unit), s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s match sc: case Bind{nm, oc, u} <> r: pop.go(U32.is_eq(u, 1), nm, u)(k, St{h, src, r, vc, ns}) case Nil{}: k(Unit{}, St{h, src, Nil{}, vc, ns})

def sc_find.go(+n: String, +oc: Occ, +u: U32, hit: Bool, p: List<&2, Bind> & Maybe<&2, Occ>) -> List<&2, Bind> & Maybe<&2, Occ>: match hit: case True{}: (r, m) = p (Bind{n, oc, (u + 1 : U32)} <> r, Some{oc}) case False{}: (r, m) = p (Bind{n, oc, u} <> r, m)
```

the nearest binder named nm, its use counted; done: one was found

```
def sc_find(sc: List<&2, Bind>, +nm: String, +done: Bool) -> List<&2, Bind> & Maybe<&2, Occ>: match sc: case Bind{n, oc, u} <> r: +hit = Bool.not(done) && String.eq(n, nm) sc_find.go(n, oc, u, hit, sc_find(r, nm, done || hit)) case Nil{}: (Nil{}, None{})
```

the occurrence term of a binder

```
def occ_term(oc: Occ) -> Tpl: match oc: case OVar{i}: TVar{i} case OCo0{l, i}: TCo0{l, i} case OCo1{l, i}: TCo1{l, i}
```

def look.go(nm: String, k: Tpl -> St -> IO(Unit), h: Hc, src: String, vc: U32, ns: Names, p: List<&2, Bind> & Maybe<&2, Occ>) -> IO(Unit): (r, m) = p match m: case Some{oc}: k(occ_term(oc), St{h, src, r, vc, ns}) case None{}: IO.print("unbound variable: " ++ nm)

a variable occurrence

def look(+nm: String, k: Tpl -> St -> IO(Unit), s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s look.go(nm, k, h, src, vc, ns, sc_find(sc, nm, False{}))

is the source over

def eof(k: Bool -> St -> IO(Unit), s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s match src: case SNil{}: k(True{}, St{h, SNil{}, sc, vc, ns}) case SCon{c, t}: k(False{}, St{h, SCon{c, t}, sc, vc, ns})

def names_find.old(n: U32, p: Map<&2, U32> & U32) -> Names & U32: (m, ix) = p (Names{m, n}, ix)

def names_find.go(+nm: String, +n: U32, p: Map<&2, U32> & Bool) -> Names & U32: (m, has) = p match has: case True{}: names_find.old(n, Map.get(U32, 0, m, nm)) case False{}: (Names{Map.set(&2, U32, m, nm, n), (n + 1 : U32)}, n)

the index of a name: the next one when the name is new

def names_find(ns: Names, +nm: String) -> Names & U32: Names{m, n} = ns names_find.go(nm, n, Map.has(&2, U32, m, nm))

def index.go(k: U32 -> St -> IO(Unit), h: Hc, src: String, sc: List<&2, Bind>, vc: U32, p: Names & U32) -> IO(Unit): (ns, ix) = p k(ix, St{h, src, sc, vc, ns})

the definition index of a name

def index(w: String, k: U32 -> St -> IO(Unit), s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s index.go(k, h, src, sc, vc, names_find(ns, w))

the name at the head, as a definition index

def ref() -> P(U32): do P<U32>: w : String <- name index(w)

Parser

======

a run word: n bits, w their values (outermost lowest)

def run_word(+n: U32, +w: U32) -> U32: ((n << 24n) .|. w : U32)

bit b pushed outermost onto the run word p

def run_push(+p: U32, +b: U32) -> U32: run_word(((p >> 24n) + 1 : U32), (((p .&. 16777215) << 1n) .|. b : U32))

is the run word full (24 bits)

def run_full(+p: U32) -> Bool: U32.is_eq((p >> 24n : U32), 24)

def term_pak.pak(full: Bool, +b: U32, +p: U32, t: Tpl) -> Tpl: match full: case True{}: TPak{run_word(1, b), TPak{p, t}} case False{}: TPak{run_push(p, b), t}

def term_pak.won(full: Bool, +b: U32, +p: U32) -> Tpl: match full: case True{}: TPak{run_word(1, b), TWon{p}} case False{}: TWon{run_push(p, b)}

#b{x}: the run of x grows when it has room, else a new run starts

def term_pak(b: Bool, x: Tpl) -> Tpl: match x: case TPak{p, t}: term_pak.pak(run_full(p), pick(U32, b, 1, 0), p, t) case TWon{p}: term_pak.won(run_full(p), pick(U32, b, 1, 0), p) case TOne{}: TWon{run_word(1, pick(U32, b, 1, 0))} case y: TPak{run_word(1, pick(U32, b, 1, 0)), y}

a term

@unsafe def term(ph: Ph) -> P(Tpl): match ph: case Trm{}: do P<Tpl>: h : Hc <- peek term(PAt{h}) case PAt{HEra{}}: do P<Tpl>: Unit <- drop(2n) Unit <- sym('}') return TEra{} case PAt{HSup{}}: do P<Tpl>: Unit <- drop(1n) l : U32 <- label() Unit <- sym('{') a : Tpl <- term(Trm{}) Unit <- sym(',') b : Tpl <- term(Trm{}) Unit <- sym('}') return TSup{l, a, b} case PAt{HDup{}}: do P<Tpl>: Unit <- drop(1n) Unit <- sym('&') l : U32 <- label() Unit <- sym('{') a : String <- name Unit <- sym(',') b : String <- name Unit <- sym('}') Unit <- sym('=') v : Tpl <- term(Trm{}) Unit <- sym(';') d : Dp <- bindc(l, a, b) term(PDup{d, v}) case PAt{HTup{}}: do P<Tpl>: Unit <- drop(2n) a : Tpl <- term(Trm{}) Unit <- sym(',') b : Tpl <- term(Trm{}) Unit <- sym(')') return TTup{a, b} case PAt{HCtr{b}}: do P<Tpl>: Unit <- drop(2n) br : Bool <- opt('{') x : Tpl <- term(Trm{}) Unit <- close(br) return term_pak(b, x) case PAt{HRef{}}: do P<Tpl>: Unit <- drop(1n) ix : U32 <- ref() op : Bool <- opt('(') term(PArg{op, TRef{ix}}) case PAt{HOne{}}: do P<Tpl>: Unit <- drop(2n) return TOne{} case PAt{HApp{}}: do P<Tpl>: Unit <- drop(1n) f : Tpl <- term(Trm{}) x : Tpl <- term(Trm{}) Unit <- sym(')') return TApp{f, x} case PAt{HMat{}}: do P<Tpl>: Unit <- drop(2n) Unit <- sym('#') Unit <- sym('0') Unit <- sym(':') f : Tpl <- term(Trm{}) Unit <- skip(';') Unit <- sym('#') Unit <- sym('1') Unit <- sym(':') g : Tpl <- term(Trm{}) Unit <- skip(';') Unit <- sym('}') return TMat{f, g} case PAt{HDrp{}}: do P<Tpl>: Unit <- drop(2n) f : Tpl <- term(Trm{}) return TDrp{f} case PAt{HGet{}}: do P<Tpl>: Unit <- drop(2n) f : Tpl <- term(Trm{}) return TGet{f} case PAt{HUse{}}: do P<Tpl>: Unit <- drop(2n) Unit <- sym(')') f : Tpl <- term(Trm{}) return TUse{f} case PAt{HLam{}}: do P<Tpl>: Unit <- drop(1n) x : String <- name Unit <- sym('.') i : U32 <- bindv(x) f : Tpl <- term(Trm{}) Unit <- pop return TLam{i, f} case PAt{HOth{}}: do P<Tpl>: n : String <- name look(n) case PArg{False{}, f}: P.pure(Tpl, f) case PArg{True{}, f}: do P<Tpl>: cl : Bool <- opt(')') term(PCls{cl, f}) case PCls{True{}, f}: P.pure(Tpl, f) case PCls{False{}, f}: term(PSep{True{}, f}) case PSep{True{}, f}: do P<Tpl>: a : Tpl <- term(Trm{}) m : Bool <- opt(',') term(PSep{m, TApp{f, a}}) case PSep{False{}, f}: do P<Tpl>: Unit <- sym(')') return f case PDup{Dead{}, v}: do P<Tpl>: q : Tpl <- term(Trm{}) return TApp{TDrp{q}, v} case PDup{Half{j}, v}: do P<Tpl>: q : Tpl <- term(Trm{}) Unit <- pop return TDux{j, v, q} case PDup{Both{i, j}, v}: do P<Tpl>: q : Tpl <- term(Trm{}) Unit <- pop Unit <- pop return TDup{i, j, v, q}

the definitions to the end of the source; done: is the source over

@unsafe def defs(done: Bool, es: List<&2, Ent>) -> P(List<&2, Ent>): match done: case True{}: P.pure(List<&2, Ent>, es) case False{}: do P<List<&2, Ent>>: Unit <- sym('@') ix : U32 <- ref() Unit <- sym('=') t : Tpl <- term(Trm{}) Unit <- sym(';') e : Bool <- eof defs(e, Ent{ix, t} <> es)

the definitions of the source

def prog() -> P(List<&2, Ent>): do P<List<&2, Ent>>: e : Bool <- eof defs(e, Nil{})

Pak

===

the run word without its outermost bit

def run_pop(+p: U32) -> U32: ((((p >> 24n) - 1) << 24n) .|. ((p .&. 16777215) >> 1n) : U32)

def pak_pop.go(one: Bool, +p: U32, t: Term) -> Term: match one: case True{}: t case False{}: Pak{run_pop(p), t}

def pak_pop(+p: U32, t: Term) -> Term: pak_pop.go(U32.is_eq((p >> 24n : U32), 1), p, t)

def won_pop.go(one: Bool, +p: U32) -> Term: match one: case True{}: One{} case False{}: Won{run_pop(p)}

def won_pop(+p: U32) -> Term: won_pop.go(U32.is_eq((p >> 24n : U32), 1), p)

def bit0(+p: U32) -> Bool: U32.is_eq((p .&. 1 : U32), 0)

The env

=======

def num(s: Cell) -> U32: match s: case Num{n}: n case _: 0

def cel(s: Cell) -> Term: match s: case Sub{t}: t case _: Era{}

def subst(env: Array<Cell>, i: U32, v: Cell) -> Array<Cell>: Array.set(Cell, env, i, v)

def cov.go(z: Bool, +l: U32, +i: U32) -> Term: match z: case True{}: Co0{l, i} case False{}: Co1{l, i}

the occurrence term of dup side sd (0 = Co0, 1 = Co1)

def cov(+sd: U32, +l: U32, +i: U32) -> Term: cov.go(U32.is_eq(sd, 0), l, i)

def park.go(z: Bool, w: Term) -> Cell: match z: case True{}: Haf{w} case False{}: Sha{w}

a dup value parked back: Haf when the twin is dead (fl = 0)

def park(+fl: U32, w: Term) -> Cell: park.go(U32.is_eq(fl, 0), w)

a fresh cell holding v: the free-list head (its link is the new

head), else the next id (a virgin cell's Nul link reads 0)

def alloc.go(+n: U32, +nx: U32, p: Array<Cell> & Cell) -> Al: (env, s) = p Al{env, n, num(s), nx}

def alloc.at(env: Array<Cell>, +fh: U32, +nx: U32, v: Cell, z: Bool) -> Al: match z: case True{}: alloc.go(nx, (nx + 1 : U32), Array.swap(Cell, env, nx, v)) case False{}: alloc.go(fh, nx, Array.swap(Cell, env, fh, v))

def alloc(env: Array<Cell>, +fh: U32, +nx: U32, v: Cell) -> Al: alloc.at(env, fh, nx, v, U32.is_eq(fh, 0))

def alloc2.fin(+n: U32, al: Al) -> Al2: Al{env, p, fh, nx} = al Al2{env, n, p, fh, nx}

def alloc2.go(v2: Cell, al: Al) -> Al2: Al{env, n, fh, nx} = al alloc2.fin(n, alloc(env, fh, nx, v2))

def alloc2(env: Array<Cell>, +fh: U32, +nx: U32, v1: Cell, v2: Cell) -> Al2: alloc2.go(v2, alloc(env, fh, nx, v1))

def alloc3.fin(+n: U32, +p: U32, al: Al) -> Al3: Al{env, r, fh, nx} = al Al3{env, n, p, r, fh, nx}

def alloc3.go(v3: Cell, al: Al2) -> Al3: Al2{env, n, p, fh, nx} = al alloc3.fin(n, p, alloc(env, fh, nx, v3))

def alloc3(env: Array<Cell>, +fh: U32, +nx: U32, v1: Cell, v2: Cell, v3: Cell) -> Al3: alloc3.go(v3, alloc2(env, fh, nx, v1, v2))

Book

====

def book_put(book: Array<Tpl>, r: U32, t: Tpl) -> Array<Tpl>: Array.set(Tpl, book, r, t)

Collection

==========

A dropped term returns its cells to the free list, as hvm5.c's

collect. A lambda's binder reads Era from now on (its variable may

live in a dup's copy). A dropped dup side leaves the twin sole

(Haf), frees the copy parked for it, or marks a value still under

evaluation (Ded), whose copy the settling side frees.

def free_var(k: FK, +fh: U32, +i: U32, p: Array<Cell> & Cell) -> FS: (env, s) = p match s: case Sub{u}: FS{env, u, k, i} case _: FS{env, Era{}, k, i}

def free_cov(k: FK, +fh: U32, +i: U32, p: Array<Cell> & Cell) -> FS: (env, s) = p match s: case Sha{v}: FS{subst(env, i, Haf{v}), Era{}, k, fh} case Haf{v}: FS{subst(env, i, Num{fh}), v, k, i} case Sub{u}: FS{subst(env, i, Num{fh}), u, k, i} case _: FS{subst(env, i, Ded{}), Era{}, k, fh}

@unsafe def free_run(s: FS) -> Fr: FS{env, t, k, fh} = s match t: case Var{+i}: free_run(free_var(k, fh, i, Array.swap(Cell, env, i, Num{fh}))) case Co0{l, +i}: free_run(free_cov(k, fh, i, Array.swap(Cell, env, i, Nul{}))) case Co1{l, +i}: free_run(free_cov(k, fh, i, Array.swap(Cell, env, i, Nul{}))) case Lam{i, f}: free_run(FS{subst(env, i, Sub{Era{}}), f, k, fh}) case Sup{l, a, b}: free_run(FS{env, a, FNxt{b, k}, fh}) case Cal{a, b}: free_run(FS{env, a, FNxt{b, k}, fh}) case Tup{a, b}: free_run(FS{env, a,

FNxt{b, k}, fh}) case Mat{a, b}: free_run(FS{env, a, FNxt{b, k}, fh}) case Drp{f}: free_run(FS{env, f, k, fh}) case Use{f}: free_run(FS{env, f, k, fh}) case Get{f}: free_run(FS{env, f, k, fh}) case Pak{p, f}: free_run(FS{env, f, k, fh}) case _: match k: case FEnd{}: Fr{env, fh} case FNxt{u, q}: free_run(FS{env, u, q, fh})

def free(env: Array<Cell>, t: Term, +fh: U32) -> Fr: free_run(FS{env, t, FEnd{}, fh})

Nodes

=====

def node1(sh: Sh1, x: Term) -> Term: match sh: case SUse{}: Use{x} case SGet{}: Get{x} case SDrp{}: Drp{x} case SLam{n}: Lam{n, x} case SPak{p}: Pak{p, x}

def node2(sh: Sh2, a: Term, b: Term) -> Term: match sh: case STup{}: Tup{a, b} case SMat{}: Mat{a, b} case SApp{}: Cal{a, b} case SSup{l}: Sup{l, a, b}

Instantiation

=============

inst copies a borrowed template, riding fh/nx, moving each binder's

occurrence out of its cell. The phase (IM) says what to do with the

template node in hand, t, and its sibling, s: copy it; copy it and

wrap the copy in a one-child shape; copy it as the left child of a

two-child shape, then its sibling (ILt holds the left copy); copy it

under a fresh binder cell; copy a dup's value, take its cell, then

copy the body; copy it once a dropped argument is freed. At the root

of a reference (FRoot) and under its fused binders (FLam) a leading

lambda or drop meets a pending Arg frame instead of being copied

(L_RFO), and a root that is a constructor eliminator defers to its

scrutinee (K_DISP): the copy answers the scrutinee under an Rfk

frame (IEnd).

def inst.var(k: Kont, +c: U32, +fh: U32, +nx: U32, p: Array<Cell> & Cell) -> Ins: (env, s) = p Ins{env, cel(s), k, c, fh, nx}

a dup side's occurrence: the cell holds the fresh dup cell's id

def inst.cov(k: Kont, +c: U32, +fh: U32, +nx: U32, +l: U32, z: Bool, p: Array<Cell> & Cell) -> Ins: (env, s) = p Ins{env, cov.go(z, l, num(s)), k, c, fh, nx}

def inst.one(sh: Sh1, r: Ins) -> Ins: Ins{env, t, k, c, fh, nx} = r Ins{env, node1(sh, t), k, c, fh, nx}

def inst.two(sh: Sh2, a: Term, r: Ins) -> Ins: Ins{env, b, k, c, fh, nx} = r Ins{env, node2(sh, a, b), k, c, fh, nx}

a lambda: fused with a pending argument, else under a fresh cell

def inst.lam(env: Array<Cell>, +fh: U32, +nx: U32, fz: Fz, k: Kont, +c: U32, +i: U32) -> IM: match fz: case FOff{}: IAl{alloc(env, fh, nx, Nul{}), i, k, c} case _: match k: case Arg{x, q}: IGo{subst(env, i, Sub{x}), fh, nx, FLam{}, q, (c + 1 : U32)} case k2: IAl{alloc(env, fh, nx, Nul{}), i, k2, c}

a drop: fused with a pending argument (freed), else copied

def inst.drp(env: Array<Cell>, +fh: U32, +nx: U32, fz: Fz, k: Kont, +c: U32) -> IM: match fz: case FOff{}: IUn{SDrp{}, env, fh, nx, k, c} case _: match k: case Arg{x, q}: IFr{free(env, x, fh), nx, q, (c + 1 : U32)} case k2: IUn{SDrp{}, env, fh, nx, k2, c}

a unit eliminator: fused with a pending () (L_RF0), else copied

def inst.use(env: Array<Cell>, +fh: U32, +nx: U32, fz: Fz, k: Kont, +c: U32) -> IM: match fz: case FOff{}: IUn{SUse{}, env, fh, nx, k, c} case _: match k: case Arg{x, q}: match x: case One{}: IGo{env, fh, nx, FLam{}, q, (c + 1 : U32)} case x2: IUn{SUse{}, env, fh, nx, Arg{x2, q}, c} case k2: IUn{SUse{}, env, fh, nx, k2, c}

a pair eliminator: fused with a pending pair (its parts become the

pending arguments), else copied

def inst.get(env: Array<Cell>, +fh: U32, +nx: U32, fz: Fz, k: Kont, +c: U32) -> IM: match fz: case FOff{}: IUn{SGet{}, env, fh, nx, k, c} case _: match k: case Arg{x, q}: match x: case Tup{a, b}: IGo{env, fh, nx, FLam{}, Arg{a, Arg{b, q}}, (c + 1 : U32)} case x2: IUn{SGet{}, env, fh, nx, Arg{x2, q}, c} case k2: IUn{SGet{}, env, fh, nx, k2, c}

a constructor eliminator meets a pending bit: the branch it selects

(t for #0, the sibling s for #1) runs on the rest, fused

def inst.mat.bit(env: Array<Cell>, +fh: U32, +nx: U32, k: Kont, +c: U32, y: Term, z: Bool) -> IM: match z: case True{}: IGo{env, fh, nx, FLam{}, Arg{y, k}, (c + 1 : U32)} case False{}: IGr{env, fh, nx, Arg{y, k}, (c + 1 : U32)}

a constructor eliminator meets a pending argument that is no bit:

at the root it defers to the argument's whnf (K_DISP), else the

eliminator is copied

def inst.mat.no(+fz: Fz, env: Array<Cell>, +fh: U32, +nx: U32, x: Term, q: Kont, +c: U32) -> IM: match fz: case FRoot{r}: IEnd{Ins{env, x, Rfk{r, q}, c, fh, nx}} case _: ITw{SMat{}, env, fh, nx, Arg{x, q}, c}

def inst.mat(env: Array<Cell>, +fh: U32, +nx: U32, +fz: Fz, k: Kont, +c: U32) -> IM: match fz: case FOff{}: ITw{SMat{}, env, fh, nx, k, c} case _: match k: case Arg{x, q}: match x: case Pak{p, t}: inst.mat.bit(env, fh, nx, q, c, pak_pop(p, t), bit0(p)) case Won{p}: inst.mat.bit(env, fh, nx, q, c, won_pop(p), bit0(p)) case x2: inst.mat.no(fz, env, fh, nx, x2, q, c) case k2: ITw{SMat{}, env, fh, nx, k2, c}

an application under fusion: the argument's copy becomes a pending

argument of the head (an application spine ends in the @ref it

calls, expanded by the loop)

def inst.app(env: Array<Cell>, +fh: U32, +nx: U32, fz: Fz, k: Kont, +c: U32) -> IM: match fz: case FOff{}: ITw{SApp{}, env, fh, nx, k, c} case _: IAg{env, fh, nx, k, c}

a dup's cell taken: both binders point at it

def inst.dq(+a: U32, +b: U32, al: Al, k: Kont, +c: U32) -> IM: Al{env, n, fh, nx} = al IGo{subst(subst(env, a, Num{n}), b, Num{n}), fh, nx, FOff{}, k, c}

def inst.xq(+a: U32, al: Al, k: Kont, +c: U32) -> IM: Al{env, n, fh, nx} = al IGo{subst(env, a, Num{n}), fh, nx, FOff{}, k, c}

@unsafe def inst(m: IM, t: Tpl, s: Tpl) -> Ins: match m: case IGo{env, fh, nx, fz, k, c}: match t: case TVar{i}: inst.var(k, c, fh, nx, Array.swap(Cell, env, i, Nul{})) case TCo0{l, i}: inst.cov(k, c, fh, nx, l, True{}, Array.swap(Cell, env, i, Nul{})) case TCo1{l, i}: inst.cov(k, c, fh, nx, l, False{}, Array.swap(Cell, env, i, Nul{})) case TSup{l, a, b}: inst(ITw{SSup{l}, env, fh, nx, k, c}, a, b) case TDup{a, b, v, q}: inst(IDv{a, b, env, fh, nx, k, c}, v, q) case TDux{a, v, q}: inst(IXv{a, env, fh, nx, k, c}, v, q) case TLam{i, f}: inst(inst.lam(env, fh, nx, fz, k, c, i), f, TEra{}) case TDrp{f}: inst(inst.drp(env, fh, nx, fz, k, c), f, TEra{}) case TApp{f, x}: inst(inst.app(env, fh, nx, fz, k, c), f, x) case TUse{f}: inst(inst.use(env, fh, nx, fz, k, c), f, TEra{}) case TPak{p, f}: inst(IUn{SPak{p}, env, fh, nx, k, c}, f, TEra{}) case TWon{p}: Ins{env, Won{p}, k, c, fh, nx} case TTup{a, b}: inst(ITw{STup{}, env, fh, nx, k, c}, a, b) case TGet{f}: inst(inst.get(env, fh, nx, fz, k, c), f, TEra{}) case TMat{f, g}: inst(inst.mat(env, fh, nx, fz,

k, c), f, g) case TOne{}: Ins{env, One{}, k, c, fh, nx} case TRef{n}: Ins{env, Ref{n}, k, c, fh, nx} case _: Ins{env, Era{}, k, c, fh, nx} case IUn{sh, env, fh, nx, k, c}: inst.one(sh, inst(IGo{env, fh, nx, FOff{}, k, c}, t, TEra{})) case ITw{sh, env, fh, nx, k, c}: inst(ILt{sh, inst(IGo{env, fh, nx, FOff{}, k, c}, t, TEra{})}, s, TEra{}) case ILt{sh, r}: Ins{env, a, k, c, fh, nx} = r inst.two(sh, a, inst(IGo{env, fh, nx, FOff{}, k, c}, t, TEra{})) case IAl{al, i, k, c}: Al{env, n, fh, nx} = al inst.one(SLam{n}, inst(IGo{subst(env, i, Sub{Var{n}}), fh, nx, FOff{}, k, c}, t, TEra{})) case IDv{a, b, env, fh, nx, k, c}: inst(IDq{a, b, inst(IGo{env, fh, nx, FOff{}, k, c}, t, TEra{})}, s, TEra{}) case IDq{a, b, r}: Ins{env, v, k, c, fh, nx} = r inst(inst.dq(a, b, alloc(env, fh, nx, Sha{v}), k, c), t, TEra{}) case IXv{a, env, fh, nx, k, c}: inst(IXq{a, inst(IGo{env, fh, nx, FOff{}, k, c}, t, TEra{})}, s, TEra{}) case IXq{a, r}: Ins{env, v, k, c, fh, nx} = r inst(inst.xq(a, alloc(env, fh, nx, Haf{v}), k, c), t, TEra{}) case IFr{fr, nx, k, c}: Fr{env, fh} = fr inst(IGo{env, fh, nx, FLam{}, k, c}, t, TEra{}) case IGr{env, fh, nx, k, c}: inst(IGo{env, fh, nx, FLam{}, k, c}, s, TEra{}) case IAg{env, fh, nx, k, c}: inst(IAr{inst(IGo{env, fh, nx, FOff{}, k, c}, s, TEra{})}, t, TEra{}) case IAr{r}: Ins{env, x, k, c, fh, nx} = r inst(IGo{env, fh, nx, FLam{}, Arg{x, k}, c}, t, TEra{}) case IEnd{ins}: ins

References

==========

A reference's template is moved out of the book, copied in place

(the copy borrows it) and written back (ref_fin).

def ref_fin(book: Array<Tpl>, +r: U32, u: Tpl, ins: Ins) -> S: Ins{env, t, k, c, fh, nx} = ins S{env, book_put(book, r, u), Go{t}, k, c, fh, nx}

def ref_tpl(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +r: U32, +u: Tpl) -> S: ins = inst(IGo{env, fh, nx, FRoot{r}, k, (c + 1 : U32)}, u, TEra{}) ref_fin(book, r, u, ins)

def go_ref(env: Array<Cell>, k: Kont, +c: U32, +fh: U32, +nx: U32, +r: U32, p: Array<Tpl> & Tpl) -> S: (book, u) = p ref_tpl(env, book, k, c, fh, nx, r, u)

K_DISP: a deferred eliminator @ref meets its scrutinee's whnf: a bit

selects the branch to copy (fused with the rest of the scrutinee),

any other whnf meets a copy of the whole template

def dsp_bit(env: Array<Cell>, k: Kont, +c: U32, +fh: U32, +nx: U32, f: Tpl, g: Tpl, y: Term, z: Bool) -> Ins: match z: case True{}: inst(IGo{env, fh, nx, FLam{}, Arg{y, k}, (c + 1 : U32)}, f, TEra{}) case False{}: inst(IGo{env, fh, nx, FLam{}, Arg{y, k}, (c + 1 : U32)}, g, TEra{})

def dsp_go(env: Array<Cell>, k: Kont, +c: U32, +fh: U32, +nx: U32, w: Term, u: Tpl) -> Ins: match w: case Pak{b, t}: match u: case TMat{f, g}: dsp_bit(env, k, c, fh, nx, f, g, pak_pop(b, t), bit0(b)) case _: Ins{env, Era{}, k, c, fh, nx} case Won{b}: match u: case TMat{f, g}: dsp_bit(env, k, c, fh, nx, f, g, won_pop(b), bit0(b)) case _: Ins{env, Era{}, k, c, fh, nx} case w2: inst(IGo{env, fh, nx, FOff{}, Arg{w2, k}, c}, u, TEra{})

def dsp_tpl(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +r: U32, w: Term, +u: Tpl) -> S: ins = dsp_go(env, k, c, fh, nx, w, u) ref_fin(book, r, u, ins)

def dsp(env: Array<Cell>, k: Kont, +c: U32, +fh: U32, +nx: U32, +r: U32, w: Term, p: Array<Tpl> & Tpl) -> S: (book, u) = p dsp_tpl(env, book, k, c, fh, nx, r, w, u)

The evaluator

==========

Every helper answers the next S; run matches the stack and the mode.

the mode to resume after a free (one interaction)

def freed(book: Array<Tpl>, k: Kont, +c: U32, +nx: U32, m: Mode, fr: Fr) -> S: Fr{env, fh} = fr S{env, book, m, k, (c + 1 : U32), fh, nx}

a Var read its cell: the substitution continues (the cell is free),

else the variable is stuck

def go_var(book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +i: U32, p: Array<Cell> & Cell) -> S: (env, s) = p match s: case Sub{t}: S{env, book, Go{t}, k, c, i, nx} case u: S{subst(env, i, u), book, Sk{Var{i}}, k, c, fh, nx}

a dup var read its cell (left empty): the twin's copy, or an

unresolved value evaluated under a Dpk frame, or stuck

def go_cov(book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +l: U32, +i: U32, +sd: U32, p: Array<Cell> & Cell) -> S: (env, s) = p match s: case Sub{t}: S{subst(env, i, Num{fh}), book, Go{t}, k, c, i, nx} case Sha{v}: S{env, book, Go{v}, Dpk{l, i, sd, 1, k}, c, fh, nx} case Haf{v}: S{env, book, Go{v}, Dpk{l, i, sd, 0, k}, c, fh, nx} case u: S{subst(env, i, u), book, Sk{cov(sd, l, i)}, k, c, fh, nx}

Application

---

def app_sup(book: Array<Tpl>, k: Kont, +c: U32, +l: U32, a: Term, b: Term, al: Al) -> S: Al{env, n, fh, nx} = al S{env, book, Kn{Sup{l, Cal{a, Co0{l, n}}, Cal{b, Co1{l, n}}}}}, k, (c + 1 : U32), fh, nx}

def app(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, w: Term, x: Term) -> S: match w: case Lam{i, f}: S{subst(env, i, Sub{x}), book, Go{f}, k, (c + 1 : U32), fh, nx} case Drp{f}: freed(book, k, c, nx, Go{f}, free(env, x, fh)) case Sup{l, a, b}: app_sup(book, k, c, l, a, b, alloc(env, fh, nx, Sha{x})) case Era{}: freed(book, k, c, nx, Kn{Era{}}, free(env, x, fh)) case Use{f}: S{env, book, Go{x}, Usk{f, k}, c, fh, nx} case Mat{f, g}: S{env, book, Go{x}, Mtk{f, g, k}, c, fh, nx} case Get{f}: S{env, book, Go{x}, Gtk{f, k}, c, fh, nx} case u: S{env, book, Sk{Cal{u, x}}, k, c, fh, nx}

a one-child eliminator (shape sh) meets a sup: each side gets it,

with its child shared through a fresh cell

def eli_sup(book: Array<Tpl>, k: Kont, +c: U32, +m: U32, x: Term, y: Term, +sh: Sh1, al: Al) -> S: Al{env, n, fh, nx} = al S{env, book, Kn{Sup{m, Cal{node1(sh, Co0{m, n}), x}, Cal{node1(sh, Co1{m, n}), y}}}, k, (c + 1 : U32), fh, nx}

def use(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, w: Term, f: Term) -> S: match w: case One{}: S{env, book, Go{f}, k, (c + 1 : U32), fh, nx} case Era{}: freed(book, k, c, nx, Kn{Era{}}, free(env, f, fh)) case Sup{m, x, y}: eli_sup(book, k, c, m, x, y, SUse{}, alloc(env, fh, nx, Sha{f})) case u: S{env, book, Sk{Cal{Use{f}, u}}, k, c, fh, nx}

the branch the bit selects runs on the rest; the other is freed

def mat_bit(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, f: Term, g: Term, y: Term, z: Bool) -> S: match z: case True{}: freed(book, Arg{y, k}, c, nx, Go{f}, free(env, g, fh)) case False{}: freed(book, Arg{y, k}, c, nx, Go{g}, free(env, f, fh))

def mat_sup(book: Array<Tpl>, k: Kont, +c: U32, +m: U32, x: Term, y: Term, al: Al2) -> S: Al2{env, n, p, fh, nx} = al S{env, book, Kn{Sup{m, Cal{Mat{Co0{m, n}, Co0{m, p}}, x}, Cal{Mat{Co1{m, n}, Co1{m, p}}, y}}}, k, (c + 1 : U32), fh, nx}

def mat(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, w: Term, f: Term, g: Term) -> S: match w: case Pak{b, t}: mat_bit(env, book, k, c, fh, nx, f, g, pak_pop(b, t), bit0(b)) case Won{b}: mat_bit(env, book, k, c, fh, nx, f, g, won_pop(b), bit0(b)) case Era{}: freed(book, k, c, nx, Kn{Era{}}, free(env, Mat{f, g}, fh)) case Sup{m, x, y}: mat_sup(book, k, c, m, x, y, alloc2(env, fh, nx, Sha{f}, Sha{g})) case u: S{env, book, Sk{Cal{Mat{f, g}, u}}, k, c, fh, nx}

def get(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, w: Term, f: Term) -> S: match w: case Tup{a, b}: S{env, book, Go{f}, Arg{a, Arg{b, k}}, (c + 1 : U32), fh, nx} case Era{}: freed(book, k, c, nx, Kn{Era{}}, free(env, f, fh)) case Sup{m, x, y}: eli_sup(book, k, c, m, x, y, SGet{}, alloc(env, fh, nx, Sha{f})) case u: S{env, book, Sk{Cal{Get{f}, u}}, k, c, fh, nx}

Duplication

---

a fires, the twin's copy b is freed with cell i

def settle_fin(book: Array<Tpl>, k: Kont, +c: U32, +nx: U32, +i: U32, a: Term, fr: Fr) -> S: Fr{env, fh} = fr S{subst(env, i, Num{fh}), book, Go{a}, k, (c + 1 : U32), i, nx}

def settle_ded(book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +i: U32, a: Term, p: Array<Cell> & Cell) -> S: (env, s) = p match s: case Sub{b}: settle_fin(book, k, c, nx, i, a, free(env, b, fh)) case _: S{env, book, Go{a}, k, (c + 1 : U32), fh, nx}

b parked in cell i, unless the twin died meanwhile (Ded)

def settle_park(book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +i: U32, a: Term, p: Array<Cell> & Cell) -> S: (env, s) = p match s: case Ded{}: settle_ded(book, k, c, fh, nx, i, a, Array.swap(Cell, env, i, Nul{})) case _: S{env, book, Go{a}, k, (c + 1 : U32), fh, nx}

def settle_go(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +i: U32, a: Term, b: Term, left: Bool, dead: Bool) -> S: match left dead: case True{} True{}: settle_fin(book, k, c, nx, i, a, free(env, b, fh)) case True{} False{}: settle_park(book, k, c, fh, nx, i, a, Array.swap(Cell, env, i, Sub{b})) case False{} True{}: settle_fin(book, k, c, nx, i, b, free(env, a, fh)) case False{} False{}: settle_park(book, k, c, fh, nx, i, b, Array.swap(Cell, env, i, Sub{a}))

side sd fires with its copy; the twin's parks in cell i (fl: the

twin is alive) or is freed with it

def settle(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +i: U32, +sd: U32, +fl: U32, a: Term, b: Term) -> S: settle_go(env, book, k, c, fh, nx, i, a, b, U32.is_eq(sd, 0), U32.is_eq(fl, 0))

def dup_sup(book: Array<Tpl>, k: Kont, +c: U32, +i: U32, +sd: U32, +fl: U32, +l: U32, +m: U32, al: Al2) -> S: Al2{env, n, p, fh, nx} = al settle(env, book, k, c, fh, nx, i, sd, fl, Sup{m, Co0{l, n}, Co0{l, p}}, Sup{m, Co1{l, n}, Co1{l, p}})

def dup_sup_if(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +i: U32, +sd: U32, +fl: U32, +l: U32, +m: U32, x: Term, y: Term, same: Bool) -> S: match same: case True{}: settle(env, book, k, c, fh, nx, i, sd, fl, x, y) case False{}: dup_sup(book, k, c, i, sd, fl, l, m, alloc2(env, fh, nx, Sha{x}, Sha{y}))

def dup_lam(book: Array<Tpl>, k: Kont, +c: U32, +i: U32, +sd: U32, +fl: U32, +l: U32, +j: U32, al: Al3) -> S: Al3{env, n, p, r, fh, nx} = al settle(subst(env, j, Sub{Sup{l, Var{p}, Var{r}}}), book, k, c, fh, nx, i, sd, fl, Lam{p, Co0{l, n}}, Lam{r, Co1{l, n}})

a one-child node (shape sh) copied: the child parks in a fresh cell

def dup_one(book: Array<Tpl>, k: Kont, +c: U32, +i: U32, +sd: U32, +fl: U32, +l: U32, +sh: Sh1, al: Al) -> S: Al{env, n, fh, nx} = al settle(env, book, k, c, fh, nx, i, sd, fl, node1(sh, Co0{l, n}), node1(sh, Co1{l, n}))

a two-child node (shape sh) copied: the children park in two cells

def dup_two(book: Array<Tpl>, k: Kont, +c: U32, +i: U32, +sd: U32, +fl: U32, +l: U32, +sh: Sh2, al: Al2) -> S: Al2{env, n, p, fh, nx} = al settle(env, book, k, c, fh, nx, i, sd, fl, node2(sh, Co0{l, n}, Co0{l, p}), node2(sh, Co1{l, n}, Co1{l, p}))

def dup_pak(book: Array<Tpl>, k: Kont, +c: U32, +i: U32, +sd: U32, +fl: U32, +l: U32, +p: U32, al: Al) -> S: Al{env, n, fh, nx} = al settle(env, book, k, ((c + (p >> 24n)) - 1 : U32), fh, nx, i, sd, fl, Pak{p, Co0{l, n}}, Pak{p, Co1{l, n}})

def dup(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +l: U32, +i: U32, +sd: U32, +fl: U32, w: Term) -> S: match w: case Era{}: settle(env, book, k, c, fh, nx, i, sd, fl, Era{}, Era{}) case One{}: settle(env, book, k, c, fh, nx, i, sd, fl, One{}, One{}) case Sup{m, x, y}: dup_sup_if(env, book, k, c, fh, nx, i, sd, fl, l, m, x, y, U32.is_eq(l, m)) case Lam{j, f}: dup_lam(book, k, c, i, sd, fl, l, j, alloc3(env, fh, nx, Sha{f}, Nul{}, Nul{})) case Drp{f}: dup_one(book, k, c, i, sd, fl, l, SDrp{}, alloc(env, fh, nx, Sha{f})) case Use{f}: dup_one(book, k, c, i, sd, fl, l, SUse{}, alloc(env, fh, nx, Sha{f})) case Get{f}: dup_one(book, k, c, i, sd, fl, l, SGet{}, alloc(env, fh, nx, Sha{f})) case Pak{p, t}: match t: case One{}: settle(env, book, k, (c + (p >> 24n) : U32), fh, nx, i, sd, fl, Pak{p, One{}}, Pak{p, One{}}) case u: dup_pak(book, k, c, i, sd, fl, l, p, alloc(env, fh, nx, Sha{u})) case Won{p}: settle(env, book, k, (c + (p >> 24n) : U32), fh, nx, i, sd, fl, Won{p}, Won{p}) case Tup{a, b}: dup_two(book, k, c, i, sd, fl, l, STup{}, alloc2(env, fh, nx, Sha{a}, Sha{b})) case Mat{f, g}: dup_two(book, k, c, i, sd, fl, l, SMat{}, alloc2(env, fh, nx, Sha{f}, Sha{g})) case u: S{env, book, Sk{u}, Dpk{l, i, sd, fl, k}, c, fh, nx}

Normalization

---

def nf_lam(book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, +i: U32, f: Term, p: Array<Cell> & Cell) -> S: (env, q) = p +n = (num(q) + 1 : U32) S{subst(subst(env, 0, Num{n}), i, Sub{Nam{n}}), book, Go{f}, NfK{NAbs{n, k}}, c, fh, nx}

the whnf under an NfK frame: children get normalized left to right

def nf(env: Array<Cell>, book: Array<Tpl>, k: Kont, +c: U32, +fh: U32, +nx: U32, w: Term) -> S: match w: case Sup{l, a, b}: S{env, book, Go{a}, NfK{NSupA{l, b, k}}, c, fh, nx} case Tup{a, b}: S{env, book, Go{a}, NfK{NTupA{b, k}}, c, fh, nx} case Mat{f, g}: S{env, book, Go{f}, NfK{NMatA{g, k}}, c, fh, nx} case Pak{p, t}: S{env, book, Go{t}, NfK{NPak{p, k}}, c, fh, nx} case Use{f}: S{env, book, Go{f}, NfK{NUse{k}}, c, fh, nx} case Get{f}: S{env, book, Go{f}, NfK{NGet{k}}, c, fh, nx} case Drp{f}: S{env, book, Go{f}, NfK{NDrp{k}}, c, fh, nx} case Lam{i, f}: nf_lam(book, k, c, fh, nx, i, f, Array.swap(Cell, env, 0, Nul{})) case u: S{env, book, Kn{u}, k, c, fh, nx}

The loop

@unsafe def run(s: S) -> Out: S{env, book, m, k, c, fh, nx} = s match m: case Go{t}: match t: case Cal{f, x}: run(S{env, book, Go{f}, Arg{x, k}, c, fh, nx}) case Var{+i}: run(go_var(book, k, c, fh, nx, i, Array.swap(Cell, env, i, Num{fh}))) case Co0{l, +i}: run(go_cov(book, k, c, fh, nx, l, i, 0, Array.swap(Cell, env, i, Nul{}))) case Co1{l, +i}: run(go_cov(book, k, c, fh, nx, l, i, 1, Array.swap(Cell, env, i, Nul{}))) case Ref{r}: Exp{env, book, k, c, fh, nx, r} case u: run(S{env, book, Kn{u}, k, c, fh, nx}) case Sk{w}: match k: case Arg{x, q}: run(S{env, book, Sk{Cal{w, x}}, q, c, fh, nx}) case Usk{f, q}: run(S{env, book, Sk{Cal{Use{f}, w}}, q, c, fh, nx}) case Mtk{f, g, q}: run(S{env, book, Sk{Cal{Mat{f, g}, w}}, q, c, fh, nx}) case Gtk{f, q}: run(S{env, book, Sk{Cal{Get{f}, w}}, q, c, fh, nx}) case Dpk{l, i, sd, fl, q}: run(S{subst(env, i, park(fl, w)), book, Sk{cov(sd, l, i)}, q, c, fh, nx}) case Rfk{r, q}: run(S{env, book, Sk{Cal{Ref{r}, w}}, q, c, fh, nx}) case q: run(S{env, book, Kn{w}, q, c, fh, nx}) case Kn{w}: match k: case Arg{x, q}: run(app(env, book, q, c, fh, nx, w, x)) case Usk{f, q}: run(use(env, book, q, c, fh, nx, w, f)) case Mtk{f, g, q}: run(mat(env, book, q, c, fh, nx, w, f, g)) case Gtk{f, q}: run(get(env, book, q, c, fh, nx, w, f)) case Dpk{l, i, sd, fl, q}: run(dup(env, book, q, c, fh, nx, l, i, sd, fl, w)) case Rfk{r, q}: Dsp{env, book, q, c, fh, nx, r, w} case NfK{q}: run(nf(env, book, q, c, fh, nx, w)) case NSupA{l, b, q}: run(S{env, book, Go{b}, NfK{NSupB{l, w, q}}, c, fh, nx}) case NSupB{l, a, q}: run(S{env, book, Kn{Sup{l, a, w}}, q, c, fh, nx}) case NTupA{b, q}: run(S{env, book, Go{b}, NfK{NTupB{w, q}}, c, fh, nx}) case NTupB{a, q}: run(S{env, book, Kn{Tup{a, w}}, q, c, fh, nx}) case NMatA{g, q}: run(S{env, book, Go{g}, NfK{NMatB{w, q}}, c, fh, nx}) case NMatB{f, q}: run(S{env, book, Kn{Mat{f, w}}, q, c, fh, nx}) case NPak{p, q}: run(S{env, book, Kn{Pak{p, w}}, q, c, fh, nx}) case NUse{q}: run(S{env, book, Kn{Use{w}}, q, c, fh, nx}) case NGet{q}: run(S{env, book, Kn{Get{w}}, q, c, fh, nx}) case NDrp{q}: run(S{env, book, Kn{Drp{w}}, q, c, fh, nx}) case NAbs{n, q}: run(S{env, book, Kn{Abs{n, w}}, q, c, fh, nx}) case _: Fin{env, book, w, c}

The driver

---

run is a flat loop; the copier is not (it recurses), so a reference

comes back to the driver, which expands it and resumes the loop.

@unsafe def drive(o: Out) -> Out: match o: case Exp{env, book, k, c, fh, nx, +r}: drive(run(go_ref(env, k, c, fh, nx, r, Array.swap(Tpl, book, r, TEra{})))) case Dsp{env, book, k, c, fh, nx, +r, w}: drive(run(dsp(env, k, c, fh, nx, r, w, Array.swap(Tpl, book, r, TEra{})))) case o2: o2

Show

---

one display-name digit

def nam_digit(d: U32) -> String: SCon{Chr{str_at(alphabet(), U32.to_nat(d))}, ""}

a name: base-64 digits, most significant first (print_name); one: n

is a single digit; d bounds the digits

```
def nam_show(d: Nat, one: Bool, +n: U32) -> String: match d one: case 0n _: "" case 1n+p True{}: nam_digit(n) case 1n+p False{}: +h = (n >> 6n : U32) nam_show(p, U32.is_lt(h, 64), h) ++ nam_digit(U32.and(n, 63))
```

a name from its code

```
def nam(+n: U32) -> String: nam_show(6n, U32.is_lt(n, 64), n)
```

the constructors a run word opens, outermost first; n: bits left

```
def bits_open(n: Nat, +w: U32) -> String: match n: case 0n: "" case 1n+p: "#" ++ pick(String, bit0(w), "0", "1") ++ "{" ++ bits_open(p, (w >> 1n : U32))
```

```
def bits_close(n: Nat) -> String: match n: case 0n: "" case 1n+p: "}" ++ bits_close(p)
```

```
def run_len(+p: U32) -> Nat: U32.to_nat((p >> 24n : U32))
```

```
def term_show(t: Term) -> String: match t: case Nam{n}: nam(n) case Abs{n, f}: lam() ++ nam(n) ++ "." ++ term_show(f) case Lam{i, f}: lam() ++ nam(i) ++ "." ++ term_show(f) case Drp{f}: lam() ++ "- " ++ term_show(f) case Use{f}: lam() ++ "() " ++ term_show(f) case Get{f}: lam() ++ "! " ++ term_show(f) case Mat{f, g}: lam() ++ "{#0:" ++ term_show(f) ++ "; #1:" ++ term_show(g) ++ "}" case Era{}: "&{}" case One{}: "()" case Sup{l, a, b}: "&" ++ nam(l) ++ "{" ++ term_show(a) ++ "," ++ term_show(b) ++ "}" case Tup{a, b}: "#(" ++ term_show(a) ++ "," ++ term_show(b) ++ ")" case Pak{p, x}: bits_open(run_len(p), p) ++ term_show(x) ++ bits_close(run_len(p)) case Won{p}: bits_open(run_len(p), p) ++ "()" ++ bits_close(run_len(p)) case Cal{f, x}: "(" ++ term_show(f) ++ " " ++ term_show(x) ++ ")" case u: "<var>"
```

Main

```
====
```

the env: 2^16 cells, the free list keeps the live set small (cell 0

holds the display-name counter)

```
def env_new() -> Array<Cell>: [Nul{} : Cell*65536n]
```

```
def out(o: Out) -> IO(Unit): match o: case Fin{env, book, w, c}: IO.print(term_show(w) ++ "\n- Itrs: " ++ U32.show(c) ++ " interactions") case _: IO.print("stuck")
```

the depth of an array with room for n entries; f: the bits left to try

```
def book_depth(f: Nat, +n: U32, +d: Nat) -> Nat: match f: case 0n: d case 1n+p: pick(Nat, U32.is_le(n, (1 << d : U32)), d, book_depth(p, n, 1n+d))
```

the definitions written into the book

```
def book_load(es: List<&2, Ent>, a: Array<Tpl>) -> Array<Tpl>: match es: case Ent{ix, t} <> r: book_load(r, book_put(a, ix, t)) case Nil{}: a
```

```
def ents_has.go(eq: Bool, e: Ent, p: List<&2, Ent> & Bool) -> List<&2, Ent> & Bool: (r, b) = p (e <> r, eq || b)
```

is definition ix among the parsed ones

```
def ents_has(es: List<&2, Ent>, +ix: U32) -> List<&2, Ent> & Bool: match es: case Ent{+i, t} <> r: ents_has.go(U32.is_eq(i, ix), Ent{i, t}, ents_has(r, ix)) case Nil{}: (Nil{}, False{})
```

@main normalized and printed; a book without it is an error

```
def book_run.fin(+ix: U32, +n: U32, +vc: U32, p: List<&2, Ent> & Bool) -> IO(Unit): (es, ok) = p match ok: case True{}: out(drive(run(S{env_new(), book_load(es, [TEra{} : Tpl^book_depth(32n, n, 0n)]), Go{Ref{ix}}, NfK{End{}}, 0, 0, vc}))) case False{}: IO.print("undefined function @main")
```

```
def book_run.go(es: List<&2, Ent>, +vc: U32, p: Names & U32) -> IO(Unit): (Names{m, n}, +ix) = p book_run.fin(ix, n, vc, ents_has(es, ix))
```

the parsed definitions run: binder ids end at vc, fresh ids follow

```
def book_run(es: List<&2, Ent>, s: St) -> IO(Unit): St{h, src, sc, vc, ns} = s book_run.go(es, vc, names_find(ns, "main"))

def main() -> IO(Unit): prog()(book_run, st_new(program(), Nil{}, 1, Names{Map.new(&2, U32), 0}))
```

G.65 · `bend-runtime/hvm5/reference/hvm5-mini/laws.bend`

HVM5 in Bend -- the laws. Claims about the pure logic under the

evaluator: the lexer (a word and the rest re-form the source, a

blank skip only moves the cursor), the parser's cursor (drops

compose, a read past a drop is a read at the sum), the printer (a

pack of n constructors prints n more braces than its body), and

the closed facts about run words, name codes and the book's depth.

PROOF.bend proves them.

```
import Base import ./main.bend as Hvm
```

LAW: the lexer loses nothing: for any source and any head class,

the word at the head and the rest re-form the source.

```
law word_joins: for +s: String for z: Bool {Hvm.word_cat(Hvm.str_word(s, z)) == s : String}
```

LAW: skipping blanks and comments only moves the cursor forward:

for any source and any head class, the skipped source is the source

past some n characters (n is the witness).

```
law skip_is_drop: for +s: String for z: Hvm.Ws exs n: Nat {Hvm.str_skip(s, z) == Hvm.str_drop(s, n) : String}
```

LAW: the printer's packs are balanced: the display of a pack of run

p around x opens run_len(p) braces more than the display of x, for

any run word and any term.

```
law pak_braces: for +p: U32 for +x: Hvm.Term {Nat.add(Hvm.run_len(p), Hvm.opens(Hvm.term_show(x))) == Hvm.opens(Hvm.term_show(Hvm.Pak{p, x})) : Nat}
```

LAW: the cursor composes: dropping m characters, then n, drops

m + n.

```
law drop_drop: for s: String for m: Nat for n: Nat {Hvm.str_drop(Hvm.str_drop(s, m), n) == Hvm.str_drop(s, Nat.add(m, n)) : String}
```

LAW: the code at k past a cursor moved by n is the code at n + k.

```
law at_drop: for s: String for n: Nat for k: Nat {Hvm.str_at(Hvm.str_drop(s, n), k) == Hvm.str_at(s, Nat.add(n, k)) : U32}
```

LAW: a bit pushed outermost onto a run word is the bit bit0 reads back, and popping it restores the word (here the one-bit run #1{..}, for either bit) (sanity).

law run_push_pop: for +b: Bool {(Hvm.bit0(Hvm.run_push(Hvm.run_word(1, 1), Hvm.pick(U32, b, 1, 0))), Hvm.run_pop(Hvm.run_push(Hvm.run_word(1, 1), Hvm.pick(U32, b, 1, 0)))) == (Bool.not(b), Hvm.run_word(1, 1)) : Bool & U32}

LAW: a name survives its code: the parser's six-bits-per-character code of a name prints back as the name (sanity).

law name_round_trip: {(Hvm.nam(Hvm.name_code("main", 0)), Hvm.nam(Hvm.name_code("F", 0))) == ("main", "F") : String & String}

LAW: the book's depth is the least d with 2^d slots for n definitions (sanity).

law book_depth_fits: {(Hvm.book_depth(32n, 1, 0n), Hvm.book_depth(32n, 5, 0n), Hvm.book_depth(32n, 8, 0n), Hvm.book_depth(32n, 9, 0n)) == (0n, 3n, 3n, 4n) : Nat & Nat & Nat & Nat}

G.66 · `bend-runtime/hvm5/reference/hvm5-mini/proof.bend`

HVM5 in Bend -- the proofs. This file imports the interpreter (as Hvm) and the claims (as Laws) and fills every law. word_joins is an induction over the source: a name character is put back by go_cat (the head over the re-joined rest) and the rest is the induction hypothesis. skip_is_drop builds its witness by induction: a skipped head adds one to the rest's witness (skip_step), a kept head or an empty source gives 0. pak_braces splits the display with opens_cat (a count over an append is the sum), counts the open braces of the run (bits_open_opens, an induction over the run's length with the bit's digit split by pick_opens) and the none of its closing braces (bits_close_opens). The cursor laws are inductions over the source and the position; the closed laws compute.

import Base import ./main.bend as Hvm import ./laws.bend as Laws

The lexer

---

the head put back over the re-joined rest

law go_cat: for c: Char for p: String & String {SCon{c, Hvm.word_cat(p)} == Hvm.word_cat(Hvm.str_word.go(c, p)) : String}

def go_cat(c, p): (w, u) = p {==}

def Laws.word_joins(s, z): match s z: case SCon{c, +t} True{}: %go_cat(c, Hvm.str_word(t, Hvm.chr_name(Hvm.str_at(t, 0n)))) : {_ == SCon{c, t} : String} %Laws.word_joins(t, Hvm.chr_name(Hvm.str_at(t, 0n))) : {SCon{c, Hvm.word_cat(Hvm.str_word(t, Hvm.chr_name(Hvm.str_at(t, 0n))))} == SCon{c, _} : String} {==} case SCon{c, t} False{}: {==} case SNil{} _: {==}

a skipped head: the cursor moves one more than the rest's witness

law skip_step: for c: Char for -t: String for -y: Hvm.Ws for w: &n: Nat -> {Hvm.str_skip(t, y) == Hvm.str_drop(t, n) : String} exs n: Nat {Hvm.str_skip(t, y) == Hvm.str_drop(SCon{c, t}, n) : String}

def skip_step(c, t, y, w): (n, e) = w (1n+n, e)

def Laws.skip_is_drop(s, z): match s z: case SCon{c, +t} Hvm.Sp{}: skip_step(c, t, Hvm.str_ws(t), Laws.skip_is_drop(t, Hvm.str_ws(t))) case SCon{c, +t} Hvm.Cm{}: skip_step(c, t, Hvm.str_cm(t), Laws.skip_is_drop(t, Hvm.str_cm(t))) case SCon{c, t} Hvm.No{}: (0n, {==}) case SNil{} _: (0n, {==})

The printer

---

law go_add: for brace: Bool for -x: Nat for -y: Nat {Nat.add(Hvm.opens.go(brace, x), y) == Hvm.opens.go(brace, Nat.add(x, y)) : Nat}

def go_add(brace, x, y): match brace: case True{}: {==} case False{}: {==}

the braces of an append are the sum of the braces

law opens_cat: for a: String for -b: String {Nat.add(Hvm.opens(a), Hvm.opens(b)) == Hvm.opens(a ++ b) : Nat}

def opens_cat(a, b): match a: case SNil{}: {==} case SCon{Chr{c}, t}: %opens_cat(t, b) : {Nat.add(Hvm.opens.go(Hvm.chr_is(c, '{'), Hvm.opens(t)), Hvm.opens(b)) == Hvm.opens.go(Hvm.chr_is(c, '{'), _) : Nat} go_add(Hvm.chr_is(c, '{'), Hvm.opens(t), Hvm.opens(b))

a bit's digit, then "{": one brace more than the rest, for either bit

law pick_opens: for b: Bool for -r: String {1n+Hvm.opens(r) == Hvm.opens(Hvm.pick(String, b, "0", "1") ++ ("{" ++ r)) : Nat}

def pick_opens(b, r): match b: case True{}: {==} case False{}: {==}

a run of n constructors opens n braces

law bits_open_opens: for n: Nat for +w: U32 {n == Hvm.opens(Hvm.bits_open(n, w)) : Nat}

def bits_open_opens(n, w): match n: case 0n: {==} case 1n+p: %pick_opens(Hvm.bit0(w), Hvm.bits_open(p, (w >> 1n : U32))) : {1n+p == _ : Nat} %bits_open_opens(p, (w >> 1n : U32)) : {1n+p == 1n+_ : Nat} {==}

the closing braces open none

law bits_close_opens: for n: Nat {0n == Hvm.opens(Hvm.bits_close(n)) : Nat}

def bits_close_opens(n): match n: case 0n: {==} case 1n+p: bits_close_opens(p)

law add_zero: for x: Nat {x == Nat.add(x, 0n) : Nat}

def add_zero(x): match x: case 0n: {==} case 1n+p: %add_zero(p) : {1n+p == 1n+_ : Nat} {==}

def Laws.pak_braces(p, x): %opens_cat(Hvm.bits_open(Hvm.run_len(p), p), Hvm.term_show(x) ++ Hvm.bits_close(Hvm.run_len(p))) : {Nat.add(Hvm.run_len(p), Hvm.opens(Hvm.term_show(x))) == _ : Nat} %opens_cat(Hvm.term_show(x), Hvm.bits_close(Hvm.run_len(p))) : {Nat.add(Hvm.run_len(p), Hvm.opens(Hvm.term_show(x))) == Nat.add(Hvm.opens(Hvm.bits_open(Hvm.run_len(p), p)), *) : Nat} %bits_open_opens(Hvm.run_len(p), p) : {Nat.add(Hvm.run_len(p), Hvm.opens(Hvm.term_show(x))) == Nat.add(*, Nat.add(Hvm.opens(Hvm.term_show(x)), Hvm.opens(Hvm.bits_close(Hvm.run_len(p))))) : Nat} %bits_close_opens(Hvm.run_len(p)) : {Nat.add(Hvm.run_len(p), Hvm.opens(Hvm.term_show(x))) == Nat.add(Hvm.run_len(p), Nat.add(Hvm.opens(Hvm.term_show(x)), _)) : Nat} %add_zero(Hvm.opens(Hvm.term_show(x))) : {Nat.add(Hvm.run_len(p), Hvm.opens(Hvm.term_show(x))) == Nat.add(Hvm.run_len(p), _) : Nat} {==}

The cursor

---

def Laws.drop_drop(s, m, n): match s m: case SNil{} On: {==} case SNil{} 1n+p: {==} case SCon{h, t} On: {==} case SCon{h, t} 1n+p: Laws.drop_drop(t, p, n)

def Laws.at_drop(s, n, k): match s n: case SNil{} On: {==} case SNil{} 1n+p: {==} case SCon{Chr{c}, t} On: {==} case SCon{Chr{c}, t} 1n+p: Laws.at_drop(t, p, k)

Sanity

---

def Laws.run_push_pop(b): match b: case True{}: {==} case False{}: {==}

def Laws.name_round_trip(): {==}

def Laws.book_depth_fits(): {==}

PART H · Primary sources: Victor Taelin, HigherOrderCO, bend2.dev, upstream bendlang/bend (verbatim)

The source-material pass retrieved 2026-09-05 (bend2.dev pages; repo READMEs for HVM1-HVM4, Kind, Bend, Bend2, Interaction-Calculus, hvm4-formal; the HVM4 docs primer/IC-theory; Victor Taelin's profile and posts indexed under victor/; the concept map). Plus, at the top, four files fetched fresh from github.com/bendlang/bend on 2026-11-06: the current README (the reference the user asked for) and the guide/EFFECTS/SHADERS documents.

Source root: `bend-docs/openspec/changes/bend-docs-site/research + fresh fetches` · **55 files · 480.7 KB**

▼ In this part (file list)

- **H.1** · `bendlang/bend · README.md`
- **H.2** · `bendlang/bend · guide/GUIDE.md`
- **H.3** · `bendlang/bend · guide/EFFECTS.md`
- **H.4** · `bendlang/bend · guide/SHADERS.md`
- **H.5** · `fresh fetch · retrieval record`
- **H.6** · `bend-docs/openspec/changes/bend-docs-site/research/README.md`
- **H.7** · `bend-docs/openspec/changes/bend-docs-site/research/concept-map.md`
- **H.8** · `bend-docs/openspec/changes/bend-docs-site/research/RETRIEVED.txt`
- **H.9** · `bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-release-date.md`
- **H.10** · `bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-vs-bend1.md`
- **H.11** · `bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-vs-lean.md`
- **H.12** · `bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-vs-mojo.md`
- **H.13** · `bend-docs/openspec/changes/bend-docs-site/research/sources/contents.md`
- **H.14** · `bend-docs/openspec/changes/bend-docs-site/research/sources/gpu.md`
- **H.15** · `bend-docs/openspec/changes/bend-docs-site/research/sources/hello.md`
- **H.16** · `bend-docs/openspec/changes/bend-docs-site/research/sources/home.md`
- **H.17** · `bend-docs/openspec/changes/bend-docs-site/research/sources/hvm4-explained.md`
- **H.18** · `bend-docs/openspec/changes/bend-docs-site/research/sources/interaction-nets-explained.md`
- **H.19** · `bend-docs/openspec/changes/bend-docs-site/research/sources/learn-index.md`
- **H.20** · `bend-docs/openspec/changes/bend-docs-site/research/sources/parallelism.md`
- **H.21** · `bend-docs/openspec/changes/bend-docs-site/research/sources/proofs.md`
- **H.22** · `bend-docs/openspec/changes/bend-docs-site/research/sources/what-is-bend2.md`
- **H.23** · `bend-docs/openspec/changes/bend-docs-site/research/repos/_meta.txt`
- **H.24** · `bend-docs/openspec/changes/bend-docs-site/research/repos/victortaelin-profile.md`
- **H.25** · `bend-docs/openspec/changes/bend-docs-site/research/repos/taelin-theorem-proving-vs-testing.md`
- **H.26** · `bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_Bend-README.md`
- **H.27** · `bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM1-README.md`
- **H.28** · `bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM2-README.md`
- **H.29** · `bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM3-README.md`
- **H.30** · `bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM4-README.md`
- **H.31** · `bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_Kind-README.md`
- **H.32** · `bend-docs/openspec/changes/bend-docs-site/research/repos/VictorTaelin_Bend2-README.md`
- **H.33** · `bend-docs/openspec/changes/bend-docs-site/research/repos/VictorTaelin_Interaction-Calculus-README.md`
- **H.34** · `bend-docs/openspec/changes/bend-docs-site/research/repos/VictorTaelin_hvm4-formal-README.md`
- **H.35** · `bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-collapser.md`
- **H.36** · `bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-core.md`
- **H.37** · `bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-memory.md`
- **H.38** · `bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-syntax.md`
- **H.39** · `bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/ic-theory.md`

- **H.40** · `bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/primer.md`
- **H.41** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hoc-history.md`
- **H.42** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hoc-optimal-computer.md`
- **H.43** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-10min.md`
- **H.44** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-collapse-monad.md`
- **H.45** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-deforestation.md`
- **H.46** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-how.md`
- **H.47** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-linker.md`
- **H.48** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-mystery.md`
- **H.49** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-sat.md`
- **H.50** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-search1.md`
- **H.51** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-search3.md`
- **H.52** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-sieve.md`
- **H.53** · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-unordered-sup.md`
- **H.54** · `bend-docs/openspec/changes/bend-docs-site/research/victor/kind-conv-checker.md`
- **H.55** · `bend-docs/openspec/changes/bend-docs-site/research/victor/term-checker.md`

H.1 · bendlang/bend · README.md

Bend (bendlang/bend) — README (fetched 2026-11-06, main)

<picture><source media="(prefers-color-scheme: dark)" srcset="media/hero_dark.gif">`[image not bundled: Bend: a fast language that blocks AI mistakes via proof]`</picture>

In the post-AGI economy, humans will eventually stop writing and reading code, but we still need an ambiguity-free language to communicate our intents to the AIs building the world around us. Bend is that language.

With **laws**, intents can be more precise than natural language. With **proofs**, we can mechanically verify the AI implemented our prompts correctly. And with a **fast compiler**, we can run that code at peak compute.

That's Bend - and nothing else.

BEND RUNS FAST

Target: be as fast as C on the CPU, as fast as CUDA on the GPU. **Status:**

`[image not bundled: Runtime benchmarks: Bend vs C, TypeScript, Lean, on 1 core, 16 cores and the GPU]`

Thanks to strong types, purity and linearity, Bend compiles to fast executables as fast as hand-written C (single-core), and even faster (on 10000s cores). The entire language runs on the GPU, with full memory unification.

BEND CHECKS FAST

Target: outperform every proof assistant by several OOMs. **Status:**

`[image not bundled: Checker benchmarks: Bend vs Isabelle, Agda, Lean, Rocq]`

Bend's compiler is so powerful it can verify mathematical proofs. Usually, this is slow. Bend is not. It checks, in under a second, files that other projects would take minutes, making proofs way more practical.

BEND IS PARALLEL

No threads, no locks, no kernels to write. Split the work in two, and Bend spreads the calls over every core it can find, then joins them back. Below, `pow2(20)` divides until one task sits on each of 4,096 GPU cores:

`[image not bundled: pow2 splitting over 4,096 GPU cores, then folding back]`

BEND BLOCKS MISTAKES - WITH PROOF

PROBLEM: How can you **trust** AI code, without reading it?

SOLUTION: By forcing your AI to write a **correctness proof**.

Bend introduces `LAWS.bend`, a file where you declare rules that your app must not break. Bend's compiler then **guarantees** that these laws always hold, by demanding **mathematical proof** whenever your code is edited. For example, consider a game with one law: *winning is impossible*. Here's how it plays out:

Law: winning is **impossible**
`[image not bundled: The player walks up and bumps the wall of the flag's room]`
So far, it works!

New feature: "Claude, make the board wrap around"

Without LAWS.bend:
`[image not bundled: The player wraps around the edge and takes the flag]`
*Laws broken. AI mistake: **merged**.*

With LAWS.bend:
`[image not bundled: A wall on the far edge stops the player]`
*Laws intact. AI mistake: **blocked**!*

Without `LAWS.bend`, a bug was merged. With it, the AI had to retry, until no bugs were left! In this case, it added a wall, but it could have moved the flag, made the room kill you, or whatever. The only thing it can't do is commit a bug, because it is **mathematically impossible** to break laws in `LAWS.bend`. The compiler *enforces* it.

Using `LAWS.bend` is simple.

1. Ask your AI to formalize your app's rules in `LAWS.bend`. Example:
   - LAW: *"the sum of all balances must be zero"*
   - LAW: *"players can never pass through solid walls"*
   - LAW: *"list_sort() must always return ascending numbers"*
   - LAW: *"array_set() may never be called out-of-bounds"*
   - LAW: *"winning is impossible"* (the demo above!)
   - And so on. Anything you can spell can become a law.
2. Ask your AI to run `bend PROOF.bend` after editing any code.
3. That's it. Rejoice as your app never again breaks or violates your rules.

You can also edit `LAWS.bend` yourself. Here's how it looks:

```
# LAWS.bend
law you_cant_win:                          # "winning is impossible"
  for moves: List<Game.Move>               # any sequence of moves
  board = Game.replay(Game.start(), moves)  # replayed from the start
  {Game.is_won(board) == False{} : Bool}    # never leads to victory
```

```
# PROOF.bend
def Laws.you_cant_win(moves):
  # ... written by the AI
```

In short, `LAWS.bend` is `AGENTS.md` backed by **proof**.

With `LAWS.bend`, *"make no mistakes"* becomes enforceable.

Skeptical? Edit the demo's code and break the "you can't win" law! `(https://bend-lang.com/#lab)`

Get Started

1. INSTALL:

```
curl -fsSL https://bend-lang.com/install.sh | sh
```

This `bend` ignores a project's `bunfig.toml` and `.env`; `bun bend2/main.ts` from a checkout reads them, so check untrusted code with `bend`.

2. TELL YOUR AGENT TO USE BEND:

Add this to your `AGENTS.md`:

```
When using Bend:
- run `bend guide` to learn it
- use `LAWS.bend` to keep important rules
- run `bend PROOF.bend` before committing
- parallelize the code whenever possible
```

Then, just say: "use Bend"!

3. ENJOY BUG-FREE, FAST VIBE-CODED APPS!

Hints:

- Ask it to write laws for anything that can't break.
- Ask it to parallelize anything you want to be fast.
- Bend is young. If anything goes wrong, ask it to open an issue. <3

Bend works best on the back-end, on Linux or macOS.

Examples

SYNTAX == PYTHON + DEPENDENT TYPES

```
import Base

# Performs effects on the CPU.
def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER"))
    IO.print("Hello, " ++ name)
```

PARALLELISM == DIVIDE-AND-CONQUER

```
import Base

# Computes 2^d in parallel: a tree of d levels, one leaf per unit.
def pow2(+d: Nat) -> U32:
  match d:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)

# Runs pow2 on the GPU, via `!`.
def main() -> IO(Unit):
  result = pow2!(20n)
  IO.print(U32.show(result))
```

THEOREMS == LAWS, PROOFS == DEFS

```
import Base

# CLAIM: for every nat x, x + 0 equals x.
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

# PROOF: induction on `x`, one rewrite (`%`) per step.
def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+xp:
      %add_zero(xp) : {1n+Nat.add(xp, 0n) == 1n+_ : Nat}
      {==}
```

References

- Guide: GUIDE.md, also printed by `bend guide`.
- Demos: demos/, apps, servers and proofs, each with its `LAWS.bend`.
- Base: base.bend, the base library, also printed by `bend base`.
- Paper: BendTT: An Affine Dependent Type Theory.
- Paper: BendRT: A Parallel Runtime for CPUs and GPUs.
- Formalization: bendtt.lean, the kernel of `--verdict` and its proofs, in Lean.
- Benches: bench/, every bench used to make the charts above.
- Formatter: bend-fmt-lsp, a formatting-only Bend 2 language server.
- Community language server: bend2-lsp (https://github.com/don2e4/bend2-lsp), with formatting, diagnostics, and hover.
- Community workshop: Bend 2 pocket workshop (https://np.github.io/bend-workshop/), the checker and JS compiler in one HTML page, with goals and proof tools, phone first.

Community

- Website: https://bend-lang.com (https://bend-lang.com)
- Discord: https://discord.bend-lang.com (https://discord.bend-lang.com)
- Twitter/X: https://x.com/bendlang (https://x.com/bendlang)
- Reddit: https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/)
- Issues: https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues)

Limitations

```
 - Bend 2 is a new language. Bend 1 programs and HVM do not carry over.
- Everything is annotated and nothing is inferred, so code is verbose.
- No type classes, no traits, and no macros beyond compile-time templates.
- Bend has no tactics or proof search; proving theorems takes extra effort.
- Values are affine: closures and arrays cannot be shared.
- Recursion must terminate; `@unsafe` or `def f?(..)` turns the check off.
- Computed matches (`match f(x)`) aren't supported. Must split it manually.
- There is no syntax for if-then-else: a branch is a match on True and False.
- Numbers are Nat, U32 and F32 only: no U64, I64 or F64 (Metal has no f64).
- F32 is axiomatic: nothing about floating point can be proven.
- Strings are linked lists of characters, so text processing is slow.
- Base is small: expect to write helpers other languages ship built in.
- Effects are few: print, env, time, sleep, spawn, channels, files, TCP, UDP.
  UDP buffers fit IPv4's 65,507-byte payload limit.
- No TLS, HTTP library, JSON or regex for now (you can add them as foreigns).
- Targets are C, Metal, CUDA and JavaScript; Lua, Luau and Python are planned.
- The JavaScript target runs on one core and has no graphics or audio.
- Parallelism requires balanced calls. Flexible parallelism will be added later.
- Atomic arrays shared across threads are experimental and need `@unsafe`.
- One GPU per program, one event loop, and no multi-machine execution yet.
- One C file per program: no separate compilation, no incremental builds.
- Compiling to native is slow (clang/CUDA/Metal). For fast development, use JS.
- The compiler is young and has blind spots (unusually slow programs). Report.
- We don't have as many benchmarks as we'd like yet, especially for the checker.
- The compiler (not kernel) is 99% AI-written and not yet fully audited.
- The checker has no proof and may have bugs; `--verdict` uses a proven kernel.
- Laws are proven about Bend's semantics. The backends can still differ, for example in F32 NaN. See WONTFIX.
- `--verdict` checks the translated terms, the parser and translation into the kernel are not proven.
- A binary needs clang 14+; ! needs 19+, Metal or CUDA 12.
- No Windows (WSL works); on Linux, Window and Audio need X11 and ALSA headers.
- A hub package is a hash, unless its author names and versions it after `bend login`.
- Error messages are terse; no debugger, profiler or REPL.
- The editor tool only formats; the community bend2-lsp adds errors and hover.
- No test framework and no documentation beyond the guide.
- And more that escape me. Be patient, report bugs and request features!

Most of these limitations are being addressed and will improve over time!
```

BEND IS YOUNG. EXPECT BUGS AND REPORT THEM (https://github.com/bendlang/bend/issues).

Credits

Bend is created by Victor Taelin (https://github.com/VictorTaelin) and built by the team:

- Lorenzo W Battistela (https://github.com/Lorenzobattistela)
- Paulo J Cavalcanti (https://github.com/pjcavalcanti)
- Nico Abril (https://github.com/nicolas-abril)
- Vanessa Ostroski (https://github.com/Ostrowskii)
- Vitor Chiarelli Neves (https://github.com/Sipher)
- Alex Van de Sande (https://x.com/avsa)

If you were part of this and your name is missing, please get in touch so we can add it here.

Thanks to Ayush Somani (https://ayushsomani.me/) for reserving the `bend-lang` name for us.

H.2 · bendlang/bend · guide/GUIDE.md

Bend guide/GUIDE.md (fetched 2026-11-06, main; also printed by `bend guide`)

Bend

Bend is a new programming language that combines Lean-like formal proofs with C-like speeds and CUDA-like parallelism. It gives humans an ambiguity-free language to communicate their intents to AIs, a compiler capable of mechanically checking that the AI implemented these intents to unquestionable mathematical correctness, and a compiler that runs that code fast on CPUs and GPUs.

HELLO, WORLD!

Bend's syntax is Python-shaped, but its semantics are closer to Haskell / Lean, while being resource-aware like Rust (if less annoyingly). A hello world is just a typed definition returning an IO block:

```
import Base

def main() -> IO(Unit):
  do IO<Unit>:
    IO.print("Hello, world!")
```

To run it, install Bend (`curl -fsSL https://bend-lang.com/install.sh | sh`) and type `bend hello.bend` .

Bend is a *pure language*, with effects denoted via a Haskell-inspired IO Monad (https://wiki.haskell.org/Introduction_to_IO). It comes with a list of built-in effects for files, networking, audio, graphics, input, and more, and the user can extend them with foreign C and JS imports; more on that later.

CORE FEATURES

TYPES AND FUNCTIONS

A typical Bend program is a set of datatypes and functions over them:

```
import Base

type Shape is Data:
  Circle{r: U32}
  Square{s: U32}

def area(x: Shape) -> U32:
  match x:
    case Circle{+r}:
      (3 * r * r : U32)
    case Square{+s}:
      (s * s : U32)

def main() -> U32:
  area(Square{5})
```

Bend, by default, is *affine*, meaning variables must be used, at most, once.

Here, `is Data` declares that `Shape` may be copied, while `is Type` keeps it non-copiable. The `+` annotation before a variable name allows using it more than once, if the variable is Data-kinded.

Bend does almost no inference, meaning it requires more annotations than similar languages. This is what allows Bend's checker to be significantly faster than other provers, and its error messages more precise, at the expense of programs and proofs being more verbose. When the checker can't decide the type of an expression, just annotate it, as in `{3 : U32}` .

CLOSURES

Functions are values and can be stored, passed, and returned from other functions.

```
import Base

def adder(k: U32) -> U32 -> U32:
  x => (x + k : U32)

def main() -> U32:
  add2 = adder(2)
  add5 = adder(5)
  add5(add2(1))
```

A closure is affine: it can be called at most once, even when everything it captures is `Data` . Only top-level definitions can be called freely. Partial applications like `U32.add(2)` are closures too.

RECURSION AND TERMINATION

Bend uses recursion to repeat work. Tail calls compile to loops.

```
import Base

def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}:
      acc
    case Con{h, t}:
      sum(t, (acc + h : U32))

def main() -> U32:
  sum([1, 2, 3, 4], 0)
```

Here, `t` has one fewer element than `xs`, so `sum` eventually reaches the empty list. Bend verifies termination by requiring recursive calls to use smaller parts of their inputs, obtained through pattern matching. The check reads the arguments of a recursive call from left to right: each must be passed unchanged until one is a smaller part of its parameter, and the ones after it are free. So, put the parameter that shrinks first. Also, is no `if` syntax yet. Use `match` on `True{}` and `False{}` instead.

Termination is mandatory and mutual recursion is not allowed. Both restrictions keep Bend's proofs sound, as a function that never returns could otherwise prove anything. A loop bounded by the outside world, like a server's, counts down a `Nat` fuel argument instead, and two mutually recursive functions become one def with an extra argument selecting which to run. A `def` marked `@unsafe` recurses freely and may call a def written below it, but falls outside Bend's proof guarantees: `bend` runs it, but a check prints SOME PROOFS FAIL and names every def that relies on it. Types are not code, so the order binds only defs: two datatypes, or a datatype and a type-level def, may name each other in any order.

A `match` inspects a parameter or a variable bound by a pattern, never a computed value: `match sum(xs, 0):` is rejected. Scrutinees follow binder order, and a `let` may not precede a `match` on a parameter. To match on a computed value, pass it to a helper that matches on its parameter.

PARALLELISM

Bend's parallelism primitive is the parallel call notation:

```
import Base

def pow2(+n: Nat) -> U32:
  match n:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p) # parallel call
      (a + b : U32)

def main() -> IO(Unit):
  IO.print(U32.show(pow2!(20n))) # `!` runs on GPU
```

A parallel call promises the compiler two things:

1. The calls are independent.
2. They run in roughly the same time.

Since Bend is pure and affine, the first point always holds. The second is yours to keep: if one call finishes before the other, the speedup will be sub-ideal. Bend's current scheduler is a contention-free, binary fork-join machine: every task is handed to a core exactly once and never moved afterwards. That makes it fast and GPU-friendly, but you must keep the workload balanced.

A `!` after a function name marks a parallel call: `pow2!(20n)` hands that call, and every parallel call inside it, to the GPU. When compiled to a native executable, `pow2(20n)` runs in parallel on the CPU, while `pow2!(20n)` runs on the GPU. The heap is fully unified, so, if your chip has unified memory (as in Apple M-series processors), moving data from the CPU to the GPU is a zero-cost operation. The GPU shines on uniform numeric work like mandelbrot or nbody; divergent work like n-queens stays faster on the CPU. A machine without a GPU runs `!` on the CPU (still in parallel). What the lanes share also sets the speed: a `+` value read by every lane costs an atomic per read. Read `bend guide shaders` before you write a parallel app.

The JavaScript target ignores all that and just runs sequentially.

ARRAYS

Arrays give Bend in-place mutation without giving up purity:

```
import Base

def main() -> Array<U32> & U32:
  a = [0 : U32*8n] # new array with 8 copies of 0
  a[5] <- 42       # performs an in-place rewrite
  a[5]             # reads index 5
```

An `Array<T>` is a `Type`, so it has exactly one owner at all times. That is what lets `a[5] <- 42` overwrite the slot and hand back the same array, with no copy. A read hands the array back next to the element for the same reason: if it returned only the element, the array would be gone. Indexes wrap around. A write followed by another statement re-binds its array: `a[5] <- 42` on its own line is `a = a[5] <- 42`. As the last statement it is the written array. The slot count after `*` is a power of two; `[0 : U32^3n]` names the depth instead.

The `a[i]` sugar assumes `Array<U32>`. For other element types, call `Array.get` (`Data` elements; else `Array.swap`) and `Array.set` directly, and `Array.clone` when you need two copies. Read Bend's Base for reference. This will be generalized soon!

QUANTITIES

A quantity says how many times a variable may be used.

```
import Base

# -A: erased (gone at runtime)
#  n: affine (used at most once)
# +x: reusable (requires A to be Data)
def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n:
      Nil{}
    case 1n+p:
      x <> replicate(A, p, x)

def main() -> List<U32>:
  replicate(U32, 3n, 7)
```

Erased variables can only appear in types and proofs: the checker sees them, the compiler deletes them. Affine variables are the default, and dropping one is always free. Reusable variables require `Data`: functions, arrays and IO handles are `Type`, so they can never be copied. Note that `main` marks nothing: you write `+` where you need the copies, and `replicate` pays for them with a reference count at runtime. Matching a `+` value hands out `+` fields; on a plain one, write `+r` in the pattern to make a field reusable.

KINDS

Every type has a kind, which caps how many times its values may be used.

```
import Base

#  a: a quantity (&0, &1 or &2)
# -A: a type whose values may be used a times
def length(a, -A: Kind(a), xs: List<a, A>) -> Nat:
  match xs:
    case Nil{}:
      0n
    case Con{h, t}:
      1n+length(a, A, t)

def main() -> Nat:
  length(&2, U32, [1, 2, 3])
```

`Type` is short for `Kind(&1)` and `Data` for `Kind(&2)`, so a `Kind(a)` parameter accepts both: `length(&1, U32 -> U32, fs)` counts a list of closures just as well. A bare `a` in a parameter list is short for `-a: Quant`. Base declares `type List<a, -A: Kind(a)> is Kind(a)`, making a list exactly as reusable as its elements: `List<U32>` is short for `List<&1, U32>`, and `+List<U32>` for `List<&2, U32>`. The short form needs the type declared above it: a type named before its declaration spells every parameter, quantities included. A type holding two element types combines their quantities with `a <&> b`, the smaller of the two.

TEMPLATES

A template receives its argument as syntax and inlines it at compile time.

```
import Base

# ~f: substituted at compile time, not passed at runtime
def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))

def main() -> U32:
  twice(~(x => (x + 1 : U32)), 40)
```

Template parameters come first in the parameter list, and a `~` argument must be closed: it may mention top-level defs, but no local variable of the caller. Each distinct set of `~` arguments compiles to its own copy of `twice`, so `f` costs nothing at runtime and, unlike a closure, may be called as many times as you like. A template may call only templates declared above it. This is how `List.map` is written in Base.

LAWS AND PROOFS

A law states a fact that must hold. It must be proven inside a paired def.

```
import Base

# LAW: "for every x, x plus 0 equals x"
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}

# PROOF: case analysis:
# - base case: reflexivity
# - step case: induction, rewrite, reflexivity
def add_zero(x):
  match x:
    case 0n:
      {==}
    case 1n+p:
      %add_zero(p) : {1n+Nat.add(p, 0n) == 1n+_ : Nat}
      {==}

def main() -> {Nat.add(2n, 0n) == 2n : Nat}:
  add_zero(2n)
```

Laws are a critical feature in Bend, as they provide an ambiguity-free language on which humans can state precise specs for AI's to implement. That is, instead of writing a natural language prompt such as "implement a function that sorts a list", users can write precise laws like "implement a function F such that, for every list of numbers, `F(list)` returns the same numbers in ascending order". Models are then guaranteed to respond with bug-free code, since Bend will demand that they provide an actual proof.

> We envision that "law-driven development" will eventually become the way humans use AI to write and maintain large codebases, as it is the perfect middle point between having to code everything manually (laborious) and letting AI do it all via prompts without auditing a line of code (error/ambiguity-prone, unsecure).

By convention, a project keeps its laws in two files at its root. `LAWS.bend` imports the code and states the laws, each an open claim: the human writes it, the AI does not touch it. `PROOF.bend` imports `LAWS.bend` and proves each law with a def of the same name (`law sorted` is proven by `def Laws.sorted`): the AI writes it, along with the code. `bend PROOF.bend` is the gate: it prints SOME PROOFS FAIL while any law is open or false, and ALL PROOFS CHECK once every law holds. bend refuses a `PROOF.bend` that sits beside a `LAWS.bend` without importing it.

`bend PROOF.bend --verdict` checks the proofs a second time, with a small kernel that has a proof in Lean: it prints ALL PROOFS CHECK only when every def outside Base is a valid proof, which bend2 and the kernel both accept, and which relies on no `@unsafe` or foreign code. `-o PROOF.bendtt` writes the translation the kernel reads; the translation has no proof, so read it to confirm a law.

Bend has no tactics: a proposition is a type, and a proof is a def of that type. `{a == b : T}` is an equality; `{==}` proves it when both sides compute to the same term. Matching refines the goal in each case, a recursive call is the induction hypothesis, and `%e : P` rewrites with `e : {a == b : T} : P` is the goal with `_` marking `b`, and the goal becomes `P` with `a` there. `exs y: T` in a law asks for a witness, returned as `(y, proof)`. A failed step prints the expected and observed terms; `?name` prints the goal, `?TODO` leaves it open, and a law with no def is an open claim.

Since types are terms, a def may return a `Type`, like `def IsEven(n: Nat) -> Type:`, which is all dependent types are. A `match e:` with no cases closes a branch where `e : Empty`. To refute a clash like `e : {1n == 0n : Nat}`, rewrite it through a motive `disc(_)`, where `disc` sends `0n` to `Empty` and `1n+p` to `Unit`, and answer `Unit{}`. `{a != b : T}` is `{a == b : T} -> Empty`, and `Equal.sym`, `Equal.trans` and `Equal.cong` are in Base.

IO AND CONCURRENCY

Effects live in the `IO` type and are sequenced with `do` blocks:

```
import Base

def greet(name: String) -> IO(String):
  do IO<String>:
    IO.sleep(1000)            # a step
    return "Hello, " ++ name  # return: wraps a pure value

def main() -> IO(Unit):
  do IO<Unit>:
    name : String <- IO.try(String, IO.get_env("USER")) # <-: binds a result
    chan : Chan(String) <- IO.fork(String, greet(name)) # runs concurrently
    IO.print("Waiting...")
    text : String <- IO.join(String, chan)
    IO.print(text)
```

Every bind is annotated, and `x : T = v` binds a pure value in the middle of a block. A fallible effect answers `Result<&1, &1, U32 & String, A>`: `IO.try` unwraps it or exits with the error, and `IO.die` exits with your own. `IO.args` answers the command line, less the runtime's options (`+RTS` to `-RTS`): its head is the program as invoked, like C's `argv[0]`. A handle (`File`, `Socket`, `Window`) is an affine, opaque value, so every effect on one hands it back beside its result, and no program can forge or reuse one.

`TCP.listen(host, port)` and `UDP.bind(host, port)` bind the IPv4 literal `host: "127.0.0.1"` serves this machine only, `"0.0.0.0"` every interface. A bad address or a port above 65535 fails with `EINVAL`.

`Process.run(program, args, input, max_output, timeout_ms)` starts an executable directly, with literal arguments rather than a shell. It inherits the current directory and environment, writes UTF-8 `input` to its stdin, and answers `Done{(status, (stdout, stderr))}` even when the exit status is not zero. Both limits must be positive; failure to start, a timeout, or stdout and stderr exceeding `max_output` bytes together answer `Fail`. The call returns once the child exits and stdout and stderr close, so a descendant holding a pipe open delays it, up to the timeout. Timeouts do not kill descendants. A native build runs the child on an IO helper thread; the JavaScript lane blocks while the child runs. On Linux, native builds need glibc 2.34 or newer to close inherited descriptors. It does not sandbox the child: callers must whether a command is trusted before executing it.

A Bend program is a set of computations interleaved by one event loop, as in Node.js: one at a time, each runs its pure code, on every core through its parallel calls, up to its next effect, and one that waits on a socket, a sleep or a channel steps aside for the others. `IO.fork` starts a computation and returns the channel its result will arrive on; `IO.join` waits for it. Underneath are `IO.spawn`, `Chan.new`, `Chan.send`, `Chan.recv` and `Chan.close`. `IO.within(A, ms, act)` races `act` against a deadline and answers `None{}` if the deadline wins; the loser is not cancelled. The program ends when every computation is done, or reports a deadlock when the remaining ones all wait.

Every effect in Base is a def whose body is `import "./x.js"` plus a `.c` twin, implemented by a host function named after the def, lowercased, dots to underscores. You can add your own effects the same way. Only the event loop runs them, so proofs, termination and the GPU never touch host code. In the other direction, a JS file may `import Game from "./game.bend"` (with `bend2/main.ts` preloaded), or from the `./game.mjs` that `-o game.mjs` writes, and call every non-IO def, with constructors as `{$: "Name", field: value}` and `Nat` as a number. A value crosses without a copy: an `Array` argument is the caller's own array, updated in place, so copy it first if you keep it.

MONADS

The `do` notation works for any monad, not just IO.

```
import Base

def add_strs(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a) # a None here ends the block with None
    y : U32 <- U32.read(b)
    return (x + y : U32)

def main() -> Maybe<&2, U32>:
  add_strs("40", "2")
```

A `do M<xs.., R>:` block desugars each `x : A <- v` into `M.bind(xs.., A, R, v, x => ..)` and each `return e` into `M.pure(xs.., R, e)`, so any type with those two defs works: `IO`, `Maybe`, `Result`, or your own. The leading arguments (here, the `&2` quantity) are passed along to both.

APPS

Graphics in Bend are pure: a frame is an `Image`; `App` maps states to images.

```
import Base

# tick: folds a frame's events into the next state; None quits the app
def tick(events: List<Event>, color: U32) -> IO(Maybe<U32>):
  match events:
    case Nil{}:
      IO.pure(Maybe<U32>, Some{color})
    case Con{Close{}, rest}:
      IO.pure(Maybe<U32>, None{})
    case Con{e, rest}:
      tick(rest, (color + 1 : U32))

def main() -> IO(Unit):
  # view: returns the state beside its image; Pix paints the whole frame
  App.run(~U32, ~App{+s => (s, Pix{s}), tick}, "Hello", 256, 256, 0)
```

An `Image` is a quadtree: `Pix{color}` paints a square, and `Qua{tl, tr, bl, br}` splits it in four, so a frame is drawn by recursion like everything else, in parallel if you want. The events are Base's `Event`, each explained in `bend2/base.bend`: a pointer (the mouse, a finger or a pen) comes as `Press`, `Move`, `Leave` and `Lost`, and a key or button held as the window loses the focus comes up. For a first-person camera, `Window.grab(window, True{})` hides and holds the cursor, and the mouse's motion comes as `Look{dx, dy}` until `Window.grab(window, False{})` or the window losing focus lets it go. `App.run` opens a window and calls `view` then `tick` once per frame, until `tick` answers `None`. Since the state is affine, `view` must hand it back next to the image. Underneath are `Window.open`, `Window.frame` and `Window.close`, and `Audio.open`, `Audio.write` and `Audio.close` for sound. See `demos/app_pong_game_2d` for a complete one.

THE BASE LIBRARY

Base is small, and its names follow a scheme, so you can guess most of it:

```
import Base

def main() -> String:
  +a = (6 * 7 : U32)         # sugar for U32.mul(6, 7)
  b  = U32.to_nat(a)         # conversions are T.to_x and T.from_x
  U32.show(a) ++ " = " ++ Nat.show(b)
```

Every def is named `Type.verb`, and the same verbs recur across `Nat`, `U32` and `F32`: `add sub mul div mod` for arithmetic, `and or xor not shl shr` for bits (`U32` only), `cmp` (returning `Cmp`, not on `F32`) and `is_eq is_ne is_lt is_le is_gt is_ge` (returning `Bool`) for comparisons, `show` to `String` and `read` back from it (answering a `Maybe`). Operators and `<-` style comparisons are just sugar for these. Beyond numbers there are `Bool`, `Cmp`, `Maybe`, `Result`, `List`, `Array`, a string-keyed `Map` (`new set get has del keys`; `get` takes a default, and `get` and `has` hand the map back beside their result), `Set` on top of it, the `Equal` lemmas, and the effects. `bend base` prints all of it, `bend base --types` only the types, and `bend base Map` one name and everything under it.

MODULES

A module is a file, and an import gives it a local name:

```
# math.bend
import Base

def square(+x: U32) -> U32:
  (x * x : U32)
```

```
# main.bend
import Base
import ./math.bend as M  # M.x now names every def of math.bend

def main() -> U32:
  M.square(7)
```

The alias is local to the importing file, and dots inside a name are just characters: `U32.show` needs no module. A module's path is plain names (letters, digits, `_` and `-`, no digit first): `math.bend` is a module, `math.extra.bend` is refused. A law left open in one file may be filled in another as `def M.name(..)`, so a proof can ship separately from its claim. `import 0x<hash>/main.bend as P` imports a package by content hash, fetched from the hub and checked against it; `bend main.bend --publish` uploads a file with everything it imports and prints that line. `import <name>@<version>/main.bend as P` is the same package by the name its author gave it on the hub, with `bend main.bend --publish <name>@<version>` after `bend login`.

A publish is public and permanent, under BendHub's terms (https://bend-lang.com/bendai/terms#s18 (https://bend-lang.com/bendai/terms#s18)). Put a `LICENSE` file next to your entry file, ideally opening with a line like `SPDX-License-Identifier: MIT`; `--publish` takes every file named exactly `LICENSE` beside a published file, and a package without one is MIT-0. You are responsible for what you publish, so pick the license it may carry. Adding a `LICENSE` changes a package's hash: publish it as a new version.

TOOLING

Bend is a single command:

```
bend file.bend            # check; run main (IO compiled; a value normalized)
bend file.bend -o file    # compile to a native binary (clang 14+; 19+ with `!`)
bend file.bend -o file.c  # emit the C source instead
bend file.bend -o file.js # emit the JS source instead
bend file.bend -o f.mjs   # emit an ES module of its non-IO defs, for JS to import
bend file.bend --verdict  # check; then recheck with the proven BendTT kernel
bend page.html -o dist    # bundle a web page that imports .bend files
./file +RTS --threads 8   # run a native binary on 8 CPU threads
./file +RTS --gpu 4GB     # cap the GPU heap at 4GB; off runs ! calls on the CPU
```

A `main` that returns `IO` runs compiled; one that returns a value is normalized by the checker (slow for big work) and printed; a file with no `main` just checks. A binary that uses `!` builds its GPU program too, as `file.gpu`, which must stay beside it: on macOS it needs Metal, on Linux CUDA 12 or 13 at `$CUDA_HOME`, `/usr/local/cuda` or `/opt/cuda`. On Linux a program with a Window needs `libx11-dev`, one with Audio `libasound2-dev`. `bend guide` prints this text, `bend base` prints the Base library (`bend base Map` prints one name and everything under it), and `bend --help` lists the other commands.

SYNTAX REFERENCE

Every form of the language, grouped by where it appears. Operators, literals and brackets are sugar for names in Base.

```
# Top level
import Base                              # the prelude
import ./file.bend as M                  # a module; its defs are M.x
type D<a, -A: Kind(a)> is Kind(a):       # a datatype and its kind
  K{x: A, xs: List<a, A>}                # one constructor per line
def f(x: A, -y: B, +z: C) -> T:          # a def; the body follows
def f(x, y):                             # fills the law named f
def t(~g: A -> B, x: A) -> B:            # a template
law f:                                   # a claim, proven by def f
  for x: A                               # a parameter (also for -x, for +x)
  for y: B where P(y)                    # y is then the pair (y, P(y) proof)
  exs z: C                               # a witness the proof must return
  T                                      # the claim
@unsafe def f(x: A) -> T:                # skips the termination check
def f?(x: A) -> T:                       # the same, as a sugar
def e(x: A) -> IO(B):                    # a foreign effect
  import "./e.c"
  import "./e.js"
```

```
# Types
Type  Data  Kind(q)                      # kinds; Type = Kind(&1), Data = Kind(&2)
Quant  &0  &1  &2  a <&> b               # quantities and their minimum
A -> B  @x:A -> B  @-x:A -> B            # functions: plain, dependent, erased
A & B  &x:A -> B  A | B                  # pairs, dependent pairs, sums
D<A>  +D<A>  D<&2, A>                    # a datatype; + makes it reusable
{a == b : T}  {a != b : T}               # equality and its negation
```

```
# Terms
42  1.5  3n  'c'  "s"                    # U32, F32, Nat, Char, String
3n+e                                     # e plus 3, a Nat; no spaces; e runs to the end
[a, b]  h <> t  (a, b)                   # a list, a cons, a tuple
K{a, b}  x => e  +x => e                 # a constructor, a lambda
f(a, b)  f!(a)  t(~g, a)                 # a call, on the GPU, of a template
(a + b * c : T)  {x : T}                 # operators over T; an annotation
[v : T*n]  [v : T^d]  a[i]  a[i] <- v    # an array of n or 2^d slots; a read, a write
{==}  %e : P; e2  %e@E : P; e2           # reflexivity, a rewrite, a named one
?name  ?TODO                             # print the goal; leave it open
```

```
# Statements (a def, case, lambda or parenthesized body)
x = v  +x = v  -x = v                    # a let: affine, reusable, erased
(a, b) = v  K{x, y} = v                  # a destructuring let
a b = f(x) g(y)                          # a parallel let
match a b:                               # a match on one or more values
  case K{x, _} 1n+p:                     # patterns nest; _ catches the rest
do M<xs.., R>:                           # a monadic block over M.bind, M.pure
  x : A <- m                             # bind
  x : A = v                              # let
  m                                      # a Unit step
  return v                               # the result
```

Inside `(.. : T)`, `+ - * / %` call `T.add` through `T.mod`, `.&. .|. .^.` the bit operations, `<< >>` the shifts (by a `Nat`), and `< <= > >=` the `T.is_lt` family; without a `: T` they are refused. `&& ||` work on `Bool` and `++` on `String` anywhere. Operators need spaces on both sides. Equality of values is a call, `T.is_eq(a, b)`; `==` is only the type. A `Nat` literal past `256n` is `U32.to_nat(n)` underneath, up to `4294967295n`.

UNDER THE HOOD

Bend's compiler emits one C file, and that file is both the CPU program and the GPU kernel. clang compiles it for the CPU. Metal (on Apple) or CUDA (on NVIDIA) compiles the same file for the GPU. So a `!` call runs the same code on whichever chip it lands on.

A term is one 64-bit word: small values are stored inline, everything else is a pointer into a single heap shared by every core and by the GPU. There is no garbage collector. Since values are affine, a `match` frees the node it opens on the spot, but copying any value of a datatype makes all its nodes count their fields. There is no C stack either: each def compiles to a segment of a flat state machine, a call is a jump, and a parallel call creates a join task plus one task per call, which the scheduler deals across CPU or GPU lanes. `paper/BendRT.pdf` has the design and the benchmarks.

Bend's theory has one universe and no positivity check: `Type : Type` holds, and a datatype may recurse on the left of an arrow. What keeps this consistent is a wall between two checking modes. Code that runs is checked *live*; types, erased arguments and equations are checked *dead*. Dead code may loop forever or inhabit `Empty`, but nothing dead ever counts as live evidence, and live recursion must terminate. `bend2/bendtt.lean` is BendTT's kernel in Lean, with a proof that no def it accepts has type `Empty` and that live code halts; `--verdict` checks a file with it. `paper/BendTT.pdf` is the paper.

FURTHER READING

- `demos/` : complete programs, including the game and its proof from the video.
- `bend2/base.bend` : the Base library, also printed by `bend base`.
- `paper/BendTT.pdf` and `paper/BendRT.pdf` : the type theory and the runtime.

EXTRA

`bend guide shaders` prints "Shaders in Bend", a tutorial written by AIs for AIs on how to write efficient shaders in Bend. It distills what building `demos/app_slash_boss_3d` (120 FPS in pure Bend) taught. Read it before you write a graphical or parallel app in Bend.

`bend guide effects` prints "Effects in Bend", an AI-written note (to be revised by a human) on the C and JS side of custom effects. Read it before you write one.

H.3 · bendlang/bend · guide/EFFECTS.md

Bend guide/EFFECTS.md (fetched 2026-11-06, main)

Effects in Bend

An AI wrote this text from the runtime's source (`bend2/comp.ts` and `bend2/effs/`). A human will revise it later. Report anything wrong on GitHub.

AN EFFECT DEF

An effect is a def of type `IO(R)` whose body is two imports:

```
def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"
```

Each file registers the effect under its def with `io_eff(CID(Clock.now), ..)`. The `.c` file serves `bend x.bend -o x`; the `.js` file serves `bend x.bend -o x.js` and `bend x.bend`. Every effect in Base is built this way: `bend2/effs/*.c` and `*.js` are the reference.

THE C SIDE

The compiler splices your `.c` into the program's C source after the runtime, so every runtime symbol is in scope. Define the effect and register it in a constructor:

```
Term clock_now_run(Env e, Term* f, IoWork* w);

static void __attribute__((constructor)) clock_now_use(void) {
  io_eff(CID(Clock.now), clock_now_run);
}
```

`CID(Name)` is the C id of a constructor or of an effect def: the compiler replaces it with the name's own id, reading `Name` in the effect's namespace first (the file that declares the def), then in Base's, as the def's body would. A constructor the program does not use has no id, so `#ifdef CID(Name)` tests for it. `FID(name)` is a def's function id the same way. `f` holds the def's arguments in order: a `U32` is the word (`(u32)f[0]`), a `String` is taken with `io_cstr(e, f[0], &len)` (a `malloc`ed copy you free), a packed handle with `io_hand_v(f[0])`. The loop calls the effect at once; an effect that waits (for a handle, or for time) parks itself, as below.

The effect returns a Term: a `U32` is `(Term)n`, `Unit` is `term_pak(CID(Unit), 0)`, a `String` is `io_str(e, p, n)`, a two-field constructor is `io_node(e, CID(K), a, b)`. A `Result` is `io_done(e, v)` or `io_fail(e, code, text)` (`text` NULL prints `strerror(code)`). `io_sys_end(w, n)` stores `errno` in `w->code` when a call fails.

Descriptor and window handles pack one word with `io_hand(v)`. Their type must be one of Base's handle laws (`File`, `Socket`, ...): a user handle type is a WONTFIX entry for now, see `WONTFIX.txt`. An effect on a handle hands it back beside its result: `io_tup(e, io_hand(h), r)`, also on failure.

A handle owns its descriptor, on both sides. An effect that answers a new handle gives its fd away open: it does not close it later or keep its number. An effect that takes a handle borrows the fd and hands back the same one, still open; only a close effect (`Socket.close`, ...), which consumes the handle, closes it. A descriptor that never becomes a handle is the effect's own, closed on every path, as `tcp_connect.c` does when its setup fails. Since handles are affine, this keeps every parked fd open: an fd closed under the loop is a bug in the effect that closed it, and the number it freed may already belong to another open.

`Chan` instead owns a native heap block containing its row and buffered Terms, wrapped in a reference count at birth so a shared Data cell may hold it. Handle copies use the ordinary native reference count. Close wakes waiters but preserves buffered values for copied handles; dropping the last handle releases the block and unread values. A parked channel operation retains its input handle until it wakes or times out. The C effect consumes that handle, rather than returning it beside the result. Channels share the native heap's reference-counting lifetime model, not a tracing garbage collector.

Blocking work leaves the loop in two ways. `io_work(w, call, pack)` runs `call(w)` on a helper thread, then `pack(e, w)` on the loop; `pack`'s value is the answer. `call` has no `Env`: it may only touch `w`. `io_wait_on(w, fd, POLLIN, 0, more)` parks until `fd` is ready (`POLLIN` or `POLLOUT`), then runs `more(e, w)` on the loop; `more` may call `io_wait_on` again. Both return `IO_PARK`, which the effect returns. Replace `0` with an absolute `io_tick()` deadline to also wake on time; with no events (`io_wait_on(w, 0, 0, deadline, more)`) it waits on time alone, as `sleep.c` does. An effect that can meet a steady backlog (an accept, a datagram recv) parks before its first try, as `tcp_accept.c` does: one that answers at once never lets the loop poll, so the backlog would starve its timers.

`w` is the effect's scratch space: `hand`, `made`, `size`, `data`, `text`, `code`, and `word` until it parks (`io_wait_on` keeps its fd there). The runtime owns `w` and `f`. The effect owns `w->data`: allocate it in the run function and free it in `pack`. In `bend2/effs/`, `file_read.c` is the pattern for `io_work` and `tcp_recv.c` for `io_wait_on`.

THE JS SIDE

The `.js` file is a plain script, run once in a closure of its own, that registers the effect: `io_eff(CID(Clock.now), clock_now)`. The effect takes the def's arguments as JS values: a `U32` is a number, a `Nat` a number, a `String` a string, a constructor `{$: CID(Name), field: value}`, a handle its host value (a descriptor). `CID(Name)` is the constructor's tag, read as on the C side. It returns the answer the same way: `io_done(v)`, `io_fail(code)`, `io_tup(handle, result)`, `{$: CID(Unit)}`. A `Nat` answer that is not a whole number from 0 to 2^53 stops the run; a `Nat` inside an answer goes unchecked, so the host keeps it in that range.

A blocking effect takes one more argument, `k`, and parks with `io_park_on(fd, out, k, more)`: it returns `undefined`, and the loop calls `more()` when `fd` is ready; `more` answers the value, or `undefined` to park again. Add an absolute `performance.now()` deadline as a fifth argument to also wake on time; with no fd (`io_park_on(undefined, false, k, more, deadline)`) it waits on time alone, as `sleep.js` does. An effect that can meet a steady backlog parks before its first try, as on the C side. `io_sys()` is `libc` through `bun:ffi` (`read`, `recv`, `select`, `errno`); `tcp_accept.js` shows the full shape.

A COMPLETE EXAMPLE

`main.bend`:

```
import Base

def Clock.now() -> IO(U32):
  import "./clock.c"
  import "./clock.js"

def main() -> IO(Unit):
  do IO<Unit>:
    t : U32 <- Clock.now()
    IO.print("ms: " ++ U32.show(t))
```

`clock.c`:

```
Term clock_now_run(Env e, Term* f, IoWork* w) {
  return (Term)(uint32_t)(io_tick() / 1000000ull);
}

static void __attribute__((constructor)) clock_now_use(void) {
  io_eff(CID(Clock.now), clock_now_run);
}
```

`clock.js`:

```
function clock_now() {
  return Math.floor(performance.now()) >>> 0;
}

io_eff(CID(Clock.now), clock_now);
```

`bend main.bend -o main && ./main` prints a line; so do `bend main.bend -o main.js && bun main.js` and `bend main.bend`. The output name must not be one of the effect's files. The two sides count from different zeros: `io_tick()` reads `CLOCK_MONOTONIC`, whose zero the OS picks (on Linux, boot), and `performance.now()` starts with the process. Only the difference between two reads means the same on every lane.

COMPATIBILITY

The C side tracks the exact compiler version: the names above are the runtime's internals, and a release may rename any of them. There is no ABI promise. Rebuild your effects with every update.

H.4 · bendlang/bend · guide/SHADERS.md

Bend guide/SHADERS.md (fetched 2026-11-06, main)

Shaders in Bend

AIs wrote this tutorial, for AIs, from the logs of building `demos/app_slash_boss_3d`. A human will revise it later. Report anything wrong on GitHub.

A fast shader in Bend is one fork tree that ends in flat loops. The reference is `demos/app_slash_boss_3d/bend3d.bend`, a rasterizer that draws 1920x1200 at 120 FPS (8.0 ms a frame) on a 10-core M4 mini. Every ms below is that machine and that frame unless stated.

THE MACHINE

A `!` call (a bang) ships one call to the GPU, which runs a cube of 16384 lanes (128 x 128). A parallel let `a b = f(x) g(y)` creates one task per call plus a join task. A kernel iteration grows the fork tree breadth-first until the frontier fills the cube, then every lane runs its task to the end: ~0.4-0.5 ms plus the slowest lane. A continuation that forks again lands in the next iteration. A def is flat when it has no parallel let, no bang, no closure apply, no non-flat call and no non-tail self call: it compiles to one native tail loop (`spin_N` in the C). A non-tail self call becomes a segmented continuation: a frame per call. Memory is one heap shared by the CPU and the GPU: unified on Apple; managed on CUDA, where each page faults across PCIe on the first touch by the other side. A machine with no GPU, or `+RTS --gpu off`, runs bangs on the CPU pool. Divergent per-lane work (a search, a tree walk per ray) is faster there than on the GPU. Aim for 4^7 leaves per bang, one per lane, and sweep one fork level either side on the device: the CPU pool is flat across that sweep, so it cannot tune it. Fewer leaves leave lanes idle (256 of 16384 busy: 234 ms for a frame that takes 28 ms at 4096); more cost an iteration each.

A FRAME

1. Host: build the scene (mesh, project, light, cull) into a quadtree of 64-px cells, one cons list of flat screen triangles per cell. 1.6 ms.
2. One bang: fork four ways per node, five levels to the cells, two more over each cell's list, to 16-px tiles: 4^7 = 16384 = the lane cube. A tile filters the cell's list into its own and walks it in flat loops. Draw ~6 ms.
3. The last frame rides down the same tree as `Four` quadrants; each tile leaves its quadrant unused, so the runtime sinks it there. Drop 0 ms.

4. Host: free the last scene as one branch of the next build. 0.35 ms.

```
 2048 root   Frame.node, k=5   fork 4   levels 1-5  -> 64-px cells (32 x 32)
64-px cell  Cell.fork, k=2    fork 4   levels 6-7  -> 16-px tiles (16384)
16-px tile  Tile.b16 = 4 Tile.b8 = 4 Tile.b4       straight-line, no fork
4-px square Blk.go            one flat loop over the list, four Blk
2-px block  Blk.four          one Pix, or four
```

The root is the power of two over the frame (2048 over 1920x1200); a region off the frame, or empty, answers one `Pix` before any fork (`Frame.sky`). A fork per pixel drowns in scheduling; a fork per cell leaves lanes idle. A fold per level plus a fold per pixel: 600 ms and 31 iterations a frame, against 8 ms and 1.

THE CODE

The list is the one shape a flat loop walks with no stack. One list per cell, not per pixel: a list shared by pixels is contended atomics.

```
type Tris is Data:
  TNil{}
  TCons{+tri: Tri, next: Tris}

type Cells is Data:
  CNil{}
  CLeaf{+n: U32, +ts: Tris}
  CQua{tl: Cells, tr: Cells, bl: Cells, br: Cells}
```

The fork. `Frame.show` bangs once and returns the tree beside the image, so the host still owns it. `Image.open(old) -> Four` splits the last image at the call (a `Pix` opens into four); each tile receives `old: Four` and ignores it.

```
def Frame.node(+k: Nat, c: Cells, old: Four) -> Image:
  match k c:
    case _ CLeaf{_, ts}:
      Cell.fork(2n, ts, 0, old)
    case 1n+e CQua{tl, tr, bl, br}:
      Four{oa, ob, oc, od} = old
      a b c d = Frame.node(e, tl, Image.open(oa)) Frame.node(e, tr, Image.open(ob))
        Frame.node(e, bl, Image.open(oc)) Frame.node(e, br, Image.open(od))
      Qua{a, b, c, d}
    case _ _:
      Pix{0}

def Frame.show(+cells: Cells, old: Image) -> Image & Cells:
  (Frame.node!(5n, cells, Image.open(old)), cells)
```

`Cell.fork(+k, +cs, ..)` has the same shape over the cell's list and calls `Tile.go` at `0n` . The tile: one flat loop reads each entry once for a block of `Hit{w, c, i}` , then frees its list after the last read.

```
def Blk.go(ts: Tris, +i: U32, +h0: Hit, +h1: Hit, +h2: Hit, +h3: Hit) -> Image:
  match ts:
    case TNil{}:
      Blk.four(h0, h1, h2, h3)
    case TCons{t, n}:
      Blk.go(n, (i + 1 : U32), Hit.step(t, i, h0), Hit.step(t, i, h1),
        Hit.step(t, i, h2), Hit.step(t, i, h3))

def Tile.go(cs: Tris, +x: U32) -> Image:
  +h = {Hit{0.0, 0, 0} : Hit}
  +ts = Tris.filter(cs, x, TNil{})
  Tile.fin(Blk.go(ts, 1, h, h, h, h), ts)
```

`Blk.four` answers `Pix{k0}` when the four hits are one triangle, else a `Qua` of four `Pix` ; `Tile.fin` walks the list to `TNil` and returns the image. Transparency: the filter fills a second list (`Filt{+n, ts, bs}`) and `Blend.go` walks it in order over the blocks after the opaque loop. Text and HUD: strokes as opaque quads nearer than the world (`Text.put` , `Quad.put` at inverse depth 3000), layered by depth, order-free. The host: one parallel let per def, the last scene freed as a branch by `Cells.drop(4n)` : four forked levels in `Frame.node` 's shape, then `Tris.free` loops.

```
def Scene.build(+t: F32, used: Cells) -> Cells:
 a b c = Hero.build(t) Stage.build(t) Cells.drop(4n, used)
 Cells.m4(a, b, c, CNil{})
```

A turn keeps the last image and scene as one pair: the build takes the old scene, the bang the old image. An `IO.now()` step sits between them: a bang that follows the host's forks in one evaluation runs on the CPU pool. `Window.frame(window, image)` shows the image.

```
def Turn.go(+t: F32, last: Image & Cells) -> IO(Image & Cells):
 (old, used) = last
 do IO<Image & Cells>:
   cells : Cells = Scene.build(t, used)
   now : Nat <- IO.now()
   IO.pure(Image & Cells, Frame.show(cells, old))
```

ANY SCENE

The pixel never sees the scene. The fork tree opens each node once on the way down and hands each tile one short flat list; the pixel loop marches that list. For triangles, voxels, distance fields and sprites alike, the host walks the world once per frame into per-tile candidates (the primitives, columns or spans whose screen bounds meet the tile); the device reads only those. A field with no scene data (a procedural distance field) is the best case: one flat loop per pixel on a U32 step counter. A per-ray tree walk fails three ways: a fold over a tree is a non-tail recursion (a frame per node), the lanes of a SIMD group diverge, and a `+` tree pays an atomic per node per pixel. A reported case: a voxel world as a `+Data` quadtree of 32x32 columns, read per ray: 176 ms a frame, against 24 ms for the same picture computed per pixel with no scene data (the reporter's machine). The fix: keep the editable map on the host, and hand each tile a flat list of the column words its rays cross (a 32-high column is one U32 of bits) each frame. An `Array` has one owner, so it cannot go down a fork tree: build lists.

DATA

- Records of scalars ride in registers through every call and fork; a boxed record shared by every vertex is an atomic count per use. `Tri` is 24 flat words (barycentric, depth and colour planes, mode, alpha, box): one F32 per 64-bit word, ~200 B loaded per entry per walk.
- A wide record round-trips the stack: a 52-word record per list entry made the draw 6.3 -> 12.1 ms; a 22-word camera on every mesh call, +2-4% build. Build wide data where used (`Hook.cam(F)`).
- A def or continuation takes at most 247 words ("an arity over 247"): four 15-field records in one parallel let fail to compile. Wrap a big header in a constructor of the tree it heads.
- Use the typed `pick(c, a: F32, b: F32)` and `word(c, a: U32, b: U32)` ; the generic `Bool.pick` boxes its words.
- Tables are `word` chains and Bool-column matches. A `match` on U32 literals is a bit-by-bit tree with the default arm copied per branch: +21% host C, +6.4% `.gpu` , 1-4% slower.
- Pass a constant as a `~` template argument (a def like `Screen.w()`), not as a parameter: a parameter rides in every task.
- Tessellate in screen space: ~20-px patches; a circle gets `max(4, ceil(pi sqrt(r/2)))` segments. Slivers and sub-pixel triangles pass every box test and pile 100-400 entries on a tile (42 ms frames); exact triangle-vs-cell overlap holds the worst tile at 80-110. Length past that is not the cost: a cap of 16/32/64 entries per tile changed nothing.

OWNERSHIP

The compiler decides borrows; `+` does not. The checker needs `+` for a second use. The compiler borrows a boxed parameter (not an `Array`) that the def only matches or passes to a borrower: every read is `term_peek` , no count. It owns one that the def returns, stores in a constructor, or passes to an owner. An owned use of a value that is used later shares it: a count on the value and, since sealing is per type, on every node of that type, silently. So `+ts` above borrows: `Blk.go` only matches, `Tile.fin` consumes last. So the scene is host-owned, borrowed by the bang, and returned (`Frame.show`). Read through counts, the frame costs 8.0 -> 16.5 ms (draw 6.0 -> 14.1).

- Two consumers of one list share it: 45 ms frames and a 448 s Metal compile. A `+` where one use would do heats its type (symreg 2.83 -> 3.30 s).
- A def that returns its argument shares it: `Bool.pick(Cloth, c, a, b)` on a tree gave 14 keeps and a hot type. Match the selector and recurse into one field (`Cells.put` has eight arms for this). Refolding a matched `+` record at a call is an owned argument (8-16 keeps): pass the fields.
- The last reader of a borrowed list needs a later use, else it becomes the owner: `Tile.fin` walks the lists after the pixels read them (as dead parameters the runtime's sink is 0.55 ms slower).

- A bang inside a `do` continuation flips its callee to owner (the tree is a captured local): 2x reads. Bang in a pure def that returns the pair.
- Drop the last image inside the fork tree (a dead `Four` per tile) and the last scene on the host by a walk (35% faster than the runtime sink); never inside a bang: 12.9 ms. `_ = x` drops a value on the spot.
- Never let the device own the scene: draw 6.2 -> 12.0 ms.

Verify in the emitted C (`bend x.bend -o x.c`): in a def's body, `term_peek` is a borrow; `term_keep` , `ctr_take` and `rfc_seal` are counts. A new keep in a device def is a regression.

COST MODEL

- Mesh, project, light and cull on the host: 0.6-1.6 ms in parallel lets; on the GPU 14-33 ms (a P-core is ~75x a lane on serial work).
- Leaves are flat loops over cons lists; joins are constructors. Unroll the squares: `Tile.b16` = 4 x `Tile.b8` = 4 x `Tile.b4` ; folded into one recursive def they made the draw 5.5 -> 12.6 ms.
- Walk the list once per 4x4 square, not per pixel or per 2x2 block: 6.6 -> 5.6 ms. Anything that depends on x alone, once per column pair: sky per pixel 3.8 ms, per block 1.6, per column pair ~1.0 less.
- The draw is throughput-bound, not tail-bound: splitting heavy tiles +0.03 ms, 32-px cells +0.85. A per-lane early-out saves nothing while a SIMD sibling runs the body.
- Drops: `Image.drop` forked at every `Qua` (~100 ops of task machinery per node), 2.6 ms; base's `Image.free(7n)` , seven levels then a sink, 1.6 ms; inside the draw's tree, 0; a whole image sunk on one lane, 19.8.
- Host: one parallel let per def (a second costs an iteration: build 2.32 -> 1.93 ms); every extra branch or level ~0.1 ms; a 0.3 ms job loses to the pool wake-up; a heavy branch serializes a 16-ring chunk (1.2-1.6 ms): split it.
- CUDA: a scene built by the host and read by the device faults each page over PCIe, both ways: 17-19 FPS on an RTX 4090 (25 ms `gpu_pass` + ~21 ms host turn) for the frame the M4 draws in 8. Keep a whole frame on one side of the bus.

MEASURING

Probe as the game does (`SLASH_PROBE=1200`): scripted frames headless, each stage's microseconds as medians after warm-ups. Single-run medians vary 0.3-0.5 ms across days, and the second binary of a GPU pair runs ~7% faster: compare by 3-4 interleaved 1200-frame pairs in both orders, plus an A-vs-A pair. Check frames pixel-identical (`SLASH_DUMP=T` prints one as a tree) before trusting a gain. Count the keeps in the C and the `.gpu` bytes (+6% was a real 1-4% slowdown). Probe without audio: it wakes `audiomxd` at 70-100% of a core.

DO NOT

1. Read the scene through counts: 2x per read, 8.0 -> 16.5 ms.
2. Fold per level, or bang twice a frame: 600 ms against 8.
3. Recurse non-tail in the bang, or fold a tree per pixel: draw 5.5 -> 12.6.
4. Mesh on the GPU: 14-33 ms against 0.6-1.6.
5. Fork per pixel or per `Qua` : drop 2.6 vs 1.6 ms.
6. Leave slivers and sub-pixel triangles: 42 ms frames.
7. Wide records through frames and joins: 2x per entry.
8. `match` on U32 literals: +21% C, +6.4% `.gpu` .
9. Generic `Bool.pick` on words or trees: boxed words, hot types.
10. Free the scene in a bang (12.9 ms) or the image on the host (milliseconds).
11. Trust one run, or the CPU pool (the 600 ms frame took 11 ms there), as a device measurement.

REFERENCE

`demos/app_slash_boss_3d/bend3d.bend` is the library; its comments are these rules at their sites. `main.bend` beside it is the app, with `Play.loop` and the probes.

H.5· fresh fetch · retrieval record

SHA-256 of the four fresh fetches + retrieval timestamp

2026-11-06 13:02 UTC 90f656cf4c053063aef38050bb59e502038b6089e34798c5c71bbec276701ca0 bendlang-bend-EFFECTS.md 6c98ebfa3cda85ca95f674d27578d4d676fcc149c77cc648f8e696a2f06f408e bendlang-bend-GUIDE.md ae282947693ad6e17f93e376cd91a7cf06e599b17b552d982985687413d49fe3 bendlang-bend-README.md 0208d6fb9973ed54811d1e7dd6b1a1d435389fa56136202c4b4e2ec7335dd166 bendlang-bend-SHADERS.md

H.6 · bend-docs/openspec/changes/bend-docs-site/research/README.md

Research sources — retrieved 2026-09-05

All via `Accept: text/markdown` mirrors (same representation as HTML).

#	File	Canonical URL	Last updated (source)
1	`sources/home.md`	https://bend2.dev/ (https://bend2.dev/)	2026-08-27
2	`sources/contents.md`	https://bend2.dev/contents/ (https://bend2.dev/contents/)	2026-08-30 (index)
3	`sources/learn-index.md`	https://bend2.dev/learn/ (https://bend2.dev/learn/)	2026-08-05 (index; entries 2026-08-13/21)
4	`sources/what-is-bend2.md`	https://bend2.dev/notes/what-is-bend2/ (https://bend2.dev/notes/what-is-bend2/)	2026-08-30
5	`sources/bend2-release-date.md`	https://bend2.dev/notes/bend2-release-date/ (https://bend2.dev/notes/bend2-release-date/)	2026-08-27
6	`sources/bend2-vs-bend1.md`	https://bend2.dev/notes/bend2-vs-bend1/ (https://bend2.dev/notes/bend2-vs-bend1/)	2026-08-30
7	`sources/hvm4-explained.md`	https://bend2.dev/notes/hvm4-explained/ (https://bend2.dev/notes/hvm4-explained/)	2026-08-21
8	`sources/bend2-vs-mojo.md`	https://bend2.dev/notes/bend2-vs-mojo/ (https://bend2.dev/notes/bend2-vs-mojo/)	2026-08-30
9	`sources/bend2-vs-lean.md`	https://bend2.dev/notes/bend2-vs-lean/ (https://bend2.dev/notes/bend2-vs-lean/)	2026-08-30
10	`sources/interaction-nets-explained.md`	https://bend2.dev/notes/interaction-nets-explained/ (https://bend2.dev/notes/interaction-nets-explained/)	2026-08-21
11	`sources/hello.md`	https://bend2.dev/learn/hello/ (https://bend2.dev/learn/hello/)	2026-08-13
12	`sources/parallelism.md`	https://bend2.dev/learn/parallelism/ (https://bend2.dev/learn/parallelism/)	2026-08-13
13	`sources/gpu.md`	https://bend2.dev/learn/gpu/ (https://bend2.dev/learn/gpu/)	2026-08-13
14	`sources/proofs.md`	https://bend2.dev/learn/proofs/ (https://bend2.dev/learn/proofs/)	2026-08-21

IMAGES

No raster `<img>` on any page. All visuals are inline SVG: footer brand icons, theme-switch illustrations, copy-button icons, and 5 hand-drawn interaction-net diagrams (`figure.diagram`) on `interaction-nets-explained` . Our docs use original hand-authored SVGs (cell, confluence diamond), not copies.

DEEP PASS (2026-09-05)

- `repos/hvm4-docs/` : HVM4 `docs/primer.md` , `docs/theory/interaction_calculus.md` , `docs/hvm/{memory,core,syntax,collapser}.md` + 62 interaction rule docs listed.
- `repos/taelin-theorem-proving-vs-testing.md` : Taelin's proving-vs-testing essay.
- Formality README (master): "proofs as developer productivity" thesis.

- `VictorTaelin/hvm4-formal` : single `main.hs` (347 lines, Haskell formalization).
- HigherOrderCO org: 8 repos total (Kind, HVM1–4, Bend, bench, .github).
- VictorTaelin (261 repos, ranked): Interaction-Calculus, Formality, Interaction-Type-Theory, Cedille-Core, calculus-of-constructions, Caramel, abstract-algorithm, optlam, articles, nanoproof, BendGen, lambench noted.

HANDS-ON VERIFICATION LOG (GCC-BUILT HVM4 @ MAIN, 2026-09-05)

Program	Result	Stats
`@add = λa.λb.(a+b); @main = @add(1,2)`	`3`	3 itrs, 23 nodes
`@main = (&A{1,2} + 10)` -C10	`11 , 12`	4 itrs, 13 nodes
`!x&A = &A{1,2}; [x₀,x₁]`	`[1,2]`	1 itr
`!x&A = &B{1,2}; [x₀,x₁]` -C10	`[1,1] , [2,2]`	5 itrs
`λ&x.(x*x)` of 5	`25`	3 itrs
fib(10)	`55`	760 itrs, 19.33M itrs/s
len via `#CON:` of 3-list	`3`	16 itrs

Stale upstream syntax found (README/primer vs current parser): label-less `&{a,b}` sup, whitespace application, `<>:` match pattern. Docs show working forms.

H.7 · bend-docs/openspec/changes/bend-docs-site/research/concept-map.md

Concept map — Bend, HVM, Bend2

Distilled 2026-09-05 from `sources/*.md + repos/*` . Every claim cites its source file; source files carry canonical URLs + dates (see `README.md`).

THE ONE-PARAGRAPH IDEA

Ordinary functional code already contains all the parallelism a machine could want — independent subexpressions can run anywhere, in any order — but conventional runtimes cannot see that without programmer annotations (threads, tasks, kernels) because shared mutable state makes order matter [`interaction-nets-explained.md`]. Interaction nets (Lafont, POPL 1990) fix this by constraining the machine model itself: a program is a graph of cells, each step rewrites exactly two cells that face each other, and such rewrites on disjoint pairs cannot conflict and always converge to the same result [`interaction-nets-explained.md`]. So parallelism stops being a language feature and becomes a scheduling decision: any idle core fires any pending pair [`hvm4-explained.md` , `what-is-bend2.md`]. Bend is the language on top (HVM2 runtime, May 2024 launch); HVM4 is the fourth runtime generation; Bend2 (unreleased, est. ≈2026-09-02) adds dependent types and synthesis on the same substrate [`home.md` , `bend2-release-date.md` , `bend2-vs-bend1.md`].

BEND (BEND1): WHAT IT PROVED, WHAT IT COST

- Python-flavored syntax with recursion + ADTs, zero parallelism annotations, saturating a GPU via the HVM2 interaction-net runtime [`bend2-vs-bend1.md` , `HigherOrderCO_Bend-README.md`].
- Backends chosen at the CLI: `run` / `run-c` (parallel C), `run-rs` (sequential Rust), `run-cu` (CUDA); `gen-c` / `gen-cu` emit standalone files [`HigherOrderCO_Bend-README.md`].
- Sequential vs parallel is a dataflow property: `start + Sum(start+1, …)` chains (sequential), tree recursion fans out (parallel) [`HigherOrderCO_Bend-README.md` , `parallelism.md`].
- Costs, all from packing everything into compact net nodes: 24-bit numbers (`u24/i24/f24`), low-GB memory ceiling from fixed-width arena addressing, minimal IO, strict-only evaluation, and a large constant factor per rewrite (graph surgery in memory) that let a tuned CPU core beat GPU throughput on many workloads [`bend2-vs-bend1.md` , `what-is-bend2.md`].

INTERACTION NETS: THE SUBSTRATE

- Cell = one constructor/operation with one principal port (apex, the only reaction site) + auxiliary ports; net = ports wired in pairs; computation happens only at active pairs (principal-to-principal) [`interaction-nets-explained.md`].

- At most one rule per unordered symbol pair; firing consumes the pair and writes a fixed net on the same free ports [`interaction-nets-explained.md`].
- One cell ⇒ one active pair ⇒ redexes are disjoint **by construction**, no analysis, no locks [`interaction-nets-explained.md`].
- Theorem 1 (Lafont 1990): one-step confluence — either firing order lands on the same net in one step; hence same normal form, same step count (work is a program property), any worker may fire any pair [`interaction-nets-explained.md`].
- Theorem 2 (Lafont 1997): three symbols (γ, δ, ε), six rules suffice for universality; annihilation (same meets same) + commutation (distinct pass through, copying one cell per step) [`interaction-nets-explained.md`].
- β-reduction becomes one constant-time rule; the hard part moves to duplication: Lamping (POPL 1990) gave optimal reduction (Lévy families), Asperti–Mairson (1998) proved full-optimality bookkeeping is non-elementary; HVM trades Lamping-completeness for constant factors, making duplication a primitive rather than transparent substitution [`interaction-nets-explained.md`].
- In silicon: a cell is a tagged machine word, an active pair a queue entry, an interaction a few loads/stores; ε does GC as local rewriting, no collector thread [`interaction-nets-explained.md`].
- Practical concession: machine integers live inside the node word; leftover width after ports/tags sets the range (Bend1: 24 bits) [`interaction-nets-explained.md` , `bend2-vs-bend1.md`].

HVM LINEAGE AND THE INTERACTION CALCULUS

- HVM1 (2022, lazy, interaction combinators, Rust) → HVM2 (strict, C+CUDA, carried Bend's launch) → HVM3 (early 2026, merged strengths, `run` / `-c` compiled flag) → HVM4 (public, single file `src/hvm.c` , `clang -O2` , pushed into mid-2026) [`what-is-bend2.md` , `hvm4-explained.md` , `HigherOrderCO_HVM1-README.md` , `HigherOrderCO_HVM2-README.md` , `HigherOrderCO_HVM3-README.md` , `HigherOrderCO_HVM4-README.md`].
- Interaction Calculus (VictorTaelin/Interaction-Calculus, 948 stars): affine vars (≤1 use), global lambdas (no scope boundaries ⇒ continuations, linear HOAS, mutable refs), first-class SUP `&{a,b}` + DUP, erasable `*` ; affinity makes GC cheap and parallelism simple [`VictorTaelin_Interaction-Calculus-README.md` , `HigherOrderCO_HVM3-README.md`].
- HVM4 surface from its README: `@main = (&{1, 2} + 10)` yields both `11` and `12` , sharing common work; DUP `(! x &A= 3; (x₀ + x₁))` feeds two uses without copying; docs (theory, core, memory, interactions) sit next to code [`HigherOrderCO_HVM4-README.md` , `hvm4-explained.md`].
- Launch blockers (2026-03-05): HVM4 AOT compiler (kills per-rewrite dispatch overhead), HVM4 GPU runtime (resolved July: coding model ported Metal→CUDA overnight, CUDA beats Metal on RTX, ≈10x parallel C), Bend2–SupGen integration [`what-is-bend2.md` , `hvm4-explained.md` , `bend2-vs-bend1.md`].

BEND2: THE COMBINATION (ALL PRE-RELEASE, NONE RUNNABLE)

- Formula: auto-parallel functional code + dependent types + synthesis from precise types; types give SupGen a search target and the checker an acceptance condition; HVM4 schedules independent rewrites, no annotations [`what-is-bend2.md`].
- Syntax (documentation-only samples): Python `def` + return type after `->` ; effects in `do IO<Unit>:` blocks with `<-` binds and `IO::` paths; GPU per call via `!` (`sum!(24, 0)`); theorems as `assert` + same-named `def` proof by case analysis + `%` rewrites + `{=}` [`hello.md` , `parallelism.md` , `gpu.md` , `proofs.md` , `what-is-bend2.md`].
- Lifts two Bend1 limits in samples: `U32` args (24→32 bits), real `IO` (`input` / `print` on CPU; GPU takes pure code only); memory ceiling unknown until release [`bend2-vs-bend1.md` , `hello.md` , `gpu.md`].
- Types from the Kind lineage: linear by default (≤1 live use), `+x` to duplicate, only `Data` kinds duplicable; Dead mode (specs, unrestricted, may diverge) vs Live mode (code+proofs, linear, terminating) contains `Type : Type` paradoxes on the Dead side [`bend2-vs-lean.md` , `HigherOrderCO_Kind-README.md`].
- Proofs are term-level (Agda register): recursion is induction, no tactic layer (`simp` / `omega`) visible in the single `add_comm` sample [`proofs.md` , `bend2-vs-lean.md`].
- Public footprint as of 2026-08-30: empty `VictorTaelin/Bend2` repo (README 404s, zero code — verified), 4 posted samples, no binary/package/license; checker moved to TypeScript-on-Bun (July 2026) [`what-is-bend2.md` , `repos/_meta.txt` , `bend2-vs-bend1.md`].

THE TWO CONTRASTS

- **Mojo** (Modular, Lattner, MLIR, shipped pre-1.0): you schedule — kernels, grids, blocks, device buffers, SIMD; ideal for dense regular numerics where tiling is the performance. Bend2: the evaluator schedules; one character (`!`) picks the device [`bend2-vs-mojo.md` , `gpu.md`].
- **Lean 4** (de Moura, FRO, Apache 2, mathlib 500k+ items): the proof is the product; automation = tactics + libraries. Bend2: the program is the product; automation = SupGen proposing, checker disposing; no public library exists [`bend2-vs-lean.md`].

TIMELINE

- 2026-03-05 launch blockers → 2026-07-18 "Bend2 complete" → 2026-08-12 delay post (≈08-28) → 2026-08-27 plan: copiable-kind fix, weekend testing, Monday recording → possible Wednesday ≈2026-09-02 [`bend2-release-date.md` , `home.md` , `what-is-bend2.md`].

GAPS (EXPLICIT, NOT SILENT)

- `VictorTaelin/hvm4-formal` : README 404s on main+master; content unknown.
- `VictorTaelin/Bend2` : empty; all Bend2 code claims rest on 4 posted samples only.
- HVM4 repo carries no license text (as of 2026-08-04): no grant to build on it [`hvm4-explained.md`].
- Curriculum: only 4 of ~50 `Bend2 by Example` pages written [`learn-index.md`].

H.8 · `bend-docs/openspec/changes/bend-docs-site/research/RETRIEVED.txt`

retrieved: 2026-09-05 Sat Sep 5 18:27:34 UTC 2026

H.9 · `bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-release-date.md`

When does Bend2 release?

Last updated 2026-08-27. Canonical: https://bend2.dev/notes/bend2-release-date/ (https://bend2.dev/notes/bend2-release-date/)

Bend2 is expected around **2026-09-02**. The 2026-08-27 launch plan (https://x.com/VictorTaelin/status/2093045510101676483) puts a possible Wednesday launch after a copiable-kind fix, weekend testing, and a Monday presentation recording; its ≈2026-09-02 estimate replaces the ≈2026-08-28 date derived from the 2026-08-12 delay post.

H.10 · `bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-vs-bend1.md`

Bend2 vs Bend1: what changed

Last updated 2026-08-30. Canonical: https://bend2.dev/notes/bend2-vs-bend1/ (https://bend2.dev/notes/bend2-vs-bend1/)

Bend2 is the unreleased successor to Bend (https://github.com/HigherOrderCO/Bend), called Bend1 here; Bend1 is public and installable. Bend2 keeps the execution model, replaces the runtime generation, adds a type system, and is expected around 2026-09-02.

WHAT BEND1 PROVED

Bend1, launched May 2024 (https://news.ycombinator.com/item?id=40390287), demonstrated that recursion and algebraic data types written in a Python-flavored syntax, with no annotations of any kind, could saturate a GPU. The HVM2 runtime (https://github.com/HigherOrderCO/HVM2) evaluates programs as interaction nets, graphs reduced two adjacent nodes at a time, and rewrites that touch different pairs cannot conflict, so parallelism is a scheduling decision rather than a language feature.

WHERE THE COSTS WERE

Bend1's limits were mostly consequences of one design decision: represent everything as interaction-net nodes compact enough to move through memory fast. Numbers were 24-bit (`u24` , `i24` , `f24`) because the node encoding budgets its bits for tags and ports first, and immediates get what remains. Graph memory was capped in the low gigabytes by fixed-width arena addressing. IO stayed minimal because pure graph reduction has no natural place for effects, and the effect boundary had not been built. Evaluation was strict only: HVM1 had been the lazy, optimal-reduction line, and HVM2 traded that away for a model that mapped onto GPUs. On top of all this sat a large constant factor, so a single tuned CPU core could beat impressive-sounding GPU throughput on many workloads.

WHAT BEND2 KEEPS

Bend2 keeps automatic parallelism on the interaction-net substrate, now four runtime generations in, with HVM4 (https://github.com/HigherOrderCO/HVM4) public and pushed into mid-2026. Bend2 has Metal and CUDA backends. The CUDA port report (https://x.com/VictorTaelin/status/2078471338755232193) records a one-night port by a coding model working against the Metal implementation as executable spec and test oracle. Its benchmark puts CUDA ahead of Metal on RTX and at roughly 10x parallel C for most programs.

WHAT BEND2 ADDS

Bend2 adds dependent types in the tradition of Higher Order Company's Kind (https://github.com/HigherOrderCO/Kind): types that can mention values, so "a sorted list" or "a parser that round-trips with this printer" is a type, and the checker rejects implementations that do not satisfy it. Proofs can replace whole categories of tests, and types precise enough to pin down behavior make generated code checkable. SupGen synthesizes those implementations and appears in the launch-blocker list (https://x.com/VictorTaelin/status/2029567059881857081). The checker moved to TypeScript running under Bun in July 2026. Bun supplies its runtime.

The August 2026 samples lift two of Bend1's three limits. The parallel sum types its arguments `U32`, lifting numbers from 24 bits to 32, and hello world reads input and prints inside a `do` block typed `IO<Unit>`, which is more IO surface than Bend1 ever showed. The memory ceiling stays unknown. The public compiler is not available, so all three remain release-day checks.

WHICH TO USE TODAY

Bend1 is the only Bend that runs as of 2026-08-30, and for workloads that fit inside 24-bit numbers and minimal IO it remains a working demonstration of automatic parallelism: the Bend repository (https://github.com/HigherOrderCO/Bend). Bend2 has no public compiler yet.

H.11 · bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-vs-lean.md

Bend2 vs Lean

Last updated 2026-08-30. Canonical: https://bend2.dev/notes/bend2-vs-lean/ (https://bend2.dev/notes/bend2-vs-lean/)

Bend2 and Lean both have dependent types. Lean is a proof assistant and programming language; Bend2 is a programming language with proofs and massively parallel execution.

WHAT LEAN IS

Lean 4, the mature side of the Bend2 comparison, is a theorem prover and programming language (https://github.com/leanprover/lean4) developed under Leonardo de Moura, stewarded since 2023 by the Lean FRO, Apache 2 licensed, with its own system description (https://lean-lang.org/papers/lean4.pdf). Lean 4 is self-hosted, compiles through C, and runs on a reference-counted runtime with task-level concurrency that the compiler itself exercises. But Lean's center of mass is mathlib (https://github.com/leanprover-community/mathlib4), a single coherent library of formalized mathematics past 500,000 items (https://lean-lang.org/use-cases/mathlib/), built by hundreds of contributors, with tactic automation accumulated over a decade. The Mathlib Initiative's 2025–26 roadmap (https://mathlib-initiative.org/roadmap/) includes monthly dataset publication for AI training. New proofs can import existing definitions and lemmas from mathlib instead of rebuilding them.

HOW BEND2 USES DEPENDENT TYPES

Bend2's dependent types come from the same intellectual tradition by a different road. Higher Order Company's earlier proof language, Kind (https://github.com/HigherOrderCO/Kind), is the visible ancestor. Bend2 combines that type system with the HVM4 (https://github.com/HigherOrderCO/HVM4) interaction-net runtime and a synthesis component, SupGen, named on the 2026-03-05 launch-blocker list (https://x.com/VictorTaelin/status/2029567059881857081). Precise types specify behavior, the checker rejects wrong implementations, SupGen searches for right ones, and evaluation parallelizes without annotation. Bend2 was complete by 2026-07-18 (https://x.com/VictorTaelin/status/2078553927268589989) and remains unreleased as of 2026-08-30, expected around 2026-09-02.

Bend2's type theory uses linearity with no hidden exceptions. A plain binder is at most one live use for any type, duplication is written `+x`, and only values of kind `Data` may be duplicated. The checker splits into a Dead mode, where types and specifications live unrestricted and may diverge, and a Live mode, where code and proofs are linear and terminating. The mode boundary keeps paradox-shaped terms on the Dead side, where nothing is being proved, despite Bend2 using `Type : Type`, which Girard's paradox makes inconsistent as a logic. Lean buys the same guarantee with a universe hierarchy and a termination checker.

SAME TYPES, DIFFERENT JOBS

Bend2 and Lean both live on the propositions-as-types side of the Curry–Howard correspondence: a specification is a type, a proof is a program, checking is type checking. What differs is which side of the correspondence pays the bills. In Lean, the proof is the product. You formalize because the theorem matters, or because the algorithm's correctness matters, and the executable is often incidental. Automation means tactics: decision procedures, simp sets, `omega`, and the libraries that discharge routine obligations. In Bend2, the program is the product. Types exist to make machine-

generated code checkable, so synthesis can propose and the checker can dispose. Automation means SupGen, and if the superposition primitives in HVM4 are its mechanism, they make generation an evaluator-level search over candidate programs rather than a tactic-level search over proof terms.

The proof surface itself became visible in August 2026, when Taelin posted a complete Bend2 proof that addition commutes. A theorem is an `assert` block quantified with `for all`; its proof is a `def` of the same name that matches on constructors, applies lemmas as rewrites with `%`, and closes goals with `{=}`, the recursive call standing in for the induction hypothesis. On that one sample the register is term-level, closer to Agda than to Lean's tactic scripts, with no `simp`, no `omega`, and no visible tactic layer at all; the line-by-line walkthrough tracks the goal through both cases.

EXECUTION

Bend2's runtime story has no counterpart in Lean, and vice versa. Lean compiles to efficient sequential C; concurrency is orchestrated with tasks, not extracted from program semantics, and nobody chooses Lean because their proof needs a GPU. Bend2's entire premise is that evaluation itself is parallel: interaction-net reduction spreads across cores. The CUDA port benchmark report (https://x.com/VictorTaelin/status/2078471338755232193) puts its backend ahead of the Metal reference on RTX.

SIDE BY SIDE

	Lean 4	Bend2
Center of mass	Formalized mathematics, verification	Parallel execution, synthesis
Status 2026-08-30	Mature, FRO-stewarded, self-hosted	Unreleased, expected ≈ 2026-09-02
Library	mathlib, 500k+ items, hundreds of contributors	None public
Automation	Tactics, decision procedures	SupGen synthesis
Runtime	Compiled via C, refcounted, sequential + tasks	HVM4 interaction nets, parallel by default
License	Apache 2	None published
Hiring/community	Large, academic and industrial	Does not exist yet

WHICH TO USE

Between Bend2 and Lean, Lean is the correct default today for anything that needs a proof: verified algorithms, formalized specs, mathematics, AI-assisted theorem proving. The library asymmetry is not close. Mathlib contains more than 500,000 items, while Bend2 has no public library. Bend2 instead joins proofs to programs that run on HVM4 and to SupGen synthesis from precise types.

H.12 · bend-docs/openspec/changes/bend-docs-site/research/sources/bend2-vs-mojo.md

Bend2 vs Mojo

Last updated 2026-08-30. Canonical: https://bend2.dev/notes/bend2-vs-mojo/ (https://bend2.dev/notes/bend2-vs-mojo/)

Bend2 and Mojo pursue high-level GPU code from opposite directions. Mojo gives you explicit control and compiles kernels for NVIDIA, AMD, and Apple targets; Bend2 extracts parallelism from ordinary recursive code with no kernels and is expected around 2026-09-02.

WHAT MOJO IS

Mojo, the shipped side of this Bend2 comparison, is Modular's systems language, designed under Chris Lattner (LLVM, Swift, MLIR) and built on MLIR compilation. The syntax is deliberately close to Python and interoperates with it, but the semantics are systems-grade: value ownership and borrowing in the Rust lineage, `struct` over `class` for performance-critical code, explicit SIMD types, and compile-time metaprogramming. GPU work is explicit: you write kernels in Mojo against a GPU programming model (https://docs.modular.com/mojo/manual/gpu/intro-tutorial/) of grids, blocks, and device memory, and one kernel source compiles to PTX for NVIDIA, AMDGPU IR for ROCm, or Metal for Apple hardware. The repository (https://github.com/modular/modular) carries mixed licensing: the standard library and kernels have been Apache

2 since March 2024 (https://www.modular.com/blog/the-next-big-step-in-mojo-open-source), the compiler remains closed with a stated target of open-sourcing by the end of 2026, and a "Path to Mojo 1.0" was laid out in December 2025. This is a funded company's product with production users; the flagship consumer is Modular's own MAX inference platform.

HOW BEND2 WORKS

Bend2 keeps none of those control surfaces. Programs are ordinary high-level functional code, and the HVM4 runtime (https://github.com/HigherOrderCO/HVM4) evaluates them as interaction nets, where reduction steps are two-node-local and order-independent, so any idle core or warp takes whatever work exists. There is no kernel, no grid, no memory hierarchy in the programmer's model. Bend2 uses `!` as its device mark. `sum!(24, 0)` runs the call on the GPU where `sum(24, 0)` would not, so the device is chosen per call even though the schedule never is. Bend2 was complete by 2026-07-18 (https://x.com/VictorTaelin/status/2078553927268589989) with Metal and CUDA backends. The CUDA port benchmark report (https://x.com/VictorTaelin/status/2078471338755232193) puts the backend ahead of the Metal reference on RTX and at roughly 10x parallel C on most programs.

THE MECHANICAL DIFFERENCE

Bend2 and Mojo assign parallel scheduling to different actors. Mojo's answer is you. You decide what becomes a kernel, how work maps onto blocks, when memory moves, where SIMD applies. That is exactly the control that makes dense, regular, arithmetic-heavy workloads fast, because their performance lives in memory layout and occupancy, and no runtime discovers a tiling for you. Bend2's answer is the evaluator. Recursion over a tree fans out because the branches are disjoint subgraphs, not because anyone scheduled it. Each reduction step moves graph nodes through memory, which creates the constant factor visible in Bend1's May 2024 launch benchmarks (https://news.ycombinator.com/item?id=40390287).

SIDE BY SIDE

Bend2 against Mojo as of 2026-08-30.

	Mojo	Bend2
Origin	Modular Inc., Lattner	Higher Order Company, Taelin
Status 2026-08-30	Shipped, pre-1.0, production users	Unreleased, expected ≈ 2026-09-02
Parallelism	Explicit: kernels, grids, SIMD	Automatic: interaction-net evaluation
Compilation	MLIR → native, PTX, AMDGPU, Metal	HVM4, AOT to C, Metal and CUDA backends
Type system	Static, ownership and borrowing	Dependent types, proofs
Codegen story	AI-assisted kernel authoring	SupGen synthesis from types
License	Stdlib Apache 2, compiler closed until ~end 2026	None published

WHERE EACH WINS

Against Bend2, Mojo wins today anywhere you could name the kernel, including inference serving, image pipelines, dense linear algebra, and HPC stencils. It is a better answer to "I would have written CUDA" for anyone whose codebase is Python, and its numerics are full-width IEEE with explicit SIMD. The parallel sum forks both branches of each recursive call without a kernel. Bend1 shipped 24-bit numbers because of its node encoding; Bend2's posted samples type their integers `U32`, lifting that limit.

H.13 · bend-docs/openspec/changes/bend-docs-site/research/sources/contents.md

Contents

Every public Bend2 page on bend2.dev, with canonical URLs.

BEND2 RELEASE TRACKER

- Is Bend2 released yet? (https://bend2.dev/)

BEND2 NOTES

- What is Bend2? (https://bend2.dev/notes/what-is-bend2/): last updated 2026-08-30
- When does Bend2 release? (https://bend2.dev/notes/bend2-release-date/): last updated 2026-08-27
- Bend2 vs Bend1: what changed (https://bend2.dev/notes/bend2-vs-bend1/): last updated 2026-08-30
- HVM4, explained (https://bend2.dev/notes/hvm4-explained/): last updated 2026-08-21
- Bend2 vs Mojo (https://bend2.dev/notes/bend2-vs-mojo/): last updated 2026-08-30
- Bend2 vs Lean (https://bend2.dev/notes/bend2-vs-lean/): last updated 2026-08-30
- Interaction nets, explained (https://bend2.dev/notes/interaction-nets-explained/): last updated 2026-08-21

BEND2 BY EXAMPLE

- Bend2 by Example (https://bend2.dev/learn/)
- Hello world in Bend2 (https://bend2.dev/learn/hello/): last updated 2026-08-13
- Automatic parallelism in Bend2 (https://bend2.dev/learn/parallelism/): last updated 2026-08-13
- Running on GPUs in Bend2 (https://bend2.dev/learn/gpu/): last updated 2026-08-13
- Proofs in Bend2 (https://bend2.dev/learn/proofs/): last updated 2026-08-21

H.14 · bend-docs/openspec/changes/bend-docs-site/research/sources/gpu.md

Running on GPUs in Bend2

Last updated 2026-08-13. Canonical: https://bend2.dev/learn/gpu/ (https://bend2.dev/learn/gpu/)

Bend2 runs a function on the GPU when the call carries `!`: `sum!(24, 0)` evaluates on the GPU where `sum(24, 0)` would use the CPU. The function here is the parallel sum, unchanged; only `main` differs.

```
import Base

# Sums a range of numbers in parallel.
def sum(d: U32, i: U32) -> U32:
  match d:
    case 0:
      i
    case 1+d:
      a, b = sum(d, i*2), sum(d, i*2+1) # fork
      a + b

# Runs sum on the GPU, via `!`.
def main() -> U32:
  sum!(24, 0)
```

WHAT THE MARK DOES

Bend2's `!` chooses where a call evaluates, not how it parallelizes. The fork inside `sum` is ordinary divide-and-conquer recursion, identical on either device; the GPU gets its thousands of threads pointed at the same graph. Kernels, grids, blocks, and device memory do not appear in the language, which is the contrast with Mojo in one line.

WHAT STAYS ON THE CPU

Bend2 sends pure code through `!` and keeps effects on the host. Hello world runs its input and print in `IO` on the CPU; what goes to the GPU here is arithmetic with no IO in its type.

THE BACKENDS

Bend2's GPU runtimes are Metal and CUDA, built on HVM4; no public toolchain ships them yet. What is Bend2? tracks what exists in public, and this page will get measured numbers when a compiler ships.

H.15 · bend-docs/openspec/changes/bend-docs-site/research/sources/hello.md

Hello world in Bend2

Last updated 2026-08-13. Canonical: https://bend2.dev/learn/hello/ (https://bend2.dev/learn/hello/)

Hello world in Bend2 is a `main` function returning an IO action. The program reads a name from standard input and prints a greeting.

```
import Base

# Performs effects on the CPU.
def main() -> IO<Unit>:
  do IO<Unit>:
    name <- IO::input()
    IO::print("Hello, " ++ name)
```

THE DEFINITION LINE

Bend2 declares functions with Python's `def` plus a return type after the arrow. Type arguments go in angle brackets, so `IO<Unit>` is `IO` applied to `Unit`, a computation run only for its effects. `import Base` brings in the standard library, which supplies `IO`, and `#` starts a comment. `main` does not have to be effectful; a pure program declares `def main() -> U32` and returns a number.

THE DO BLOCK

Bend2 sequences effects in a `do` block that names its type, here `do IO<Unit>:`. Inside the block `<-` runs an effect and binds its result, so `name <- IO::input()` reads a line into `name`, and the final `IO::print(...)` is the block's value. Strings concatenate with `++`, and library functions are reached with `::` paths, as in `IO::input`.

WHERE EFFECTS RUN

Bend2 runs effects on the CPU. GPU evaluation, written by marking a call with `!`, is for pure functions; input and output stay on the host.

H.16 · bend-docs/openspec/changes/bend-docs-site/research/sources/home.md

Is Bend2 released yet?

No. As of 2026-08-27, Bend2 has not been publicly released. Victor Taelin's current estimate is approximately 2026-09-02.

Bend2 is Higher Order Company's massively parallel programming language with dependent types, successor to Bend.

NOTES

- What is Bend2? (https://bend2.dev/notes/what-is-bend2/)
- When does Bend2 release? (https://bend2.dev/notes/bend2-release-date/)
- Bend2 vs Bend1: what changed (https://bend2.dev/notes/bend2-vs-bend1/)
- HVM4, explained (https://bend2.dev/notes/hvm4-explained/)
- Bend2 vs Mojo (https://bend2.dev/notes/bend2-vs-mojo/)
- Bend2 vs Lean (https://bend2.dev/notes/bend2-vs-lean/)
- Interaction nets, explained (https://bend2.dev/notes/interaction-nets-explained/)

BY EXAMPLE

- Bend2 by Example (https://bend2.dev/learn/)

Last updated 2026-08-27. Canonical: https://bend2.dev/ (https://bend2.dev/)

Official repository (https://github.com/VictorTaelin/Bend2).

Unofficial. Not affiliated with Higher Order Company.

H.17 · bend-docs/openspec/changes/bend-docs-site/research/sources/hvm4-explained.md

HVM4, explained

Last updated 2026-08-21. Canonical: https://bend2.dev/notes/hvm4-explained/ (https://bend2.dev/notes/hvm4-explained/)

HVM4 is Higher Order Company's fourth-generation runtime and the substrate Bend2 is built on. It is public on GitHub (https://github.com/HigherOrderCO/HVM4): a C implementation of what its documentation calls the Interaction Calculus (https://github.com/HigherOrderCO/HVM4/blob/main/docs/theory/interaction_calculus.md), created in November 2025 and pushed through mid-2026, with a README that opens "you're here before launch. Use at your own risk."

THE MODEL OF COMPUTATION

HVM4 evaluates a program as a graph and computes by rewriting: each rule consumes one adjacent pair of nodes and writes a bounded replacement in its place, consulting nothing else. Two guarantees follow from that narrowness. A rewrite owns exactly the memory of its two nodes, so rewrites on disjoint pairs proceed with no locks; and the rules are confluent, so every order of rewrites reaches the same result, which lets any idle core take any pending pair. Parallelism stops being a language feature and becomes a scheduling decision. What locality does not buy is a small constant: every step is graph surgery against real memory, and closing that gap is what the runtime generations have been about.

SUPERPOSITIONS

HVM4's calculus has a primitive most runtimes lack: a term can be a superposition of two values. From the README:

```
@main = (&{1, 2} + 10)
//11
//12
```

One expression, two results, because `&{1, 2}` flows both values through the addition, sharing all the work that does not depend on the choice. The dual primitive, duplication, lets one value feed two uses without copying. Superpositions are how an evaluator explores many branches of a search space while paying for the common structure once.

FOUR GENERATIONS

HVM4 closes a lineage that has been public since 2022. HVM1 (https://github.com/HigherOrderCO/HVM1) was the lazy, optimal-reduction evaluator that showed interaction combinators could be fast in practice on CPUs. HVM2 (https://github.com/HigherOrderCO/HVM2) went strict, compiled to C and CUDA, and carried the original Bend at its May 2024 launch (https://news.ycombinator.com/item?id=40390287), along with Bend1's documented limits: 24-bit numbers from the compact node encoding, a memory ceiling from fixed-width arena addressing. HVM3 (https://github.com/HigherOrderCO/HVM3) iterated through early 2026. HVM4 consolidates the line into a strikingly small artifact: the whole runtime builds with one command, `clang -O2 -o src/hvm src/hvm.c`, and ships its memory layout (https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/memory.md) and interaction rules (https://github.com/HigherOrderCO/HVM4/tree/main/docs/hvm/interactions) as documentation next to the code.

WHAT THE AOT COMPILER CHANGES

HVM4's ahead-of-time compiler was one of three items on the 2026-03-05 launch-blocker list (https://x.com/VictorTaelin/status/2029567059881857081), and it targets the lineage's known weakness. An interpreted evaluator pays dispatch overhead on every rewrite: fetch the node, branch on its kind, mutate the graph. Compiling a program's rewrite rules ahead of time to native C removes the dispatch and lets the compiler specialize memory access per rule, which attacks exactly the constant factor that made a tuned CPU core competitive with Bend1's GPU throughput. The second blocker, the GPU runtime, resolved in July: a coding model ported the Metal backend to CUDA in a night (https://x.com/VictorTaelin/status/2078471338755232193), and the port outran its reference on RTX.

WHAT HVM4 MEANS FOR BEND2

HVM4 is the half of Bend2 you can read today. The language, the type checker, and SupGen remain private, but the runtime's calculus, memory layout, and interaction rules are inspectable, and they constrain what the language above them can be: parallel by default, superposition-capable, compiled rather than interpreted. The repository carries no license text as of 2026-08-04, so there is no grant to build on it yet.

H.18 · bend-docs/openspec/changes/bend-docs-site/research/sources/interaction-nets-explained.md

Interaction nets, explained

Last updated 2026-08-21. Canonical: https://bend2.dev/notes/interaction-nets-explained/ (https://bend2.dev/notes/interaction-nets-explained/)

Interaction nets are a model of computation published by Yves Lafont at POPL 1990 (https://dl.acm.org/doi/10.1145/96709.96718): a program is a graph of cells, and every step of execution rewrites exactly two cells that face each other. They are the substrate under HVM4 (https://github.com/HigherOrderCO/HVM4) and therefore under Bend2.

THE PROBLEM THEY SOLVE

Interaction nets answer a scheduling question: how can ten thousand workers rewrite one program without coordinating? Take (λx. x + x) E for an expensive E. Substitute textually and you evaluate E twice; share E as a graph node and every worker that reaches it must agree on who computes it, which in a conventional runtime means a lock, a tag word, or a compare-and-swap, with a garbage collector underneath. The standard designs put coordination in the runtime and tune it. Lafont instead constrained the model itself, making a single step so narrow that no interleaving of steps can go wrong.

The shape came from logic. Girard's linear logic ("Linear logic", Theoretical Computer Science 50, 1987) treats propositions as resources consumed exactly once, and its proofs are nets rather than trees. Lafont distilled the computational core into a freestanding machine model, fourteen pages in the same POPL 1990 proceedings (https://dl.acm.org/doi/10.1145/96709.96718) that carried Lamping's optimal-reduction algorithm.

THE MODEL

An interaction net is a graph of cells. A cell is one constructor or one operation of the program: a number, a +, a list node, a λ. Its symbol says which of these it is, its auxiliary ports are where its arguments and results wire in, and its single principal port, drawn at the apex, is the one place it can react.

<Diagram of="cell" caption="A cell: symbol α, one principal port at the apex, auxiliary ports x_1 … x_n." />

A net is any graph built by wiring ports together in pairs; ports left unwired are the net's interface. Computation happens in exactly one configuration, two cells wired principal port to principal port, an active pair. For each unordered pair of symbols the system declares at most one rule,

$$\alpha(x_1, \ldots, x_m) \bowtie \beta(y_1, \ldots, y_n) \longrightarrow N$$

where N is a fixed net exposing the same free ports $x_1 \ldots x_m$, $y_1 \ldots y_n$. A system over k symbols therefore has at most $k(k+1)/2$ rules, and a firing consumes the two cells and writes N in their place.

<Diagram of="rule" caption="An interaction rule fires only where two principal ports meet, and replaces the pair with a fixed net N on the same free ports." />

A cell has one principal port, so it faces at most one other cell, so it belongs to at most one active pair, so two active pairs can never share a cell. Redexes in an interaction net are disjoint by construction, before any analysis runs, and nothing about a step requires seeing the rest of the graph.

THE TWO THEOREMS

Interaction nets rest on two theorems: one-step confluence, from the 1990 paper (https://dl.acm.org/doi/10.1145/96709.96718), and universality from three symbols, from its 1997 sequel (https://www.sciencedirect.com/science/article/pii/S0890540197926432).

Fire two different active pairs of a net N in either order and the results close in one step.

$$N \longrightarrow N_1,\ N \longrightarrow N_2 \Longrightarrow \exists M\colon N_1 \longrightarrow M \longleftarrow N_2$$

The proof is one sentence. The two active pairs share no cell, so firing one leaves the other intact, and the two orders write the same graph.

<Diagram of="diamond" caption="One-step confluence: firing a then b, or b then a, lands on the same net M." />

Every reduction order reaches the same normal form, so no scheduler can produce a wrong answer. Every reduction to normal form takes the same number of steps, so the work in a program is a property of the program, and a profiler's count means something independent of core count. And any idle worker may fire any active pair it finds, because nothing it does can invalidate a redex elsewhere. This theorem is why hvm4-explained can say parallelism stops being a language feature and becomes a scheduling decision.

The second theorem is about how little suffices. Lafont's interaction combinators (https://www.sciencedirect.com/science/article/pii/S0890540197926432) (Information and Computation 137, 1997) cut the alphabet to three symbols, γ, δ, ε, under six rules, and prove the system universal. Theorem 1 translates any interaction system into it while preserving the structure of the computation, degree of parallelism included. The rules come in two families. A symbol meeting itself annihilates, leaving wires. Distinct symbols commute, each passing through the other and copying it.

<Diagram of="commute" caption="Commutation of δ against γ: each passes through and copies the other, one cell per step, interleaved with every other rewrite in the net." />

That commutation square is what sharing looks like here. A δ does not copy a subgraph in one gulp; it copies one cell per interaction and moves on, so duplication is incremental and interleaves with the computation consuming the copies. Erasure is the same trick run by ε, garbage collection as local rewriting, with no collector thread to pause anything.

THE Λ-CALCULUS ARRIVES, WITH FINE PRINT

Interaction nets run the λ-calculus with β-reduction as a single constant-time rule, and the fine print on duplication is where three decades of research live. Encode abstraction and application as cells and a β-redex is exactly an active pair:

<Diagram of="beta" caption="β-reduction as one interaction. Application meets abstraction, the argument wire becomes the variable, the result wire becomes the body." />

β becomes cheap; duplication carries the real difficulty, because a variable used twice needs a δ, and how much a δ should copy, and when, is the whole question. Lamping's POPL 1990 algorithm (https://dl.acm.org/doi/10.1145/96709.96711) answered it in the strongest sense, with reduction optimal in Lévy's 1978 definition, never contracting two members of the same redex family separately. The fine print arrived in 1998, when Asperti and Mairson proved (https://dl.acm.org/doi/10.1145/268946.268971) that the bookkeeping full optimality requires is not bounded by any elementary function of the number of optimal steps. Optimal counts β-steps; it does not promise the accounting is cheap. The HVM line draws the engineering conclusion. Duplication stays bookkeeping-free, defined by HVM4's interaction calculus (https://github.com/HigherOrderCO/HVM4/blob/main/docs/theory/interaction_calculus.md) as a primitive of its own rather than a transparent implementation of substitution, and Lamping's completeness is traded for constant factors.

IN THE MACHINE

In HVM4 a cell is a tagged machine word, an active pair is an entry in a queue, and an interaction is a handful of loads and stores against a documented memory layout (https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/memory.md). None of it needs a lock, because disjointness lets any core or warp pull pending pairs from a queue while confluence keeps the answer independent of who pulled what. The cost is the constant factor. Every step is graph surgery against real memory, and grinding that factor down is what the runtime generations are for. Numbers meet the same wall. A pure net would encode 1000 as a chain of a thousand cells, so practical runtimes store machine integers inside the node word, where the width left over after ports and tags sets the range; Bend1 had 24 bits left. Duplication surfaces to the programmer as HVM4's superpositions, covered in hvm4-explained. The warrant for building a language on all this is Lafont's translation theorem, which Taelin cited directly in April 2024 (https://x.com/VictorTaelin/status/1779203515098845626). Compilation into combinators preserves the degree of parallelism, so a program lowered onto nets inherits the diamond, and the scheduler inherits its freedom.

WHERE TO READ

Interaction nets have a short canonical shelf. Lafont 1990 (https://dl.acm.org/doi/10.1145/96709.96718), pages 95–108 of the POPL proceedings, has the definition, the type discipline, and the deadlock-freedom argument. Lafont 1997 (https://www.sciencedirect.com/science/article/pii/S0890540197926432) has the combinators, three symbols under six rules, and the universality translation. Lamping 1990 (https://dl.acm.org/doi/10.1145/96709.96711) and Asperti–Mairson 1998 (https://dl.acm.org/doi/10.1145/268946.268971) are the optimality arc, the promise and its price, and Asperti and Guerrini's *The Optimal Implementation of Functional Programming Languages* (Cambridge University Press, 1998) is the book-length account. The HVM4 repository (https://github.com/HigherOrderCO/HVM4) keeps the calculus and the memory layout as documentation next to the code, which is where theory stops being the right word.

H.19 · `bend-docs/openspec/changes/bend-docs-site/research/sources/learn-index.md`

Bend2 by Example

Source mirror of https://bend2.dev/learn/ (https://bend2.dev/learn/) retrieved 2026-09-05. Canonical: https://bend2.dev/learn/ (https://bend2.dev/learn/)

Bend2 by Example is a planned curriculum of small programs. Written pages use public samples and remain marked not runnable until the compiler is public.

BEND2 CURRICULUM

- Hello world (https://bend2.dev/learn/hello/): written; last updated 2026-08-13
- Functions: planned
- Let bindings: planned

- Conditionals: planned
- Booleans: planned
- Numbers: planned
- Strings and characters: planned
- Tuples: planned
- Lists: planned
- Imports: planned
- Algebraic data types: planned
- Records: planned
- Maps: planned
- Pattern matching: planned
- Lambda match: planned
- Recursion: planned
- Mutual recursion: planned
- Higher-order functions: planned
- Closures: planned
- Folds: planned
- The bend construct: planned
- Automatic parallelism `(https://bend2.dev/learn/parallelism/)`: written; last updated 2026-08-13
- Sharing and duplication: planned
- Unscoped lambdas: planned
- Superpositions: planned
- Evaluation order: planned
- Running on GPUs `(https://bend2.dev/learn/gpu/)`: written; last updated 2026-08-13
- Measuring speedup: planned
- Type annotations: planned
- Polymorphism: planned
- Dependent types, first contact: planned
- Sized lists: planned
- Equality: planned
- Proofs `(https://bend2.dev/learn/proofs/)`: written; last updated 2026-08-21
- Induction: planned
- Totality: planned
- Holes: planned
- Types as specifications: planned
- SupGen: planned
- IO: planned
- Files: planned
- Command-line arguments: planned
- Compiling to C: planned
- Compiling to JavaScript: planned
- Compiling to Python: planned

H.20 · `bend-docs/openspec/changes/bend-docs-site/research/sources/parallelism.md`

Automatic parallelism in Bend2

Last updated 2026-08-13. Canonical: https://bend2.dev/learn/parallelism/ `(https://bend2.dev/learn/parallelism/)`

Bend2 parallelizes ordinary recursion: split the work in two, and the runtime evaluates the halves concurrently, with no pragmas, no threads, and no scheduler in the program. The program sums 2^{24} numbers this way.

```
import Base

# Sums a range of numbers in parallel.
def sum(d: U32, i: U32) -> U32:
  match d:
    case 0:
      i
    case 1+d:
      a, b = sum(d, i*2), sum(d, i*2+1) # fork
      a + b

# Runs sum on the GPU, via `!`.
def main() -> U32:
  sum!(24, 0)
```

THE FORK

Bend2 forks where the data flow splits. `a, b = sum(d, i*2), sum(d, i*2+1)` starts both recursive calls, and nothing joins them until `a + b` needs both results. The calls share no state, so on the interaction-net runtime they are disjoint subgraphs, and any idle core can reduce either one without locks; the mechanism is in Interaction nets, explained.

THE MATCH

Bend2 pattern-matches on numbers with a successor case. `case 0:` handles zero, and `case 1+d:` matches any value of at least one, binding `d` to one less than the value matched; the new `d` shadows the argument, and the recursive calls use it. A branch body is an expression, with no `return` anywhere. The arguments are `U32`, 32-bit unsigned integers, where Bend1's node encoding stopped at 24 bits.

WHAT IT COMPUTES

The Bend2 program sums every integer below 2^{24}. Each call owns a block of 2^d consecutive integers and splits it in half, and at depth 0 the block is the single number `i`, so `sum(24, 0)` covers 0 through 16,777,215. The exact sum is 140,737,479,966,720, which does not fit in a `U32`.

RUNNING IT ON THE GPU

Bend2 picks the device separately from the parallelism: `sum!(24, 0)` runs the same call on the GPU, and the fork is what gives the GPU work to spread. The `!` has its own page.

H.21 · bend-docs/openspec/changes/bend-docs-site/research/sources/proofs.md

Proofs in Bend2

Last updated 2026-08-21. Canonical: https://bend2.dev/learn/proofs/ `(https://bend2.dev/learn/proofs/)`

A Bend2 theorem is an `assert` block, and its proof is an ordinary `def` carrying the same name. The program proves that addition commutes.

```
import Base

# Theorem: "for all numbers a and b, a + b equals b + a".
assert add_comm:
  for all a : Nat
  for all b : Nat
  {add(a, b) = add(b, a) : Nat}

# Proof: case analysis, induction, rewrites (`%`).
def add_comm(a, b):
  match a:
    case Z{}:
      %add_zero(b)
      {=}
    case S{p}:
      %add_succ(b, p)
      %add_comm(p, b)
      {=}
```

THE STATEMENT

Bend2 states propositions as types. `for all a : Nat` , spelled as two words, quantifies over the naturals, and `{add(a, b) = add(b, a) : Nat}` is an equality between two `Nat` expressions, braces around the proposition. The `assert` names the obligation; the `def` of the same name discharges it, with parameter types supplied by the assert.

THE PROOF

A Bend2 proof works by case analysis and rewriting. `match a:` splits on the constructors of `Nat` , `Z{}` for zero and `S{p}` for successor, written as brace-suffixed patterns. Each `%` line rewrites the goal with a named equation: `%add_zero(b)` and `%add_succ(b, p)` apply lemmas from `Base` , and `%add_comm(p, b)` , the recursive call, is the induction hypothesis. `{=}` closes a branch once rewriting has made the two sides identical.

FOLLOWING THE GOALS

Bend2's `%` rewrites are easiest to read by tracking each branch's goal. In the zero case the goal is `add(Z, b) = add(b, Z)` ; the left side reduces by definition, `%add_zero(b)` rewrites `add(b, Z)` to `b` , and `{=}` closes `b = b` . In the successor case the goal is `add(S(p), b) = add(b, S(p))` ; the left reduces to `S(add(p, b))` , `%add_succ(b, p)` turns the right into `S(add(b, p))` , and the induction hypothesis makes both sides equal for `{=}` . This is the standard Peano argument: induction on the first argument, two helper lemmas.

NEXT TO LEAN

Bend2 proofs are term-level, in the style of Agda: the function's own recursion is the induction, with no tactic layer like Lean's `simp` or `omega` in sight. Which side of that trade pays off is the subject of Bend2 vs Lean.

H.22 · bend-docs/openspec/changes/bend-docs-site/research/sources/what-is-bend2.md

What is Bend2?

Last updated 2026-08-30. Canonical: https://bend2.dev/notes/what-is-bend2/ (https://bend2.dev/notes/what-is-bend2/)

Bend2 is the in-development successor to Bend (https://github.com/HigherOrderCO/Bend), Higher Order Company's massively parallel programming language. There is no public code as of 2026-08-30; Taelin's 2026-08-27 launch plan (https://x.com/VictorTaelin/status/2093045510101676483) puts the current estimate around 2026-09-02.

WHAT BEND2 COMBINES

Bend2 combines functional code that parallelizes automatically, a dependent type system that proves program properties, and program synthesis from precise types (https://x.com/VictorTaelin/status/2029567059881857081). Automatic parallelism exists in array languages, dependent types in Lean and Higher Order Company's Kind (https://github.com/HigherOrderCO/Kind), and synthesis in research languages. Precise types give SupGen a search target and the checker an acceptance condition; HVM4 (https://github.com/HigherOrderCO/HVM4) schedules independent graph rewrites instead of requiring annotations. The public Bend2 repository (https://github.com/VictorTaelin/Bend2) contains no compiler as of 2026-08-30.

WHAT THE SYNTAX LOOKS LIKE

Bend2 reads as Python with dependent types. Functions are Python-style `def` with typed parameters and a return type after the arrow; effects run in `do` blocks over an explicit `IO` type; a call runs on the GPU when its name carries `!` , as in `sum!(24, 0)` ; and a theorem is an `assert` block whose proof is an ordinary `def` of the same name, written by case analysis and `%` rewrites. The numbers in the samples are typed `U32` , where Bend1's node encoding stopped at 24 bits. Each program is walked through line by line in Bend2 by Example: hello world, the parallel sum, the GPU call, and the commutativity proof.

WHY INTERACTION NETS PARALLELIZE

Bend2's performance model rests on the same substrate as Bend1's: interaction nets. A program is a graph, and evaluation is a small fixed set of rewrite rules, each of which touches exactly two adjacent nodes. Two useful properties fall out of that locality. First, rewrites that share no nodes cannot conflict, so every independent redex in the graph can fire concurrently without locks. Second, the rules are confluent, meaning the final result does not depend on the order in which redexes fire, so a scheduler can hand work to any idle thread, or any idle GPU warp, without coordination. Every step still performs graph surgery with real memory traffic, and that constant factor is where the criticism of Bend1 concentrated (https://news.ycombinator.com/item?id=40390287) at its launch.

THE RUNTIME LINEAGE

Bend2 targets the fourth generation of a runtime line that Higher Order Company has been iterating on since 2022: HVM1 (https://github.com/HigherOrderCO/HVM1), the original lazy evaluator for interaction combinators, then HVM2 (https://github.com/HigherOrderCO/HVM2), the strict rewrite that compiled to C and CUDA and carried Bend at its May 2024 launch (https://news.ycombinator.com/item?id=40390287), then HVM3 (https://github.com/HigherOrderCO/HVM3), and now HVM4 (https://github.com/HigherOrderCO/HVM4), which is public and saw pushes into mid-2026. Bend2 was complete by 2026-07-18 (https://x.com/VictorTaelin/status/2078553927268589989), and the CUDA port report (https://x.com/VictorTaelin/status/2078471338755232193) records an overnight implementation by a coding model (Anthropic's Fable) from the reference Metal runtime, reusing most of the existing code. Its benchmark puts CUDA ahead of Metal on RTX and at roughly 10x parallel C for most programs.

WHAT EXISTS IN PUBLIC

Bend2's public footprint, as of 2026-08-30, is an empty repository plus the four posted samples in What the syntax looks like. `VictorTaelin/Bend2` on GitHub carries the description "Bend2 - WIP", zero bytes of code, and no branches. There is no documentation, binary, package, or license text. Bend (https://github.com/HigherOrderCO/Bend) still receives maintenance, and HVM4 (https://github.com/HigherOrderCO/HVM4) is public. The 2026-03-05 launch-blocker list (https://x.com/VictorTaelin/status/2029567059881857081) named HVM4's AOT compiler, HVM4's GPU runtime, and Bend2-SupGen integration.

BEND2 COMPARED WITH EIGHT LANGUAGES

Language	Execution and memory	Work discovery	Shared-state contract	Static guarantee	GPU boundary
Bend2	Bend2 uses HVM4 graph reduction (https://github.com/HigherOrderCO/HVM4).	**Automatic.** The evaluator schedules independent redexes from ordinary calls.	**No task API.** The published sum uses recursion and a tuple assignment; no locks or shared buffers appear.	**Proof `def`.** A same-named `def` discharges each `assert`.	**One character.** The GPU sample changes `sum` to `sum!`; no grid or buffer code appears.
Bend1	HVM2 graph reduction with Rust, C, and CUDA backends (https://github.com/HigherOrderCO/Bend)	**Automatic.** The evaluator finds independent redexes (https://github.com/HigherOrderCO/Bend).	**Ordinary calls.** Source contains no task API (https://github.com/HigherOrderCO/Bend); the CLI chooses the backend.	No theorem form in the published syntax (https://github.com/HigherOrderCO/Bend).	**Whole program.** `run-cu` or `gen-cu` (https://github.com/HigherOrderCO/Bend#running-bend-programs) selects CUDA.
Lean 4	Native functional code with reference counting (https://lean-lang.org/doc/reference/latest/Run-Time-Code/Reference-Counting/)	**Explicit.** The programmer creates `Task` values (https://lean-lang.org/doc/reference/latest/IO/Tasks-and-Threads/) for the runtime pool.	Immutable values are the default; task creation, observation, and cancellation remain in source.	**Kernel-checked proofs.** Dependent type theory checks proof terms (https://lean-lang.org/doc/reference/latest/).	External packages or foreign code.
Haskell	Lazy graph reduction with garbage collection	**Annotated.** Sparks, strategies, or lightweight threads mark work; the runtime work-steals sparks (https://downloads.haskell.org/ghc/latest/docs/users_guide/using-concurrent.html).	Pure values are the default; `MVar`, STM, and related primitives coordinate shared effects.	GADTs and type families (https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/types.html); no proof kernel.	Libraries or the foreign-function interface; kernels stay separate.

Language	Execution and memory	Work discovery	Shared-state contract	Static guarantee	GPU boundary
OCaml	Eager native or bytecode execution with a generational garbage collector (https://ocaml.org/manual/5.5/parallelism.html#parallel-garbage-collection)	**Explicit.** Domains map one-to-one to OS threads (https://ocaml.org/manual/5.5/parallelism.html); task libraries divide work above them.	The domains share a heap; atomics and locks protect mutable state.	GADTs and modules; no dependent proof kernel.	Libraries or C bindings.
Rust	Native code with ownership and no garbage collector (https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html)	**Explicit.** Threads, async tasks, and data-parallel crates define the work graph.	**Compiler-checked sharing.** `Send` and `Sync` (https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html) reject unsafe crossings.	Ownership, lifetimes, and traits enforce safety properties, not theorems.	Host crates plus a separate kernel toolchain.
Java	JVM bytecode, JIT compilation, and garbage collection	**Explicit.** Platform or virtual threads (https://docs.oracle.com/en/java/javase/26/core/virtual-threads.html), `ForkJoinPool`, streams, and executors define work.	The shared heap uses locks, atomics, concurrent collections, and memory-model rules.	Nominal static types; no theorem language.	Libraries, JNI, or an offload runtime.
Python	CPython bytecode and reference counting; the free-threaded build is optional (https://docs.python.org/3/howto/free-threading-python.html)	**Explicit.** Thread or process executors (https://docs.python.org/3/library/concurrent.futures.html), async tasks, and native libraries define work.	The default GIL serializes Python bytecode; locks or process IPC coordinate work outside it.	Runtime checks and optional static types; no proof kernel.	Array and tensor libraries own kernels and transfers.
Mojo	Native code with value semantics and ownership (https://docs.modular.com/mojo/manual/values/ownership/)	**Explicit.** `parallelize` (https://docs.modular.com/mojo/std/algorithm/backend/cpu/parallelize/parallelize/), SIMD, and GPU threads divide work.	Ownership governs values; the programmer manages device buffers and synchronization.	Traits and compile-time parameters; no theorem kernel.	Explicit kernels, grids, thread blocks, and `DeviceContext` (https://docs.modular.com/mojo/manual/gpu/fundamentals/).

H.23 · `bend-docs/openspec/changes/bend-docs-site/research/repos/_meta.txt`

VictorTaelin/Bend2 :: Bend2 - WIP | stars: 14 | pushed: 2025-06-17T20:28:27Z | license: none VictorTaelin/Interaction-Calculus :: A programming language and model of computation that matches the optimal λ-calculus reduction algorithm perfectly. | stars: 948 | pushed: 2025-11-13T16:25:27Z | license: none VictorTaelin/hvm4-formal :: None | stars: 9 | pushed: 2025-11-06T18:04:43Z | license: none HigherOrderCO/Bend :: A massively parallel, high-level programming language | stars: 19817 | pushed: 2026-07-07T19:51:01Z | license: Apache-2.0 HigherOrderCO/HVM4 :: None | stars: 119 | pushed: 2026-05-30T02:43:04Z | license: none HigherOrderCO/Kind :: A modern proof language | stars: 3765 | pushed: 2025-01-22T16:54:12Z | license: MIT HigherOrderCO/HVM1 :: A massively parallel, optimal functional runtime in Rust | stars: 38 | pushed: 2024-08-07T17:35:16Z | license: MIT HigherOrderCO/HVM2 :: A massively parallel, optimal functional runtime in Rust | stars: 11342 | pushed: 2024-11-21T14:33:04Z | license: Apache-2.0 HigherOrderCO/HVM3 :: HVM3 | stars: 301 | pushed: 2026-01-29T12:35:48Z | license: MIT

H.24 · bend-docs/openspec/changes/bend-docs-site/research/repos/victortaelin-profile.md

Taelin

Hi! I'm a programmer and founder of the HigherOrderCO (https://higherorderco.com). Reach me on:

- Twitter: @victortaelin (https://twitter.com/victortaelin)
- Instagram: @victortaelin (https://instagram.com/victortaelin)
- Reddit: SrPeixinho (https://www.reddit.com/user/SrPeixinho/)
- Email: taelin@higherorderco.com

COMPANY:

- HOC: Towards an Optimal Computer (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b): why interaction paradigm can improve computers.
- HOC: Can we build an optimal processor? (https://x.com/VictorTaelin/status/1806690584670679387): like above, but as a tweet.
- HOC: Complete Historical Overview (https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1): the lore behind HOC's creation.

LEARNING:

- HVM: how? (https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md): an intuitive explanation on how HVM works.
- HVM: Optimal Evaluation in 10 Minutes (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6): same as above, but shorter and more serious.
- HVM: A Parallel Runtime - VictorTaelin (USP) (https://www.youtube.com/watch?v=sDPuQ-UjhVQ): same as above, but as a talk (Eng subs).
- HVM: The mystery behind Optimal Evaluation (https://gist.github.com/VictorTaelin/85b94b5ba5b8b5440ded64bba8c89ac2): why some functions run exponentially faster on HVM.
- HVM: Automatic Deforestation (https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474): β-Optimality gives fusion "for free"
- Kind: Conversion Checker (https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5): notes on Kind's equality algorithm

PROMPTS:

- Mini Termination Checker via Fold Generation (https://gist.github.com/VictorTaelin/676241eb4e290a006e5618f97b8a1c25): finding the legal recursive calls of a clause, without comparing sizes

RESEARCH:

- GPT-4: Increase context via compression (https://gist.github.com/VictorTaelin/d293328f75291b23e203e9d9db9bd136): a meme-ish post that went viral
- HVM: A Parallel Evaluator for Interaction Combinators (https://docs.google.com/viewer?url=https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf): the complete HVM2 specification
- HVM: Optimizing a Prime Sieve (https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8): explores linearity techniques
- HVM: Implementing FFT with λ-terms (https://gist.github.com/VictorTaelin/5776ede998d0039ad1cc9b12fd96811c): explores linearity techniques
- HVM: Implementing a simple SAT solver (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615): explains SUP/DUP nodes
- HVM: Fast Discrete Program Search #1 (https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34): finds the add-carry function given a template
- HVM: Fast Discrete Program Search #2 (https://gist.github.com/VictorTaelin/7fe49a99ebca42e5721aa1a3bb32e278): finds bits functions enumerating a simple DSL
- HVM: Fast Discrete Program Search #3 (https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e): how to enumerate superposed λ-terms
- HVM: Optimal Atomic Linker (with Polarization) (https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b): the optimal linker, featured on HVM3
- HVM: Truly Optimal Evaluation with Unordered Superpositions (https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a): solving HVM O(n^2) case

- HVM: The Collapse Monad in Haskell (https://gist.github.com/VictorTaelin/60d3bc72fb4edefecd42095e44138b41): flattens a collapsed value to a list
- HVM: Optimal Linear Context Passing (https://gist.github.com/VictorTaelin/fb798a5bd182f8c57dd302380f69777a): optimization tricks and pure mutable refs
- Rotating an 1D space with binary trees (https://gist.github.com/VictorTaelin/c672b5bd9e734bae63ac35c010be38ab): just a quick memo of this algorithm
- Theory: Equality on Recursive λ-Terms (https://gist.github.com/VictorTaelin/1af22e2c87f176da0e5ff8cd3430b04f): equality on expressions with fixed points

ANNOUNCEMENTS:

- Bend: Release Day (https://x.com/VictorTaelin/status/1791213162525524076)

COVERAGE:

- Mind-bending new programming language for GPUs just dropped... - Fireship (https://www.youtube.com/watch?v=HCOQmKTFzYY)
- The New Massively Parallel Language - ThePrimeTimeagen (https://www.youtube.com/watch?v=NaytZOiX3fs)
- Bend: a high-level language that runs on GPUs - HackerNews (https://news.ycombinator.com/item?id=40390287)

H.25 · bend-docs/openspec/changes/bend-docs-site/research/repos/taelin-theorem-proving-vs-testing.md

Theorem proving VS testing VS auditing

On this article, I'll explain how theorem proving can be used as an alternative to testing and human auditing on the task of writing correct code.

IMPLEMENTING THE CODE

When trying to find bugs in functions, the most widely adopted methods are auditing, and writing tests. For example, suppose that we wrote an "adder" function, which computes the addition of two bitstrings with a carry bit:

```
data Bit = O | I deriving Show
type Bits = [Bit]

adder :: Bit -> Bits -> Bits -> Bits
adder O []       []       = []
adder O []       (O : ys) = O : adder O [] ys
adder O []       (I : ys) = I : adder O [] ys
adder O (O : xs) []       = O : adder O xs []
adder O (O : xs) (O : ys) = O : adder O xs ys
adder O (O : xs) (I : ys) = I : adder O xs ys
adder O (I : xs) []       = I : adder O xs []
adder O (I : xs) (O : ys) = I : adder O xs ys
adder O (I : xs) (I : ys) = O : adder I xs ys
adder I []       []       = I : []
adder I []       (O : ys) = I : adder O [] ys
adder I []       (I : ys) = O : adder I [] ys
adder I (O : xs) []       = I : adder O xs []
adder I (O : xs) (O : ys) = I : adder O xs ys
adder I (O : xs) (I : ys) = O : adder I xs ys
adder I (I : xs) []       = O : adder I xs []
adder I (I : xs) (O : ys) = I : adder O xs ys
adder I (I : xs) (I : ys) = I : adder I xs ys
```

AUDITING THE CODE

One way to check if that function is correct is to audit it. That is a cool way to say we examine it carefully until we are convinced it is correct. Some do it profissionally, so, hiring them might be a good idea. I've looked it for some time and, while I'm a little bit sleepy, I didn't notice anything suspect, so I'm proud to claim the audit a success. Let's move on.

TESTING THE CODE

Another way to improve our confidence is by actually testing the function. That means we run it and check if it returns what we expect. For example:

```
main :: IO ()
main = print (adder [I,O,O,I] [O,I,I,O]) // prints [I,I,I,I]!
```

This program outputs `1111`, which is the sum of `1001 + 0110`, as expected. Great, but that isn't enough. To improve our rigor, we could create a `test` function, which compares our adder to the result of the native integer addition function:

```
test :: Integer -> Integer -> Bool
test a b = to (adder O (from a) (from b)) == a + b where

  from :: Integer -> Bits
  from = go where
    go 0 = []
    go n | mod n 2 == 0 = O : go (div n 2)
         | otherwise    = I : go (div n 2)

  to :: Bits -> Integer
  to = go 1 where
    go :: Integer -> Bits -> Integer
    go k []        = 0
    go k (O : xs) = go (k * 2)  xs
    go k (I : xs) = k + go (k * 2)  xs
```

We could, then, use it to write a lot of unit tests:

```
main :: IO ()
main = do
  print $ test 3 7
  print $ test 1 5
  print $ test 8 4
  print $ test 6 9
  print $ test 5 5
  print $ test 7 4
  print $ test 3 2
  print $ test 9 12
  print $ test 32 64
  print $ test 99 42
  print $ test 69 69
  print $ test 100 200
  print $ test 321 456
  print $ test 200 1239
  print $ test 256 1024
  print $ test 4096 7654
  print $ test 98765 10546
```

All the tests pass, so, I declare my tests as a success. Let's move on.

PROVING THE CODE CORRECT

Let's try something different now: instead of testing, let's try to **prove** our function correct. Can we? First, let's translate it to Agda:

```
adder : Bit -> Bits -> Bits -> Bits
adder O []       []        = []
adder O []       (O :: ys)  = O :: adder O [] ys
adder O []       (I :: ys)  = I :: adder O [] ys
adder O (O :: xs) []        = O :: adder O xs []
adder O (O :: xs) (O :: ys)  = O :: adder O xs ys
adder O (O :: xs) (I :: ys)  = I :: adder O xs ys
adder O (I :: xs) []        = I :: adder O xs []
adder O (I :: xs) (O :: ys)  = I :: adder O xs ys
adder O (I :: xs) (I :: ys)  = O :: adder I xs ys
adder I []       []        = I :: []
adder I []       (O :: ys)  = I :: adder O [] ys
adder I []       (I :: ys)  = O :: adder I [] ys
adder I (O :: xs) []        = I :: adder O xs []
adder I (O :: xs) (O :: ys)  = I :: adder O xs ys
adder I (O :: xs) (I :: ys)  = O :: adder I xs ys
adder I (I :: xs) []        = O :: adder I xs []
adder I (I :: xs) (O :: ys)  = I :: adder O xs ys
adder I (I :: xs) (I :: ys)  = I :: adder I xs ys
```

Now, let's arbitrarily try to prove that `adder` commutes, as it should. For that, we must write the following function:

```
adder-comm : forall x a b -> adder x a b ≡ adder x b a
```

This is a function that, for every bit `x`, and for all bitstrings `a` and `b`, returns a term of type `adder x a b ≡ adder x b a`. Writing a function of that type is the same as proving `adder` commutes, for reasons out of scope. But wtf is that type? Well, it is a type that, given the rules of Agda, can only be instantiated if the two sides of the equation are actually equal. What? Whatever, let's start our implementation by writing all the cases:

```
adder-comm : forall x a b -> adder x a b ≡ adder x b a
adder-comm O []      []      = ?
adder-comm O []      (O :: b) = ?
adder-comm O []      (I :: b) = ?
adder-comm O (O :: a) []      = ?
adder-comm O (I :: a) []      = ?
adder-comm O (O :: a) (O :: b) = ?
adder-comm O (O :: a) (I :: b) = ?
adder-comm O (I :: a) (O :: b) = ?
adder-comm O (I :: a) (I :: b) = ?
adder-comm I []      []      = ?
adder-comm I []      (O :: b) = ?
adder-comm I []      (I :: b) = ?
adder-comm I (O :: a) []      = ?
adder-comm I (I :: a) []      = ?
adder-comm I (O :: a) (O :: b) = ?
adder-comm I (O :: a) (I :: b) = ?
adder-comm I (I :: a) (O :: b) = ?
adder-comm I (I :: a) (I :: b) = ?
```

We've been implementing functions for our entire lives. Implementing this one can't be that hard, can it? In fact, it is almost complete. The only new thing here is returning a term of type `adder x a b ≡ adder x b a`. But how?

One of the cool things about Agda is that it allows us to ask the type of a hole. If we, thus, ask for the type of any of those `?`s, we should see `adder x a b ≡ adder x b a`, right? After all, that's what I just said we are trying to return! Right? Well, no. If we ask the type of the first one, this is what we see:

```
[] ≡ []
```

But why? That's different from the return type of the function. Because Agda substituded `x`, `a` and `b` by `O`, `[]` and `[]`, which are the concrete values of those variables on the first case. This causes the type to reduce from:

```
adder x a b ≡ adder x a b
```

To:

```
adder O [] [] ≡ adder O [] []
```

To:

```
[] ≡ []
```

Which is simpler. But, again, how the hell do we instantiate a term of type `[] ≡ []`? Well, Agda has this cool function called `refl` which, for any term `x`, returns a term of type `x ≡ x`. Or, in other words: *"any term is equal to itself"*. Can we, thus, just use `refl` there? Yes, we can:

```
adder-comm : forall x a b -> adder x a b ≡ adder x b a
adder-comm O []      []      = refl {x = []}
adder-comm O []      (O :: b) = ?
adder-comm O []      (I :: b) = ?
adder-comm O (O :: a) []      = ?
adder-comm O (I :: a) []      = ?
adder-comm O (O :: a) (O :: b) = ?
adder-comm O (O :: a) (I :: b) = ?
adder-comm O (I :: a) (O :: b) = ?
adder-comm O (I :: a) (I :: b) = ?
adder-comm I []      []      = ?
adder-comm I []      (O :: b) = ?
adder-comm I []      (I :: b) = ?
adder-comm I (O :: a) []      = ?
adder-comm I (I :: a) []      = ?
adder-comm I (O :: a) (O :: b) = ?
adder-comm I (O :: a) (I :: b) = ?
adder-comm I (I :: a) (O :: b) = ?
adder-comm I (I :: a) (I :: b) = ?
```

What? OK, if that didn't make any sense to you, don't worry, it is hard to grasp `refl` on the first try. Not because it is complex, it is just different. The point is, I managed to implement the first case, because:

1. Agda allowed me to, on that case, return a term of type `[] ≡ []`, instead of `adder x a b ≡ adder x b a`, because it substituted the variables by the concrete values of that case.
2. Agda has a magic function called `refl` which, for any term `x`, returns a term of type `x ≡ x`.
3. I called `refl {x = []}`, which gave me a term of type `[] ≡ []`.
4. I returned it, because that is what I needed to return!

In English-math terms, what I did is the equivalent of saying: *"if `x` is `0`, and `a` and `b` are empty, then, `adder x a b` reduces to `[]`, and `adder x b a` also reduces to `[]` ... those are syntactically equal (`refl`), thus, for that case, `adder x a b ≡ adder x b a`".*

Ok, let's move on and check the type of the second hole.

```
0 :: adder 0 [] b ≡ 0 :: adder 0 b []
```

This time, Agda substituted the variables `x`, `a` and `b` by `0`, `[]` and `(0 :: b)`, again their concrete values on that case. But, now, it didn't reduce to a type that is obviously equal. If we write:

```
refl {x = 0 :: adder 0 [] b}
```

We'll have a term of type:

```
0 :: adder 0 [] b ≡ 0 :: adder 0 [] b
```

Which is not what we need to return: notice the position of `b`. The so magical "refl" won't help us this time. What now? Again, we need to build a term of type `0 :: adder 0 [] b ≡ 0 :: adder 0 b []`. Let's try the following:

```
adder-comm : forall x a b -> adder x a b ≡ adder x b a
adder-comm O []      []      = refl {x = []}
adder-comm O []      (O :: b) = let rec = adder-comm O [] b in ?
adder-comm O []      (I :: b) = ?
adder-comm O (O :: a) []      = ?
adder-comm O (I :: a) []      = ?
adder-comm O (O :: a) (O :: b) = ?
adder-comm O (O :: a) (I :: b) = ?
adder-comm O (I :: a) (O :: b) = ?
adder-comm O (I :: a) (I :: b) = ?
adder-comm I []      []      = ?
adder-comm I []      (O :: b) = ?
adder-comm I []      (I :: b) = ?
adder-comm I (O :: a) []      = ?
adder-comm I (I :: a) []      = ?
adder-comm I (O :: a) (O :: b) = ?
adder-comm I (O :: a) (I :: b) = ?
adder-comm I (I :: a) (O :: b) = ?
adder-comm I (I :: a) (I :: b) = ?
```

So, what did happen? We just called `adder-comm` recursively with the arguments `O`, `[]` and `(O :: b)`. That returned a term, `rec`, of type:

```
rec : adder O [] b ≡ adder O b []
```

How? Well, notice the function `adder-comm` (which we are defining) returns a term of type `adder x a b ≡ adder x b a`, so we used it on `O`, `[]` and `b` to get a term of type `adder O [] b ≡ adder O b []`. Can we do that? Yes, we can, in the same way that we can call functions recursively in Haskell. That's called the induction hypothesis.

Now we have a term, `rec`, with a type that is surprisingly close to what we need to return:

```
O :: adder O [] b ≡ O :: adder O b []
```

The only difference is that there is an extra `O ::` on each side. What now? Well, Agda has another nice function called `cong`, with the following type:

```
cong : {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
```

Or, in other words, it allows us to take a term of type `x ≡ y`, and return a term of type `f x ≡ f y`, for any `f`. If we apply `λ x -> O :: x` to both sides of the type of `rec`, then we get a term of type:

```
O :: adder O [] b ≡ O :: adder O b []
```

Which is exactly the type we needed! We can, thus, complete that case by applying `cong` to `rec`. In fact, we can apply the same technique and keep filling all the cases below it:

```
adder-comm : forall x a b -> adder x a b ≡ adder x b a
adder-comm O []      []      = refl {x = []}
adder-comm O []      (O :: b) = cong (λ x → O :: x) (adder-comm O [] b)
adder-comm O []      (I :: b) = cong (λ x → I :: x) (adder-comm O [] b)
adder-comm O (O :: a) []      = cong (λ x → O :: x) (adder-comm O a [])
adder-comm O (I :: a) []      = cong (λ x → I :: x) (adder-comm O a [])
adder-comm O (O :: a) (O :: b) = cong (λ x → O :: x) (adder-comm O a b)
adder-comm O (O :: a) (I :: b) = cong (λ x → I :: x) (adder-comm O a b)
adder-comm O (I :: a) (O :: b) = cong (λ x → I :: x) (adder-comm O a b)
adder-comm O (I :: a) (I :: b) = cong (λ x → O :: x) (adder-comm I a b)
adder-comm I []      []      = refl {x = []}
adder-comm I []      (O :: b) = cong (λ x → I :: x) (adder-comm O [] b)
adder-comm I []      (I :: b) = cong (λ x → O :: x) (adder-comm I [] b)
adder-comm I (O :: a) []      = cong (λ x → I :: x) (adder-comm O a [])
adder-comm I (I :: a) []      = cong (λ x → O :: x) (adder-comm I a [])
adder-comm I (O :: a) (O :: b) = cong (λ x → I :: x) (adder-comm O a b)
adder-comm I (O :: a) (I :: b) = ?
adder-comm I (I :: a) (O :: b) = ?
adder-comm I (I :: a) (I :: b) = ?
```

Cool, isn't it? But there is a problem with the 3rd-last case. If we ask for the type of its hole, we get:

```
0 :: adder I a b ≡ I :: adder 0 b a
```

Here, the induction hypothesis won't help. Even if we could call `adder-comm` recursively to get a term of type `adder I a b ≡ adder 0 b a`, there would be no way to turn it into `0 :: adder I a b ≡ I :: adder 0 b a`. The problem is that each side has a list which starts with a different bit. So, what can we do to complete that clause? Nothing! There is no function in Agda that helps us here, because, guess what? Our function is **not** commutative. We've found a bug! How could that be? Let's check the corresponding clauses:

```
adder I (0 :: xs) (I :: ys)  = 0 :: adder I xs ys
adder I (I :: xs) (0 :: ys)  = I :: adder 0 xs ys
```

There is our mistake: on the `1 + 1 + 0` case, we return `1`, with a carry bit `0`, whereas we should return `0`, with a carry bit `1`! Let's fix it:

```
adder : Bit -> Bits -> Bits -> Bits
adder 0 []       []       = []
adder 0 []       (0 :: ys)  = 0 :: adder 0 [] ys
adder 0 []       (I :: ys)  = I :: adder 0 [] ys
adder 0 (0 :: xs) []       = 0 :: adder 0 xs []
adder 0 (0 :: xs) (0 :: ys)  = 0 :: adder 0 xs ys
adder 0 (0 :: xs) (I :: ys)  = I :: adder 0 xs ys
adder 0 (I :: xs) []       = I :: adder 0 xs []
adder 0 (I :: xs) (0 :: ys)  = I :: adder 0 xs ys
adder 0 (I :: xs) (I :: ys)  = 0 :: adder I xs ys
adder I []       []       = I :: []
adder I []       (0 :: ys)  = I :: adder 0 [] ys
adder I []       (I :: ys)  = 0 :: adder I [] ys
adder I (0 :: xs) []       = I :: adder 0 xs []
adder I (0 :: xs) (0 :: ys)  = I :: adder 0 xs ys
adder I (0 :: xs) (I :: ys)  = 0 :: adder I xs ys
adder I (I :: xs) []       = 0 :: adder I xs []
adder I (I :: xs) (0 :: ys)  = 0 :: adder I xs ys
adder I (I :: xs) (I :: ys)  = I :: adder I xs ys
```

Can we complete `adder-comm` now? Sure:

```
adder-comm : forall x a b -> adder x a b ≡ adder x b a
adder-comm 0 []      []      = refl {x = []}
adder-comm 0 []      (0 :: b) = cong (λ x → 0 :: x) (adder-comm 0 [] b)
adder-comm 0 []      (I :: b) = cong (λ x → I :: x) (adder-comm 0 [] b)
adder-comm 0 (0 :: a) []      = cong (λ x → 0 :: x) (adder-comm 0 a [])
adder-comm 0 (I :: a) []      = cong (λ x → I :: x) (adder-comm 0 a [])
adder-comm 0 (0 :: a) (0 :: b) = cong (λ x → 0 :: x) (adder-comm 0 a b)
adder-comm 0 (0 :: a) (I :: b) = cong (λ x → I :: x) (adder-comm 0 a b)
adder-comm 0 (I :: a) (0 :: b) = cong (λ x → I :: x) (adder-comm 0 a b)
adder-comm 0 (I :: a) (I :: b) = cong (λ x → 0 :: x) (adder-comm I a b)
adder-comm I []      []      = refl {x = []}
adder-comm I []      (0 :: b) = cong (λ x → I :: x) (adder-comm 0 [] b)
adder-comm I []      (I :: b) = cong (λ x → 0 :: x) (adder-comm I [] b)
adder-comm I (0 :: a) []      = cong (λ x → I :: x) (adder-comm 0 a [])
adder-comm I (I :: a) []      = cong (λ x → 0 :: x) (adder-comm I a [])
adder-comm I (0 :: a) (0 :: b) = cong (λ x → I :: x) (adder-comm 0 a b)
adder-comm I (0 :: a) (I :: b) = cong (λ x → 0 :: x) (adder-comm I a b)
adder-comm I (I :: a) (0 :: b) = cong (λ x → 0 :: x) (adder-comm I a b)
adder-comm I (I :: a) (I :: b) = cong (λ x → I :: x) (adder-comm I a b)
```

If we compile the program above, we get no type error, which means we successfully proved `adder` commutes! Yay!

CONCLUSION

By trying to prove a property we expected to hold on `adder`, the compiler naturally guided us to finding a bug; a bug that passed unnoticed through all tests! Did we prove the function correct? No, because... *"define what correct means"*! But we've proven a property which we expect from any correct implementation of `add`. If we wanted a stronger proof, we could implement this function:

```
adder-adds : forall a b -> to (adder O (from a) (from b)) = a + b
```

Which is equivalent to the tests we did comparing `adder` to the built-in `add` function, except this time it'd be a proof rather than just tests. We could, similarly, prove any arbitrary specification of what we consider to be a correct implementation of "add".

RELEVANCE TO SMART-CONTRACTS

Now, you might think: "if we wrote more tests, we'd have spotted that one too!" True, but that's a pretty simple function. Now, think of something much more complex, like:

```
contract-is-undrainable : forall txs -> balance (EVM-apply txs (EMV-deploy bytecode)) > 0
contract-is-undrainable = ...
```

The type of that function says that, no matter what transactions certain contract receives, its balance will never fall to zero. That is a nice property to have in a contract that must be undrainable. You could write as many tests as you want, but you'd never be 100% certain your contract ins undrainable. A proof gives 100% certainty.

That is, IMO, how we will begin trusting smart-contracts that deal with a ton of money. Rather than doing several auditions and writing lots of tests (like we did with TheDAO), we'll, instead, ask for the developers to prove that the contract satisfies certain expectations, which are expressed as specifications (types) that are much easier to read than the code of the contract itself.

(edit)

This is, by the way, a much simpler implementation:

```
inc : {n : Nat} -> Bits n -> Bits n
inc []       = []
inc (O :: xs) = I :: xs
inc (I :: xs) = O :: inc xs

add : {n : Nat} -> Bits n -> Bits n -> Bits n
add []       []       = []
add (O :: xs) (O :: ys) = O :: (add xs ys)
add (O :: xs) (I :: ys) = I :: (add xs ys)
add (I :: xs) (O :: ys) = I :: (add xs ys)
add (I :: xs) (I :: ys) = inc (I :: (add xs ys))

add-comm : {n : Nat} -> (a : Bits n) -> (b : Bits n) -> add a b ≡ add b a
add-comm []       []       = refl
add-comm (O :: xs) (O :: ys) = cong O:: (add-comm xs ys)
add-comm (O :: xs) (I :: ys) = cong I:: (add-comm xs ys)
add-comm (I :: xs) (O :: ys) = cong I:: (add-comm xs ys)
add-comm (I :: xs) (I :: ys) = cong O:: (cong inc (add-comm xs ys))
```

Guess my skills are slighly above they were before on this seemingly infinite stairway.

H.26 · bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_Bend-README.md

Bend

A high-level, massively parallel programming language

INDEX

1. Introduction
2. Important Notes
3. Install
4. Getting Started
5. Speedup Example
6. Additional Resources

INTRODUCTION

Bend offers the feel and features of expressive languages like Python and Haskell. This includes fast object allocations, full support for higher-order functions with closures, unrestricted recursion, and even continuations.
Bend scales like CUDA, it runs on massively parallel hardware like GPUs, with nearly linear acceleration based on core count, and without explicit parallelism annotations: no thread creation, locks, mutexes, or atomics.
Bend is powered by the HVM2 `(https://github.com/higherorderco/hvm)` runtime.

IMPORTANT NOTES

- Bend is designed to excel in scaling performance with cores, supporting over 10000 concurrent threads.
- The current version may have lower single-core performance.
- You can expect substantial improvements in performance as we advance our code generation and optimization techniques.
- We are still working to support Windows. Use WSL2 `(https://learn.microsoft.com/en-us/windows/wsl/install)` as an alternative solution.
- We only support NVIDIA Gpus currently `(https://github.com/HigherOrderCO/Bend/issues/341)`.

INSTALL

INSTALL DEPENDENCIES

ON LINUX

```
# Install Rust if you haven't already.
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# For the C version of Bend, use GCC. We recommend a version up to 12.x.
sudo apt install gcc
```

For the CUDA runtime install the CUDA toolkit for Linux `(https://developer.nvidia.com/cuda-downloads?target_os=Linux)` version 12.x.

ON MAC

```
# Install Rust if you haven't it already.
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# For the C version of Bend, use GCC. We recommend a version up to 12.x.
brew install gcc
```

INSTALL BEND

1. Install HVM2 by running:

```
# HVM2 is HOC's massively parallel Interaction Combinator evaluator.
cargo install hvm

# This ensures HVM is correctly installed and accessible.
hvm --version
```

2. Install Bend by running:

```
# This command will install Bend
cargo install bend-lang

# This ensures Bend is correctly installed and accessible.
bend --version
```

GETTING STARTED

RUNNING BEND PROGRAMS

```
bend run    <file.bend> # uses the C interpreter by default (parallel)
bend run-rs <file.bend> # uses the Rust interpreter (sequential)
bend run-c  <file.bend> # uses the C interpreter (parallel)
bend run-cu <file.bend> # uses the CUDA interpreter (massively parallel)

# Notes
# You can also compile Bend to standalone C/CUDA files using gen-c and gen-cu for maximum performance.
# The code generator is still in its early stages and not as mature as compilers like GCC and GHC.
# You can use the -s flag to have more information on
  # Reductions
  # Time the code took to run
  # Interaction per second (In millions)
```

TESTING BEND PROGRAMS

The example below sums all the numbers in the range from `start` to `target`. It can be written in two different methods: one that is inherently sequential (and thus cannot be parallelized), and another that is easily parallelizable. (We will be using the `-s` flag in most examples, for the sake of visibility)

SEQUENTIAL VERSION:

First, create a file named `sequential_sum.bend`

```
# Write this command on your terminal
touch sequential_sum.bend
```

Then with your text editor, open the file `sequential_sum.bend`, copy the code below and paste in the file.

```
# Defines the function Sum with two parameters: start and target
def Sum(start, target):
  if start == target:
    # If the value of start is the same as target, returns start.
    return start
  else:
    # If start is not equal to target, recursively call Sum with
    # start incremented by 1, and add the result to start.
    return start + Sum(start + 1, target)

def main():
  # This translates to (1 + (2 + (3 + (..... + (999999 + 1000000)))))
  # Note that this will overflow the maximum value of a number in Bend
  return Sum(1, 1_000_000)
```

RUNNING THE FILE

You can run it using Rust interpreter (Sequential)

```
bend run-rs sequential_sum.bend -s
```

Or you can run it using C interpreter (Sequential)

```
bend run-c sequential_sum.bend -s
```

If you have a NVIDIA GPU, you can also run in CUDA (Sequential)

```
bend run-cu sequential_sum.bend -s
```

In this version, the next value to be calculated depends on the previous sum, meaning that it cannot proceed until the current computation is complete. Now, let's look at the easily parallelizable version.

PARALLELIZABLE VERSION:

First close the old file and then proceed to your terminal to create `parallel_sum.bend`

```
# Write this command on your terminal
touch parallel_sum.bend
```

Then with your text editor, open the file `parallel_sum.bend` , copy the code below and paste in the file.

```
# Defines the function Sum with two parameters: start and target
def Sum(start, target):
  if start == target:
    # If the value of start is the same as target, returns start.
    return start
  else:
    # If start is not equal to target, calculate the midpoint (half),
    # then recursively call Sum on both halves.
    half = (start + target) / 2
    left = Sum(start, half)  # (Start -> Half)
    right = Sum(half + 1, target)
    return left + right

# A parallelizable sum of numbers from 1 to 1000000
def main():
  # This translates to (((1 + 2) + (3 + 4)) + ... (999999 + 1000000)...)
  return Sum(1, 1_000_000)
```

In this example, the (3 + 4) sum does not depend on the (1 + 2), meaning that it can run in parallel because both computations can happen at the same time.

RUNNING THE FILE

You can run it using Rust interpreter (Sequential)

```
bend run-rs parallel_sum.bend -s
```

Or you can run it using C interpreter (Parallel)

```
bend run-c parallel_sum.bend -s
```

If you have a NVIDIA GPU, you can also run in CUDA (Massively parallel)

```
bend run-cu parallel_sum.bend -s
```

In Bend, it can be parallelized by just changing the run command. If your code **can** run in parallel it **will** run in parallel.

SPEEDUP EXAMPLES

The code snippet below implements a bitonic sorter `(https://en.wikipedia.org/wiki/Bitonic_sorter)` with *immutable tree rotations*. It's not the type of algorithm you would expect to run fast on GPUs. However, since it uses a divide and conquer approach, which is inherently parallel, Bend will execute it on multiple threads, no thread creation, no explicit lock management.

BITONIC SORTER BENCHMARK

- `bend run-rs` : CPU, Apple M3 Max: 12.15 seconds
- `bend run-c` : CPU, Apple M3 Max: 0.96 seconds
- `bend run-cu` : GPU, NVIDIA RTX 4090: 0.21 seconds

▼ Click here for the Bitonic Sorter code

```
# Sorting Network = just rotate trees!
def sort(d, s, tree):
  switch d:
    case 0:
      return tree
    case _:
      (x,y) = tree
      lft   = sort(d-1, 0, x)
      rgt   = sort(d-1, 1, y)
      return rots(d, s, (lft, rgt))

# Rotates sub-trees (Blue/Green Box)
def rots(d, s, tree):
  switch d:
    case 0:
      return tree
    case _:
      (x,y) = tree
      return down(d, s, warp(d-1, s, x, y))

# Swaps distant values (Red Box)
def warp(d, s, a, b):
  switch d:
    case 0:
      return swap(s ^ (a > b), a, b)
    case _:
      (a.a, a.b) = a
      (b.a, b.b) = b
      (A.a, A.b) = warp(d-1, s, a.a, b.a)
      (B.a, B.b) = warp(d-1, s, a.b, b.b)
      return ((A.a,B.a),(A.b,B.b))

# Propagates downwards
def down(d,s,t):
  switch d:
    case 0:
      return t
    case _:
      (t.a, t.b) = t
      return (rots(d-1, s, t.a), rots(d-1, s, t.b))

# Swaps a single pair
def swap(s, a, b):
  switch s:
    case 0:
      return (a,b)
    case _:
      return (b,a)

# Testing
# -------

# Generates a big tree
def gen(d, x):
  switch d:
    case 0:
      return x
    case _:
      return (gen(d-1, x * 2 + 1), gen(d-1, x * 2))

# Sums a big tree
def sum(d, t):
  switch d:
    case 0:
      return t
    case _:
      (t.a, t.b) = t
      return sum(d-1, t.a) + sum(d-1, t.b)

# Sorts a big tree
def main:
  return sum(20, sort(20, 0, gen(20, 0)))
```

if you are interested in some other algorithms, you can check our examples folder (https://github.com/HigherOrderCO/Bend/tree/main/examples)

ADDITIONAL RESOURCES

- To understand the technology behind Bend, check out the HVM2 paper (https://paper.higherorderco.com/).
- We are working on an official documentation, meanwhile for a more in depth explanation check GUIDE.md (https://github.com/HigherOrderCO/Bend/blob/main/GUIDE.md)
- Read about our features at FEATURES.md (https://github.com/HigherOrderCO/Bend/blob/main/FEATURES.md)
- Bend is developed by HigherOrderCO (https://higherorderco.com/) - join our Discord (https://discord.higherorderco.com)!

H.27 · bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM1-README.md

Higher-order Virtual Machine (HVM) ==========

Higher-order Virtual Machine (HVM) is a pure functional runtime that is **lazy**, **non-garbage-collected** and **massively parallel**. It is also **beta-optimal**, meaning that, for higher-order computations, it can, in some cases, be exponentially (in the asymptotical sense) faster than alternatives, including Haskell's GHC.

That is possible due to a new model of computation, the **Interaction Net**, which supersedes the **Turing Machine** and the **Lambda Calculus**. Previous implementations of this model have been inefficient in practice, however, a recent breakthrough has drastically improved its efficiency, resulting in the HVM. Despite being relatively new, it already beats mature compilers in some cases, and is being continuously improved.

Welcome to the massively parallel future of computers!

Production Ready Soon! ==========

The code here is a prototype, but the first production ready version is coming soon, with tons of optimizations and - most importantly - correctness work. Follow the progress here: HVM-Core (https://github.com/HigherOrderCO/HVM-Core).

Examples ========

Essentially, HVM is a minimalist functional language that is compiled to a novel runtime based on Interaction Nets (https://pdf.sciencedirectassets.com/272575/1-s2.0-S0890540100X00600/1-s2.0-S0890540197926432/main.pdf?X-Amz-Security-Token=IQoJb3JpZ2luX2VjEEkaCXVzLWVhc3QtMSJGMEQCIDQQ1rA8jV9UQEKlaFzLfqINP%2B%2FrcTj4NCbI40n%2FrGbyAiBLG%2FpnaXmp6BGaG1Yplr3YYsHzxet6aQXc1qnkbb3W0irMBAhSEAUaDDA1OTAwMzU0Njg2NSIM1FOMCDHcoyvFFAU6KqkEgJPcH6B8%2BRYsdLtUERtVlwtcXZBW38xnvb%2FPRSkmxcNaK%2BmQTa7L3ZFuZt9rpNjrB3sJHg%2Bxc%2FqdAF%2FsthEb1NreHNze7LmbStuRufZCGEcxax%2FsyjnSb9bnrHuDEpnck1Dhk2YPqR8%2Bim%2BdQisDUp%2F4torZsCK1%2BPAEQkQAmGqinioexAr8dEE0BOlHgxBz5YRIkV9pjLoq%2FjWFqiUSO2bPdVi2AfpDbXI48ek6gQs%2F6VTIFRShfezfAr1HoDlQEoyyVYnVy6wI%2Fu1WVB%2FA0JJHK1B7rZFEYilPSAdUpVSOvjhNHN9elxIxlFX6hOZz3YJ4QDeLCPztfMClYYxAex6hoBBVzTkRzszs18hK1K%2FMUMwF4o%2FDy1i3WLeUmC36CL7WXDik%2BTZ7WjJNYGVRILH6cDsHrg17A0MVI5njvw7iM%2FrYKoOgBD2ESct4nO3mpRkKVq%2F9UyKScwVT5VrNpuLWLnrg29BDvE%2BDoFI6c71cisENjhIhGPNrBCQvZLNe1k%2BD54NyfqOe4a1DguuzxBnsNj6BBD2lM6TyDvCz9w36u194aN8oks9hLuTuKp7Rk05dTt6rj4pThkHA%2FQQymmx74MlQtTXTnD5v%2F%2BmGSUz6vHzqaV2Ft5xjWf9w9NJHfTkFkpxNEv8fTUUSMBEhL4nF8wj0wiNbSwp9NvPOj3YMIG2icNxdAZyNsJYJUowOCXi4JTwCkqb2WdNOi88pOSaAautZrBg7nzCKyuCbBjqqATOzXItndBn%2Be6oyH2l8sD%2B5v%2FjIqCz8%2Bx%2Bz%2FZA3dntddFac64iWFGPbJeRGw05BiPX5TKBnrR%2BmaqfO%2F7SxoYfTV4hl5Z2lmJcoiEd%2BWUmNK2wntMlGtFn%2FmFeeljKBeMxnfh8DN0qRz10NZAfxhvqxAEBu67G0ZXpECGxr8fAiBrdvnEac6rWfv8%2FT0VA%2Fu6xjIMIrrwU65xAuVuIG%2BXpsdC073VLm1%2BEW&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20221119T011901Z&X-Amz-SignedHeaders=host&X-Amz-Expires=300&X-Amz-Credential=ASIAQ3PHCVTYYRK5XVMW%2F20221119%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Signature=74892553e56ba432350974a6f4dbebfd97418e2187a5c4e183da61dd0e951609&hash=bc1de316d0b6ee58191106c1cdbc34d1eaeab536a9bbc02dfae09818a8cc2510&host=68042c943591013ac2b2430a89b270f6af2c76d8dfd086a07176afe7c76c2c61&pii=S0890540197926432&tid=spdf-00500b38-a41c-4d5b-98bb-4a2754da3953&sid=17532fa99b4522476f2b00d636dc838e7e36gxrqa&type=client&ua=515904515402570a0401&rr=76c51d7eea7b4d36). This approach is not only memory-efficient (no GC needed), but also has two significant advantages: **automatic parallelism** and **beta-optimality**. The idea is that you write a simple functional program, and HVM will turn it into a massively parallel, beta-optimal executable. The examples below highlight these advantages in action.

Bubble Sort ----------

From: HVM/examples/sort/bubble/main.hvm	From: HVM/examples/sort/bubble/main.hs

```
// sort : List -> List
(Sort Nil)         = Nil
(Sort (Cons x xs)) = (Insert x (Sort xs))

// Insert : U60 -> List -> List
(Insert v Nil)         = (Cons v Nil)
(Insert v (Cons x xs)) = (SwapGT (> v x) v x xs)

// SwapGT : U60 -> U60 -> U60 -> List -> List
(SwapGT 0 v x xs) = (Cons v (Cons x xs))
(SwapGT 1 v x xs) = (Cons x (Insert v xs))
```

```
sort' :: List -> List
sort' Nil         = Nil
sort' (Cons x xs) = insert x (sort' xs)

insert :: Word64 -> List -> List
insert v Nil         = Cons v Nil
insert v (Cons x xs) = swapGT (if v > x then 1 else 0) v x xs

swapGT :: Word64 -> Word64 -> Word64 -> List -> List
swapGT 0 v x xs = Cons v (Cons x xs)
swapGT 1 v x xs = Cons x (insert v xs)
```

[image not bundled:]

On this example, we run a simple, recursive Bubble Sort (https://en.wikipedia.org/wiki/Bubble_sort) on both HVM and GHC (Haskell's compiler). Notice the algorithms are identical. The chart shows how much time each runtime took to sort a list of given size (the lower, the better). The purple line shows GHC (single-thread), the green lines show HVM (1, 2, 4 and 8 threads). As you can see, both perform similarly, with HVM having a small edge. Sadly, here, its performance doesn't improve with added cores. That's because Bubble Sort is an *inherently sequential* algorithm, so HVM can't improve it.

Radix Sort ----------

From: HVM/examples/sort/radix/main.hvm

```
// Sort : Arr -> Arr
(Sort t) = (ToArr 0 (ToMap t))

// ToMap : Arr -> Map
(ToMap Null)       = Free
(ToMap (Leaf a))   = (Radix a)
(ToMap (Node a b)) =
  (Merge (ToMap a) (ToMap b))

// ToArr : Map -> Arr
(ToArr x Free) = Null
(ToArr x Used) = (Leaf x)
(ToArr x (Both a b)) =
  let a = (ToArr (+ (* x 2) 0) a)
  let b = (ToArr (+ (* x 2) 1) b)
  (Node a b)

// Merge : Map -> Map -> Map
(Merge Free       Free)       = Free
(Merge Free       Used)       = Used
(Merge Used       Free)       = Used
(Merge Used       Used)       = Used
(Merge Free       (Both c d)) = (Both c d)
(Merge (Both a b) Free)       = (Both a b)
(Merge (Both a b) (Both c d)) =
  (Both (Merge a c) (Merge b d))
```

From: HVM/examples/sort/radix/main.hs

```
sort :: Arr -> Arr
sort t = toArr 0 (toMap t)

toMap :: Arr -> Map
toMap Null       = Free
toMap (Leaf a)   = radix a
toMap (Node a b) =
  merge (toMap a) (toMap b)

toArr :: Word64 -> Map -> Arr
toArr x Free       = Null
toArr x Used       = Leaf x
toArr x (Both a b) =
  let a' = toArr (x * 2 + 0) a
      b' = toArr (x * 2 + 1) b
  in Node a' b'

merge :: Map -> Map -> Map
merge Free       Free       = Free
merge Free       Used       = Used
merge Used       Free       = Used
merge Used       Used       = Used
merge Free       (Both c d) = (Both c d)
merge (Both a b) Free       = (Both a b)
merge (Both a b) (Both c d) =
  (Both (merge a c) (merge b d))
```

[image not bundled:]

On this example, we try a Radix Sort (https://en.wikipedia.org/wiki/Radix_sort), based on merging immutable trees. In this test, for now, single-thread performance was superior on GHC - and this is often the case, since GHC is much older and has astronomically more micro-optimizations - yet, since this algorithm is *inherently parallel*, HVM was able to outperform GHC given enough cores. With **8 threads**, HVM sorted a large list **2.5x faster** than GHC.

Keep in mind one could parallelize the Haskell version with `par` annotations, but that would demand time-consuming, expensive refactoring - and, in some cases, it isn't even *possible* to use all the available parallelism with `par` alone. HVM, on the other hands, will automatically distribute parallel workloads through all available cores, achieving horizontal scalability. As HVM matures, the single-thread gap will decrease significantly.

Lambda Multiplication ----------

From: HVM/examples/lambda/multiplication/main.hvm

```
// Increments a Bits by 1
// Inc : Bits -> Bits
(Inc xs) = λex λox λix
  let e = ex
  let o = ix
  let i = λp (ox (Inc p))
  (xs e o i)

// Adds two Bits
// Add : Bits -> Bits -> Bits
(Add xs ys) = (App xs λx(Inc x) ys)

// Multiplies two Bits
// Mul : Bits -> Bits -> Bits
(Mul xs ys) =
  let e = End
  let o = λp (B0 (Mul p ys))
  let i = λp (Add ys (B0 (Mul p ys)))
  (xs e o i)
```

From: HVM/examples/lambda/multiplication/main.hs

```
-- Increments a Bits by 1
inc :: Bits -> Bits
inc xs = Bits $ \ex -> \ox -> \ix ->
  let e = ex
      o = ix
      i = \p -> ox (inc p)
  in get xs e o i

-- Adds two Bits
add :: Bits -> Bits -> Bits
add xs ys = app xs (\x -> inc x) ys

-- Muls two Bits
mul :: Bits -> Bits -> Bits
mul xs ys =
  let e = end
      o = \p -> b0 (mul p ys)
      i = \p -> add ys (b0 (mul p ys))
  in get xs e o i
```

[image not bundled:]

This example implements bitwise multiplication using λ-encodings (https://en.wikipedia.org/wiki/Church_encoding). Its purpose is to show yet another important advantage of HVM: beta-optimality. This chart isn't wrong: HVM multiplies λ-encoded numbers **exponentially faster** than GHC, since it can deal with very higher-order programs with optimal asymptotics, while GHC can not. As esoteric as this technique may look, it can actually be very useful to design efficient functional algorithms. One application, for example, is to implement runtime deforestation (https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474) for immutable datatypes. In general, HVM is capable of applying any fusible function `2^n` times in linear time, which sounds impossible, but is indeed true.

Charts made on plotly.com (https://chart-studio.plotly.com/).

Getting Started ===============

1. Install Rust nightly:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
rustup toolchain install nightly
```

2. Install HVM:

```
cargo +nightly install hvm
```

3. Run an HVM expression:

```
hvm run "(@x(+ x 1) 41)"
```

That's it! For more advanced usage, check the complete guide.

More Information ===============

- To learn more about the **underlying tech**, check guide/HOW.md.
- To ask questions and **join our community**, check our Discord Server (https://discord.higherorderco.com).
- To **contact the author** directly, send an email to taelin@higherorderco.com.

FAQ ===

IS HVM FASTER THAN GHC IN A SINGLE CORE TODAY?

No. For now, HVM seems to be from 50% faster to 3x slower in single thread performance, to even worse if the Haskell code exploits optimizations that HVM doesn't have yet (ST Monad, mutable arrays, inlining, loops).

IS HVM FASTER THAN RUST TODAY?

No.

IS HVM FASTER THAN C TODAY?

No!

CAN HVM BE FASTER THAN THESE ONE DAY?

Hard question. Perhaps! The underlying model is very efficient. HVM shares the same initial core as Rust (an affine λ-calculus), has great memory management (no thunks, no garbage-collection). Some people think interaction nets are an overhead, but that's not the case - they're the *lack* of overhead. For example, a lambda on HVM uses only 2 64-bit pointers, which is about as lightweight as it gets. Furthermore, every reduction rule of HVM is a lightweight, constant-time operation that can be compiled to very fast machine code. As such, given enough optimizations, from proper inlining, to real loops, to inner mutability (FBIP-like?), I believe HVM could one day compare to GHC and even Rust or C. But we're still far from that.

WHY DO THE BENCHMARKS COMPARE SINGLE-THREAD VS MULTI-CORE?

They do not! Notice all benchmarks include a line for single-threaded HVM execution, which is usually 3x slower than GHC. We do include multi-core HVM execution to let us visualize how its performance scales with added cores, without any change of the code. We do not include multi-core GHC execution because GHC doesn't support automatic parallelism, so it is not possible to make use of threads without changing the code. Keep in mind, once again, the benchmarks are NOT claiming that HVM is faster than GHC today.

DOES HVM SUPPORT THE FULL Λ-CALCULUS, OR SYSTEM-F?

Not yet! HVM is an impementation of the bookkeeping-free version of the reduction algorithm proposed on TOIOFPL `(https://www.researchgate.net/publication/235778993_The_optimal_implementation_of_functional_programming_languages)` book, up to page 40. As such, it doesn't support some λ-terms, such as:

```
(λx.(x x) λf.λx.(f (f x)))
```

HVM is, though, Turing complete, so you could implement a full λ-calculus interpreter on it - that limitation only addresses built-in closures. Keep in mind many popular languages don't include the full λ-calculus closures either; Rust, for example, covers a very restricted subset, due to the borrow system. That said, HVM covers a wide class of λ-terms, including the the Y-combinator, church encodings (even algorithms like addition, multiplication and exponentiation), as well as arbitrary datatypes (both native and scott encoded) and recursion.

WILL HVM SUPPORT THE FULL Λ-CALCULUS, OR SYSTEM-F?

Yes! We plan to, by implementing the full algorithm described on the TOIOFPL `(https://www.researchgate.net/publication/235778993_The_optimal_implementation_of_functional_programming_languages)`, i.e., after page 40. Sadly, this results in an overhead that affects the performance of beta-reduction by about 10x. As such, we want to do so with caution to keep HVM efficient. Currently, the plan is:

1. Split lambdas into full-lambdas and light-lambdas
   - Light lambdas are what HVM has today. They're fast, but don't support the full λ-Calculus.
   - Full lambdas will be slower, but support the full λ-Calculus, via "internal brackets/croissants".
2. To decrease the overhead, convert full-lambdas to light-lambdas using EAL inference

   Elementary Affine Logic is a substructural logic that rejects the structural rule of contraction, replacing it by a controlled form of duplication. By extending HVM with EAL inference, we'll be able to convert most full-lambdas into lightweight lambdas, greatly reducing the associated slowdown.

Finally, keep in mind this only concerns lambdas. Low-order terms (constructors, trees, recursion) aren't affected.

ARE UNSUPPORTED TERMS "UNDEFINED BEHAVIOR"?

No! Unsupported λ-terms like `λx.(x x) λf.λx.(f (f x))` don't cause HVM to display undefined behavior. HVM will always behave deterministically, and give you a correct result to any input, except it will be in terms of Interaction Calculus `(https://github.com/HigherOrderCO/Kindex/blob/master/Apps/IC/_.kind2)` (IC) semantics. The IC is an alternative to the Lambda Calculus (LC) which differs slightly in how non-linear variables are treated. As such, these "unsupported" terms are just cases where the LC and the IC evaluation disagree. In theory, you could use the HVM as a Interaction Net runtime, and it would always give you perfectly correct answers under these semantics - but that's not usual, so we don't talk about it often.

WHAT IS HVM'S MAIN INNOVATION, IN SIMPLE TERMS?

In complex terms, HVM's main innovation is that it is an efficient implementation of the Interaction Net, which is a concurrent model of computation. But there is a way to translate it to more familiar terms. HVM's performance, parallelism and GC-freedom all come from the fact it is based on a linear core - just like Rust! But, on top of it, instead of adding loops and references (plus a "borrow checker"), HVM adds recursion and a *lazy, incremental cloning primitive*. For example, the expression below:

```
let xs = (Cons 1 (Cons 2 (Cons 3 Nil))) in [xs, xs]
```

Computes to:

```
let xs = (Cons 2 (Cons 3 Nil)) in [(Cons 1 xs), (Cons 1 xs)]
```

Notice the first `Cons 1` layer was cloned incrementally. This makes cloning essentially free, for the same reason Haskell's lazy evaluator allows you to make infinite lists: there is no cost until you actually read the copy! That lazy-cloning primitive is pervasive, and covers all primitives of HVM's runtime: constructors, numbers and lambdas. This idea, though, breaks down for lambdas: how do you incrementally copy a lambda?

```
let f = λx. (2 + x) in [f, f]
```

If you try it, you'll realize why that's not possible:

```
let f = (2 + x) in [λx. f, λx. f]
```

The solution to that question is the main insight that the Interaction Net model brought to the table, and it is described in more details on the HOW.md `(https://github.com/HigherOrderCO/HVM/blob/master/guide/HOW.md)` document.

IS HVM ALWAYS *ASYMPTOTICALLY* FASTER THAN GHC?

No. In most common cases, it will have the same asymptotics. In some cases, it is exponentially faster. In this issue `(https://github.com/HigherOrderCO/HVM/issues/60)`, a user noticed that HVM displays quadratic asymptotics for certain functions that GHC computes in linear time. That was a surprise to me, and, as far as I can tell, despite the "optimal" brand, seems to be a limitation of the underlying theory. That said, there are multiple ways to alleviate, or solve, this problem. One approach would be to implement "safe pointers", also described on the book, which would reduce the cloning overhead and make some quadratic cases linear. But that wouldn't work for all cases. A complimentary approach would be to do linearity analysis, converting problematic quadratic programs in faster, linear versions. Finally, in the worst case, we could add references just like Haskell, but that should be made with a lot of caution, in order not to break the assumptions made by the parallel execution engine. For a more in depth explanation, check read comment on Hacker News `(https://news.ycombinator.com/edit?id=35342297)`.

IS HVM'S OPTIMALITY ONLY RELEVANT FOR WEIRD, ACADEMIC Λ-ENCODED TERMS?

No. HVM's optimality has some very practical benefits. For example, all the "deforesting" techniques that Haskell employs as compile-time rewrite rules, happen naturally, at runtime, on the HVM. For example, Haskell optimizes:

```
map f . map g
```

Into:

```
map (f . g)
```

This is a hardcoded optimization. On HVM, that occurs naturally, at runtime, in a very general and pervasive way. So, for example, if you have something like:

```
foldr (.) id funcs :: [Int -> Int]
```

GHC won't be able to "fuse" the functions on the `funcs` list, since they're not known at compile time. HVM will do that just fine. See this issue `(https://github.com/HigherOrderCO/HVM/issues/167)` for a practical example.

Another practical application for λ-encodings is for monads. On Haskell, the Free Monad library uses Church encodings as an important optimization. Without it, the asymptotics of binding make free monads much less practical. HVM has optimal asymptotics for Church encoded data, making it great for these problems.

WHY IS HVM SO PARALLELIZABLE?

Because it is fully linear: every piece of data only occurs in one place at the same time, which reduces need for synchronization. Furthermore, it is pure, so there are no global side effects that demand communication. Because of that, reducing HVM expressions in parallel is actually quite simple: we just keep a work strealing queue of redexes, and let a pool of threads computing them. That said, there are two places where HVM needs synchronization:

- On dup nodes, used by lazy cloning: a lock is needed to prevent threads from passing through, and, thus, accessing the same data
- On the substitution operation: that's because substitution could send data from one thread to another, so it must be done atomically

In theory, Haskell could be parallelized too, and GHC devs tried it at a point, but I believe the non-linearity of the STG model would make the problem much more complex than it is for the HVM, making it hard to not lose too much performance due to synchronization overhead.

HOW IS THE MEMORY FOOTPRINT OF HVM, COMPARED TO OTHER RUNTIMES?

It is a common misconception that an "interactional" runtime would somehow consume more memory than a "procedural" runtime like Rust's. That's not the case. Interaction nets, as implemented on HVM, add no overhead, and HVM instantly collects any piece of data that becomes unreachable, just like Rust, so there are no accumulating thunks that result in world-stopping garbage collection, as happens in Haskell currently.

That said, currently, HVM doesn't implement memory-efficient features like references, loops and local mutability. As such, to do anything on HVM today, you need to use immutable datatypes and recursion, which are naturally memory-hungry. Thus, HVM programs today will have increased memory footprint, in relation to C and Rust programs. Thankfully, there is no theoretical limitation preventing us from adding loops and local mutability, and, once/if we do, one can expect the same memory footprint as Rust. The only caveat, though, is shared references: we're not sure if we want to add these, as they might impact parallelism. As such, it is possible that we choose to let lazy clones to be the only form of non-linearity, which would preserve parallelism, at the cost of making some algorithms more memory-hungry.

IS HVM MEANT TO REPLACE GHC?

No! GHC is actually a superb, glorious runtime that is very hard to match. HVM is meant to be a lightweight, massively parallel runtime for functional, and even imperative, languages, from Elm to JavaScript. That said, we do want to support Haskell, but that will require HVM being in a much later stage of maturity, as well as provide support for full lambdas, which it doesn't do yet. Once we do, HVM could be a great alternative for GHC, giving the Haskell community an option to run it in a runtime with automatic parallelism, no slow garbage-collector and beta-optimality. Which will be the best option will likely depend on the type of application you're compiling, but having more choices is generally good and, as such, HVM can be a great tool for the Haskell community.

IS HVM PRODUCTION-READY?

No. HVM is still to be considered a prototype. Right now, I had less than 3 months to work on it directly. It is considerably less mature than other compilers and runtimes like GHC and V8. That said, we're raising funds to have a proper team of engineers working on the HVM. If all goes well, we can expect a production-ready release by Q1 2024.

I'VE RAN AN HVM PROGRAM AND IT CONSUMED 1950 GB AND MY COMPUTER EXPLODED.

HVM is a prototype. Bugs are expected. Please, open an issue!

I'VE USED HVM IN PRODUCTION AND NOW MY COMPANY IS BANKRUPT.

I quit.

Related Work ============

- Inpla `(https://github.com/inpla/inpla)` - a pure interaction net framework, without the "functional/calculus" style of HVM
- HINet `(http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/)` - implementation of interaction nets in Haskell

H.28 · bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM2-README.md

Higher-order Virtual Machine 2 (HVM2) ====================================

Higher-order Virtual Machine 2 (HVM2) is a massively parallel Interaction Combinator (https://www.semanticscholar.org/paper/Interaction-Combinators-Lafont/6cfe09aa6e5da6ce98077b7a048cb1badd78cc76) evaluator.

By compiling programs from high-level languages (such as Python and Haskell) to HVM, one can run these languages directly on massively parallel hardware, like GPUs, with near-ideal speedup.

HVM2 is the successor to HVM1 (https://github.com/HigherOrderCO/HVM1), a 2022 prototype of this concept. Compared to its predecessor, HVM2 is simpler, faster and, most importantly, more correct. HOC (https://HigherOrderCO.com/) provides long-term support for all features listed on its PAPER.

This repository provides a low-level IR language for specifying the HVM2 nets and a compiler from that language to C and CUDA. It is not meant for direct human usage. If you're looking for a high-level language to interface with HVM2, check Bend (https://github.com/HigherOrderCO/Bend) instead.

Usage -----

> DISCLAIMER: Windows is currently not supported, please use WSL (https://learn.microsoft.com/en-us/windows/wsl/install) for now as a workaround.

First install the dependencies:

- If you want to use the C runtime, install a C-11 compatible compiler like GCC or Clang.
- If you want to use the CUDA runtime, install CUDA and nvcc (the CUDA compiler).
  - *HVM requires CUDA 12.x and currently only works on Nvidia GPUs.*

Install HVM2:

```
cargo install hvm
```

There are multiple ways to run an HVM program:

```
hvm run    <file.hvm> # interpret via Rust
hvm run-c  <file.hvm> # interpret via C
hvm run-cu <file.hvm> # interpret via CUDA
hvm gen-c  <file.hvm> # compile to standalone C
hvm gen-cu <file.hvm> # compile to standalone CUDA
```

All modes produce the same output. The compiled modes require you to compile the generated file (with `gcc file.c -o file` , for example), but are faster to run. The CUDA versions have much higher peak performance but are less stable. As a rule of thumb, `gen-c` should be used in production.

Language --------

HVM is a low-level compile target for high-level languages. It provides a raw syntax for wiring interaction nets. For example:

```
@main = a
  & @sum ~ (28 (0 a))

@sum = (?(((a a) @sum__C0) b) b)

@sum__C0 = ({c a} ({$([*2] $([+1] d)) $([*2] $([+0] b))} f))
  &! @sum ~ (a (b $([+] $(e f))))
  &! @sum ~ (c (d e))
```

The file above implements a recursive sum. If that looks unreadable to you - don't worry, it isn't meant to. Bend (https://github.com/HigherOrderCO/Bend) is the human-readable language and should be used both by end users and by languages aiming to target the HVM. If you're looking to learn more about the core syntax and tech, though, please check the PAPER.

H.29 · bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM3-README.md

HVM3 - Work In Progress

The **HVM** is an efficient implementation of the Interaction Calculus `(https://github.com/VictorTaelin/Interaction-Calculus)` (IC).

The Interaction Calculus is a new foundation for computing, similar to the Lambda Calculus, but theoretically optimal. The HVM is an efficient implementation of this new paradigm, and can be seen as a fast engine for symbolic computations.

In some ways, it is very similar to Haskell, but it has some key differences:

- Lambdas must be linear or affine, making it resource-aware
- Lambdas have no scope boundaries, enabling global substitutions
- First-class duplications allow a term to be copied into two locations
- First-class superpositions allow 2 terms to be stored in 1 location

These primitives allow HVM to natively represent concepts that are not present in the traditional λ-Calculus, including continuations, linear HOAS interpreters and mutable references. Moreover, superpositions and duplications allow it to perform optimal beta-reduction, allowing some expressions to be evaluated with an exponential speedup. Finally, being fully affine makes its garbage collector very efficient, and greatly simplifies parallelism.

The HVM3 is the successor to HVM1 and HVM2, combining their strengths. It aims to be the main compile target of Bend `(https://github.com/HigherOrderCO/Bend)`. It is a WIP under active development.

SPECIFICATIONS

- IC.md: Interaction Calculus, the theoretical foundation behind HVM
- HVM.md: the HVM language, which extends IC with pragmatic primitives

INSTALL

1. Install Cabal.
2. Clone this repository.
3. Run `cabal install .`

USAGE

```
hvm run file.hvml    # runs interpreted (slow)
hvm run file.hvml -c # runs compiled (fast)
```

Note: the `-c` flag will also generate a standalone `.main.c` file, which if you want, you can compile and run it independently. See examples in the book/ directory.

H.30 · bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_HVM4-README.md

HVM

HVM is a high-performance runtime for the Interaction Calculus.

NOTE: you're here before launch. Use at your own risk.

BUILDING AND RUNNING

```
# Build
clang -O2 -o src/hvm src/hvm.c

# Run a file (use collapse mode by default)
hvm devs/test/file.hvm -s -C10

# Run all tests
./devs/test/_all_.sh

# Run a benchmark file
hvm devs/bench/u32_fib.hvm -s
```

Flags:

- `-s` shows performance stats
- `-D` prints each intermediate reduction step with interaction labels
- `-C10` collapses and flattens superpositions (limit to 10 lines)

EXAMPLES

```
@main = ((@add 1) 2)
//3
```

```
@main = (&{1, 2} + 10)
//11
//12
```

```
@main = (! x &A= 3; (x₀ + x₁))
//6
```

DOCUMENTATION

- Theory: docs/theory/interaction_calculus.md
- Core language: docs/hvm/core.md
- Memory layout: docs/hvm/memory.md
- Interaction rules: docs/hvm/interactions/

H.31 · bend-docs/openspec/changes/bend-docs-site/research/repos/HigherOrderCO_Kind-README.md

Kind-Core

Kind is a very raw and minimal Type Theory.

Usage

1. Clone and install this project
2. Use the `kind` command to check/run terms

GRAMMAR:

```
<Name> ::=
  <alphanumeric-string>

<Numb> ::=
  <json-number-literal>

<Term> ::=
  | ALL: "∀(" <Name> ":" <Term> ")" <Term>
  | LAM: "λ" <Name> <Term>
  | APP: "(" <Term> <Term> ")"
  | ANN: "{" <Name> ":" <Term> "}"
  | SLF: "$(" <Name> ":" <Term> ")" <Term>
  | INS: "~" <Term>
  | DAT: "#[" <Term>* "]" "{" (<Ctor>)* "}"
  | CON: "#" <Name> "{" <Term>* "}"
  | SWI: "λ{0:" <Term> "_:" <Term> "}"
  | MAT: "λ{" ("#" <Name> ":" <Term>)* "}"
  | REF: <Name>
  | LET: "let" <Name> "=" <Term> <Term>
  | SET: "*"
  | NUM: <Numb>
  | OP2: "(" <Oper> <Term> <Term> ")"
  | TXT: '"' <string-literal> '"'
  | HOL: "?" <Name> ("[" <Term> ("," <Term>)* "]")?
  | MET: "_" <Numb>

<Ctor> ::=
  | "#" <Name> <Tele>

<Tele> ::=
  | "{" (<Name> ":" <Term>)* "}" ":" <Term>

<Oper> ::=
  | "+" | "-"  | "*"  | "/"
  | "%" | "<=" | ">=" | "<"
  | ">" | "==" | "!=" | "&"
  | "|" | "^"  | "<<" | ">>"
```

H.32 · bend-docs/openspec/changes/bend-docs-site/research/repos/VictorTaelin_Bend2-README.md

404: Not Found

H.33 · bend-docs/openspec/changes/bend-docs-site/research/repos/VictorTaelin_Interaction-Calculus-README.md

Interaction Calculus

The Interaction Calculus is a minimal term rewriting system inspired by the Lambda Calculus (λC), but with some key differences that make it inherently more efficient, in a way that closely resembles Lamping's optimal λ-calculus evaluator, and more expressive, in some ways. In particular:

1. Vars are affine: they can only occur up to one time.
2. Vars are global: they can occur anywhere in the program.
3. It features first-class *superpositions* and *duplications*.

Global lambdas allow the IC to express concepts that aren't possible on the traditional λC, including continuations, linear HOAS, and mutable references. Superpositions and duplications allow the IC to be optimally evaluated, making some computations exponentially faster. Finally, being fully affine makes its garbage collector very efficient, and greatly simplifies parallelism.

The HVM `(https://github.com/HigherOrderCO/HVM3)` is a fast, fully featured implementation of this calculus.

This repo now includes a reference implementation in C, which is also quite fast!

Now it also includes a single-file implementation in Haskell, great for learning!

USAGE

This repository includes a reference implementation of the Interaction Calculus in plain C, with some additional features, like native numbers. To install it:

```
make clean
make
```

Then, run one of the examples:

```
./bin/ic run examples/test_0.ic
```

For learning, edit the Haskell file: it is simpler, and has a step debugger.

SPECIFICATION

An IC term is defined by the following grammar:

```
Term ::=
  | VAR: Name
  | ERA: "*"
  | LAM: "λ" Name "." Term
  | APP: "(" Term " " Term ")"
  | SUP: "&" Label "{" Term "," Term "}"
  | DUP: "!" "&" Label "{" Name "," Name "}" "=" Term ";" Term
```

Where:

- VAR represents a variable.
- ERA represents an erasure.
- LAM represents a lambda.
- APP represents a application.
- SUP represents a superposition.
- DUP represents a duplication.

Lambdas are curried, and work like their λC counterpart, except with a relaxed scope, and with affine usage. Applications eliminate lambdas, like in λC, through the beta-reduce (APP-LAM) interaction.

Superpositions work like pairs. Duplications eliminate superpositions through the DUP-SUP interaction, which works exactly like a pair projection.

What makes SUPs and DUPs unique is how they interact with LAMs and APPs. When a SUP is applied to an argument, it reduces through the APP-SUP interaction, and when a LAM is projected, it reduces through the DUP-LAM interaction. This gives a computational behavior for every possible interaction: there are no runtime errors on the Interaction Calculus.

The 'Label' is just a numeric value. It affects the DUP-SUP interaction.

The core interaction rules are listed below:

```
(* a)
----- APP-ERA
*

(λx.f a)
-------- APP-LAM
x <- a
f

(&L{a,b} c)
----------------- APP-SUP
! &L{c0,c1} = c;
&L{(a c0),(b c1)}

! &L{r,s} = *;
K
-------------- DUP-ERA
r <- *
s <- *
K

! &L{r,s} = λx.f;
K
----------------- DUP-LAM
r <- λx0.f0
s <- λx1.f1
x <- &L{x0,x1}
! &L{f0,f1} = f;
K

! &L{x,y} = &L{a,b};
K
-------------------- DUP-SUP (if equal labels)
x <- a
y <- b
K

! &L{x,y} = &R{a,b};
K
-------------------- DUP-SUP (if different labels)
x <- &R{a0,b0}
y <- &R{a1,b1}
! &L{a0,a1} = a;
! &L{b0,b1} = b;
K
```

Where `x <- t` stands for a global substitution of `x` by `t`.

Since variables are affine, substitutions can be implemented efficiently by just inserting an entry in a global substitution map (`sub[var] = value`). There is no need to traverse the target term, or to handle name capture, as long as fresh variable names are globally unique. It can also be implemented in a concurrent setup with a single atomic-swap.

Below is a pseudocode implementation of these interaction rules:

```
def app_lam(app, lam):
  sub[lam.nam] = app.arg
  return lam.bod

def app_sup(app, sup):
  x0 = fresh()
  x1 = fresh()
  a0 = App(sup.lft, Var(x0))
  a1 = App(sup.rgt, Var(x1))
  return Dup(sup.lab, x0, x1, app.arg, Sup(a0, a1))

def dup_lam(dup, lam):
  x0 = fresh()
  x1 = fresh()
  f0 = fresh()
  f1 = fresh()
  sub[dup.lft] = Lam(x0, Var(f0))
  sub[dup.rgt] = Lam(x1, Var(f1))
  sub[lam.nam] = Sup(dup.lab, Var(x0), Var(x1))
  return Dup(dup.lab, f0, f1, lam.bod, dup.bod)

def dup_sup(dup, sup):
  if dup.lab == sup.lab:
    sub[dup.lft] = sup.lft
    sub[dup.rgt] = sup.rgt
    return dup.bod
  else:
    a0 = fresh()
    a1 = fresh()
    b0 = fresh()
    b1 = fresh()
    sub[dup.lft] = Sup(sup.lab, Var(a0), Var(b0))
    sub[dup.rgt] = Sup(sup.lab, Var(a1), Var(b1))
    return Dup(dup.lab, a0, a1, sup.lft, Dup(dup.lab, b0, b1, sup.rgt, dup.bod))
```

Terms can be reduced to weak head normal form, which means reducing until the outermost constructor is a value (LAM, SUP, etc.), or until no more reductions are possible. Example:

```
def whnf(term):
  while True:
    match term:
      case Var(nam):
        if nam in sub:
          term = sub[nam]
        else:
          return term
      case App(fun, arg):
        fun = whnf(fun)
        match fun.tag:
          case Lam: term = app_lam(term, fun)
          case Sup: term = app_sup(term, fun)
          case _  : return App(fun, arg)
      case Dup(lft, rgt, val, bod):
        val = whnf(val)
        match val.tag:
          case Lam: term = dup_lam(term, val)
          case Sup: term = dup_sup(term, val)
          case _  : return Dup(lft, rgt, val, bod)
      case _:
        return term
```

Terms can be reduced to full normal form by recursively taking the whnf:

```
def normal(term):
  term = whnf(term)
  match term:
    case Lam(nam, bod):
      bod_nf = normal(bod)
      return Lam(nam, bod_nf)
    case App(fun, arg):
      fun_nf = normal(fun)
      arg_nf = normal(arg)
      return App(fun_nf, arg_nf)
    ...
    case _:
      return term
```

Below are some normalization examples.

Example 0: (simple λ-term)

```
(λx.λt.(t x) λy.y)
------------------ APP-LAM
λt.(t λy.y)
```

Example 1: (larger λ-term)

```
(λb.λt.λf.((b f) t) λT.λF.T)
---------------------------- APP-LAM
λt.λf.((λT.λF.T f) t)
---------------------- APP-LAM
λt.λf.(λF.t f)
-------------- APP-LAM
λt.λf.t
```

Example 2: (global scopes)

```
{x,(λx.λy.y λk.k)}
------------------ APP-LAM
{λk.k,λy.y}
```

Example 3: (superposition)

```
!{a,b} = {λx.x,λy.y}; (a b)
--------------------------- DUP-SUP
(λx.x λy.y)
----------- APP-LAM
λy.y
```

Example 4: (overlap)

```
({λx.x,λy.y} λz.z)
------------------ APP-SUP
! {x0,x1} = λz.z; {(λx.x x0),(λy.y x1)}
--------------------------------------- DUP-LAM
! {f0,f1} = {r,s}; {(λx.x λr.f0),(λy.y λs.f1)}
---------------------------------------------- DUP-SUP
{(λx.x λr.r),(λy.y λs.s)}
------------------------ APP-LAM
{λr.r,(λy.y λs.s)}
------------------ APP-LAM
{λr.r,λs.s}
```

Example 5: (default test term)

The following term can be used to test all interactions:

```
((λf.λx.!{f0,f1}=f;(f0 (f1 x)) λB.λT.λF.((B F) T)) λa.λb.a)
---------------------------------------------------------- 16 interactions
λa.λb.a
```

COLLAPSING

An Interaction Calculus term can be collapsed to a superposed tree of pure Lambda Calculus terms without SUPs and DUPs, by extending the evaluator with the following collapse interactions:

```
λx.*
------ ERA-LAM
x <- *
*

(f *)
----- ERA-APP
*

λx.&L{f0,f1}
----------------- SUP-LAM
x <- &L{x0,x1}
&L{λx0.f0,λx1.f1}

(f &L{x0,x1})
------------------ SUP-APP
!&L{f0,f1} = f;
&L{(f0 x0),(f1 x1)}

!&L{x0,x1} = x; K
----------------- DUP-VAR
x0 <- x
x1 <- x
K

!&L{a0,a1} = (f x); K
-------------------- DUP-APP
a0 <- (f0 x0)
a1 <- (f1 x1)
!&L{f0,f1} = f;
!&L{x0,x1} = x;
K
```

DUP PERMUTATIONS

These interactions move a nested DUP out of a redex position.

```
(!&L{k0,k1}=k;f x)
----------------- APP-DUP
!&L{k0,k1}=k;(f x)

! &L{x0,x1} = (!$R{y0,y1}=Y;X); T
----------------------------------- DUP-DUP
! &L{x0,x1} = X; ! &L{y0,y1} = Y; T
```

They're only needed in implementations that store a DUP's body.

LABELED LAMBDAS

Another possible extension of IC is to include labels on lams/apps:

```
  | LAM: "&" Label "λ" Name "." Term
  | APP: "&" Label "(" Term " " Term ")"
```

The APP-LAM rule must, then, be extended with:

```
&L(&Rλx.bod arg)
---------------------- APP-LAM (if different labels)
x <- &Lλy.z
&Rλz.&L(body &R(arg y))
```

IC = LAMBDA CALCULUS U INTERACTION COMBINATORS

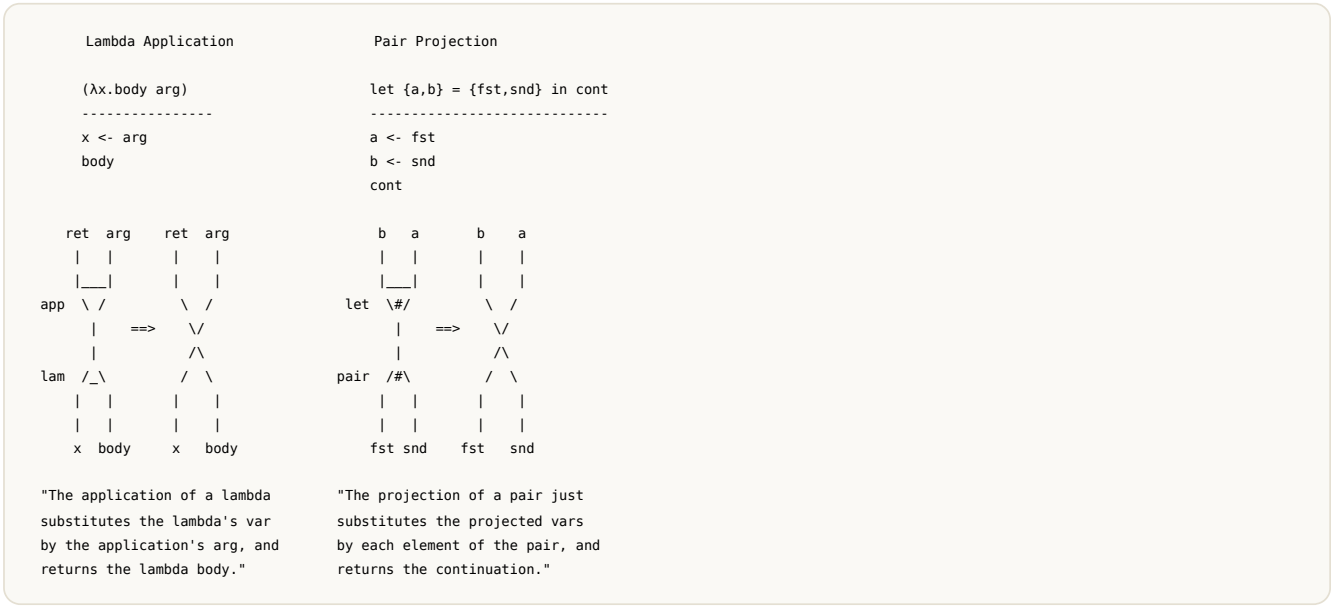
Consider the conventional Lambda Calculus, with pairs. It has two computational rules:

- Lambda Application : `(λx.body arg)`
- Pair Projection : `let {a,b} = {fst,snd} in cont`

When compiling the Lambda Calculus to Interaction Combinators:

- `lams` and `apps` can be represented as constructor nodes (γ)
- `pars` and `lets` can be represented as duplicator nodes (δ)

As such, lambda applications and pair projections are just annihilations:



But annihilations only happen when identical nodes interact. On interaction nets, it is possible for different nodes to interact, which triggers another rule, the commutation. That rule could be seen as handling the following expressions:

- Lambda Projection : `let {a b} = (λx body) in cont`
- Pair Application : `({fst snd} arg)`

But how could we ”project” a lambda or ”apply” a pair? On the Lambda Calculus, these cases are undefined and stuck, and should be type errors. Yet, by interpreting the effects of the commutation rule on the interaction combinator point of view, we can propose a reasonable reduction for these lambda expressions:

```
 Lambda Application                      Pair Application

let {a,b} = (λx.body) in cont            ({fst,snd} arg)
------------------------------            ---------------
a <- λx0.b0                              let {x0,x1} = arg in
b <- λx1.b1                              {(fst x0),(snd x1)}
x <- {x0,x1}
let {b0,b1} = body in
cont

  ret  arg         ret  arg            ret  arg         ret  arg
   |   |            |    |              |   |            |    |
   |___|            |    |              |___|            |    |
let  \#/           /_\  /_\         app  \ /            /#\  /#\
     |      ==>    |  \/  |              |      ==>    |  \/  |
     |             |_ /\ _|              |             |_ /\ _|
lam  /_\           \#/  \#/        pair  /#\           \ /  \ /
    |   |           |    |              |   |            |    |
    |   |           |    |              |   |            |    |
    x  body         x   body           var body         var  body

"The projection of a lambda         "The application of a pair is a pair
substitutes the projected vars      of the first element and the second
by a copies of the lambda that      element applied to projections of the
return its projected body, with     application argument."
the bound variable substituted
by the new lambda vars paired."
```

This, in a way, completes the lambda calculus; i.e., previously "stuck" expressions now have a meaningful computation. That system, as written, is Turing complete, yet, it is very limited, since it isn't capable of cloning pairs, or cloning cloned lambdas. There is a simple way to greatly increase its expressivity, though: by decorating lets with labels, and upgrading the pair projection rule to:

```
let &i{a,b} = &j{fst,snd} in cont
---------------------------------
if i == j:
  a <- fst
  b <- snd
  cont
else:
  a <- &j{a0,a1}
  b <- &j{b0,b1}
  let &i{a0,a1} = fst in
  let &i{b0,b1} = snd in
  cont
```

That is, it may correspond to either an Interaction Combinator annihilation or commutation, depending on the value of the labels `&i` and `&j`. This makes IC capable of cloning pairs, cloning cloned lambdas, computing nested loops, performing Church-encoded arithmetic up to exponentiation, expressing arbitrary recursive functions such as the Y-combinators and so on. In other words, with this simple extension, IC becomes extraordinarily powerful and expressive, giving us a new foundation for symbolic computing, that is, in many ways, very similar to the λ-Calculus, yet, with key differences that make it more efficient in some senses, and capable of expressing new things (like call/cc, O(1) queues, linear HOAS), but unable to express others (like `λx.(x x)`).

IC32: A 32-BIT RUNTIME

IC32 is implemented in portable C.

Each Term is represented as a 32-bit word, split into the following fields:

- sub (1-bit): true if this is a substitution
- tag (5-bit): the tag identifying the term type and label
- val (26-bit): the value, typically a pointer to a node in memory

The tag field can be one of the following:

- `VAR` : 0x00
- `LAM` : 0x01
- `APP` : 0x02
- `ERA` : 0x03

- `NUM` : 0x04
- `SUC` : 0x05
- `SWI` : 0x06
- `TMP` : 0x07
- `SP0` : 0x08
- `SP1` : 0x09
- `SP2` : 0x0A
- `SP3` : 0x0B
- `SP4` : 0x0C
- `SP5` : 0x0D
- `SP6` : 0x0E
- `SP7` : 0x0F
- `CX0` : 0x10
- `CX1` : 0x11
- `CX2` : 0x12
- `CX3` : 0x13
- `CX4` : 0x14
- `CX5` : 0x15
- `CX6` : 0x16
- `CX7` : 0x17
- `CY0` : 0x18
- `CY1` : 0x19
- `CY2` : 0x1A
- `CY3` : 0x1B
- `CY4` : 0x1C
- `CY5` : 0x1D
- `CY6` : 0x1E
- `CY7` : 0x1F

The val field depends on the variant:

- `VAR` : points to a Lam node ({bod: Term}) or a substitution.
- `LAM` : points to a Lam node ({bod: Term}).
- `APP` : points to an App node ({fun: Term, arg: Term}).
- `ERA` : unused.
- `NUM` : stores an unsigned integer.
- `SUC` : points to a Suc node ({num: Term})
- `SWI` : points to a Swi node ({num: Term, ifZ: Term, ifS: Term})
- `SP{L}` : points to a Sup node ({lft: Term, rgt: Term}).
- `CX{L}` : points to a Dup node ({val: Term}) or a substitution.
- `CY{L}` : points to a Dup node ({val: Term}) or a substitution.

A node is a consecutive block of its child terms. For example, the SUP term points to the memory location where its two child terms are stored.

Variable terms (`VAR` , `CX{L}` , and `CY{L}`) point to the location where the substitution will be placed. As an optimization, that location is always the location of the corresponding binder node (like a Lam or Dup). When the interaction occurs, we replace the binder node by the substituted term, with the 'sub' bit set. Then, when we access it from a variable, we retrieve that term, clearing the bit.

On SUPs and DUPs, the 'L' stands for the label of the corresponding node.

Note that there is no explicit DUP term. That's because Dup nodes are special: they aren't part of the AST, and they don't store a body; they "float" on the heap. In other words, `λx. !&0{x0,x1}=x; &0{x0,x1}` and `!&0{x0,x1}=x; λx. &0{x0,x1}` are both valid, and stored identically in memory. As such, the only way to access a Dup node is via its bound variables, `CX{L}` and `CY{L}` .

Before the interaction, the Dup node stores just the duplicated value (no body). After a collapse is triggered (when we access it via a `CX{L}` or `CY{L}` variable), the first half of the duplicated term is returned, and the other half is stored where the Dup node was, allowing the other variable to get it as a substitution. For example, the DUP-SUP interaction could be implemented as:

```
def dup_sup(dup, sup):
  dup_lab = dup.tag & 0x3
  sup_lab = sup.tag & 0x3
  if dup_lab == sup_lab:
    tm0 = heap[sup.loc + 0]
    tm1 = heap[sup.loc + 1]
    heap[dup.loc] = as_sub(tm1 if (dup.tag & 0x4) == 0 else tm0)
    return (tm0 if (dup.tag & 0x4) == 0 else tm1)
  else:
    co0_loc = alloc(1)
    co1_loc = alloc(1)
    su0_loc = alloc(2)
    su1_loc = alloc(2)
    su0_val = Term(SP0 + sup_lab, su0_loc)
    su1_val = Term(SP0 + sup_lab, su1_loc)
    heap[co0_loc] = heap[sup.loc + 0]
    heap[co1_loc] = heap[sup.loc + 1]
    heap[su0_loc + 0] = Term(CX0 + dup_lab, co0_loc)
    heap[su0_loc + 1] = Term(CX0 + dup_lab, co1_loc)
    heap[su1_loc + 0] = Term(CY0 + dup_lab, co0_loc)
    heap[su1_loc + 1] = Term(CY0 + dup_lab, co1_loc)
    heap[dup.loc] = as_sub(su1_val if (dup.tag & 0x4) == 0 else su0_val)
    return (su0_val if (dup.tag & 0x4) == 0 else su1_val)
```

The NUM, SUC and SWI terms extend the IC with unboxed unsigned integers.

PARSING IC32

On IC32, all bound variables have global range. For example, consider the term:

λt.((t x) λx.λy.y)

Here, the `x` variable appears before its binder, `λx .` Since runtime variables must point to their bound λ's, linking them correctly requires caution. A way to do it is to store two structures at parse-time:

1. lcs: an array from names to locations
2. vrs: a map from names to var terms

Whenever we parse a name, we add the current location to the 'uses' array, and whenever we parse a binder (lams, lets, etc.), we add a variable term pointing to it to the 'vars' map. Then, once the parsing is done, we run iterate through the 'uses' array, and write, to each location, the corresponding term. Below are some example parsers using this strategy:

```
 def parse_var(loc):
   nam = parse_name()
   uses.push((nam,loc))

def parse_lam(loc):
   lam = alloc(1)
   consume("λ")
   nam = parse_name()
   consume(".")
   vars[nam] = Term(VAR, 0, lam)
   parse_term(lam)
   heap[loc] = Term(LAM, 0, lam)

def parse_app(loc):
   app = alloc(2)
   consume("(")
   parse_term(app + 0)
   consume(" ")
   parse_term(app + 1)
   consume(")")
   heap[loc] = Term(APP, 0, app)

def parse_sup(loc):
   sup = alloc(2)
   consume("&")
   lab = parse_uint()
   consume("{")
   lft = parse_term(sup + 0)
   consume(",")
   rgt = parse_term(sup + 1)
   consume("}")
   heap[loc] = Term(SUP, lab, sup)

def parse_dup(loc):
   dup = alloc(1)
   consume("!")
   consume("&")
   lab = parse_uint()
   consume("{")
   co0 = parse_name()
   consume(",")
   co1 = parse_name()
   consume("}")
   consume("=")
   val = parse_term(dup)
   bod = parse_term(loc)
   vars[co0] = Term(DP0, lab, loc)
   vars[co1] = Term(DP1, lab, loc)
```

STRINGIFYING IC32

Converting IC32 terms to strings faces two challenges:

First, IC32 terms and nodes don't store variable names. As such, we must generate fresh, unique variable names during stringification, and maintain a mapping from each binder's memory location to its assigned name.

Second, on IC32, Dup nodes aren't part of the main program's AST. Instead, they "float" on the heap, and are only reachable via DP0 and DP1 variables. Because of that, by stringifying a term naively, Col nodes will be missing.

To solve these, we proceed as follows:

1. Before stringifying, we pass through the full term, and assign a id to each variable binder we find (on lam, let, dup, etc.)
2. We also register every Dup node we found, avoiding duplicates (remember the same dup node is pointed to by up to 2 variables, DP0 and DP1)

Then, to stringify the term, we first stringify each DUP node, and then we stringify the actual term. As such, the result will always be in the form:

```
 ! &{x0 x1} = t0
! &{x2 x3} = t1
! &{x4 x5} = t2
...
term
```

With no Dup nodes inside the ASTs of t0, t1, t2 ... and term.

H.34 · bend-docs/openspec/changes/bend-docs-site/research/repos/VictorTaelin_hvm4-formal-README.md

404: Not Found

H.35 · bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-collapser.md

Collapser (CNF Readback)

Interaction Calculus is affine and uses explicit duplication (DUP) and superposition (SUP). That is great for optimal sharing but hard for humans to read. The collapser is the readback procedure: it turns one IC term into a stream of ordinary lambda terms, i.e., collapsed normal form (CNF).

Two insights make this possible:

- Quoting removes DUPs. When a branch is ready to print, we run `cnf(term)`. Linked vars become quoted vars (BJV/BJ0/BJ1). DUP interactions are defined on quoted dup vars, so DUP nodes are cloned away and disappear from the printed term.
- Lifting removes SUPs. We lift the first SUP to the top and enumerate its branches. Same-label SUPs annihilate pairwise; different-label SUPs commute and create a cross product of branches.

ALGORITHM (AS IMPLEMENTED)

- `cnf` (`src/hvm.c`, CNF section): reduce to WNF, then lift the first SUP to the top and return immediately. ERA propagates upward; INC is left in place for the flattener.
- `eval_collapse` (`src/hvm.c`, Eval section): breadth-first traversal with a sequential priority queue. Lower numeric keys are popped first; SUP increases key, INC decreases key. Branches with the same key are popped FIFO for deterministic ordering. When a branch has no SUP, it prints `cnf(term)`.

LABEL BEHAVIOR (PAIRWISE VS CROSS PRODUCT)

Different labels commute (cross product):

```
[&A{1,2}, &B{3,4}]
```

```
[1,3]
[1,4]
[2,3]
[2,4]
```

Same labels annihilate pairwise:

```
[&A{1,2}, &A{3,4}]
```

```
[1,2]
[3,4]
```

WHERE TO LOOK

- `src/hvm.c`, `CNF` section: SUP lifting rules.
- `src/hvm.c`, `Eval` section: branch enumeration + SNF quoting for output.

H.36 · bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-core.md

HVM Core Terms

This document defines the core surface syntax for HVM terms. These terms are parsed into static (book) terms and later instantiated as dynamic terms during execution. See `docs/hvm/memory.md` for the dynamic/static memory layout.

GRAMMAR

```
Term ::=
  | Var  Name                                    -- variable
  | Dp0  Name "₀"                                -- first dup variable
  | Dp1  Name "₁"                                -- second dup variable
  | Ref  "@" Name                                -- reference
  | Nam  "^" Name                                -- name (stuck head)
  | Dry  "^" "(" Term " " Term ")"               -- dry (stuck application)
  | Era  "&{}"                                   -- erasure
  | Sup  "&" Label "{" Term "," Term "}"         -- superposition
  | Dup  "!" Name "&" Label "=" Term ";" Term    -- duplication term
  | Ctr  "#" Name "{" Term,* "}"                 -- constructor
  | Mat  "λ" "{" "#" Name ":" Term ";" Term "}"  -- pattern match
  | Swi  "λ" "{" Num ":" Term ";" Term "}"       -- number switch
  | Use  "λ" "{" Term "}"                        -- use (unbox)
  | Lam  "λ" Name "." Term                       -- lambda
  | App  "(" Term " " Term ")"                   -- application
  | Num  integer                                 -- number literal
  | Op2  "(" Term Oper Term ")"                  -- binary operation
  | Eql  "(" Term "==" Term ")"                  -- equality test
  | And  "(" Term ".&." Term ")"                 -- short-circuit AND
  | Or   "(" Term ".|." Term ")"                 -- short-circuit OR
  | DSu  "&" "(" Term ")" "{" Term "," Term "}"  -- dynamic superposition
  | DDu  "!" Name "&" "(" Term ")" "=" Term ";" Term -- dynamic duplication term
  | Inc  "↑" Term                                -- priority wrapper
  | Alo  "@" "{" Name,* "}" Term                 -- allocation
  | Uns  "!" "$" "{" Name "," Name "}" ";" Term  -- unscoped binding

Name  ::= [_A-Za-z0-9]+
Label ::= Name
Oper  ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||"
        | "^" | "~" | "<<" | ">>" | "==" | "!="
        | "<" | "<=" | ">" | ">="
```

NOTES

- Variables are affine: each variable is used at most once.
- Variables are global: a variable can occur outside its binder's lexical scope.
- Labels determine how duplications and superpositions interact; equal labels annihilate, different labels commute.
- Surface sugar accepts `λ$x. body` as an unscoped lambda, equivalent to `! f = λ x ; f(body)` with fresh `f` (see `docs/hvm/syntax.md`).

H.37 · bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-memory.md

HVM Memory Layout

HVM represents every term as a 64-bit word. The same layout is used for dynamic terms (mutable heap entries) and static terms (immutable book entries).

TERM POINTER LAYOUT (64-BIT)

```
TAG (8 bits) | EXT (24 bits) | VAL (32 bits)
```

- `TAG` : constructor variant (APP, LAM, SUP, etc.); the high bit is the SUB flag.
- `EXT` : metadata (dup label, ctor/ref name, level, op code, flags).
- `VAL` : payload (heap loc, immediate, or 0).

TERM MEMORY TABLE

TERM	TAG	EXT	VAL	NOTES
application	APP	0	node: [func, arg]	dynamic or static
linked variable	VAR	0	lam body or subst	dynamic only; follows SUB cells

TERM	TAG	EXT	VAL	NOTES
linked duplicate 0	DP0	label	dup expr or subst	dynamic only; twin of DP1
linked duplicate 1	DP1	label	dup expr or subst	dynamic only; twin of DP0
linked lambda	LAM	level+flags	node: [body]	dynamic only; binder for VAR
quoted lam	LAM	level+flags	node: [body]	quoted; level stored in EXT
superposition	SUP	label	node: [tm0, tm1]	dynamic or static
linked duplication term	DUP	label	node: [expr, body]	dynamic or static; binder for DP0/DP1
quoted duplication term	DUP	label	node: [expr, body]	expr typically `&L{BJ0,BJ1}` in quoted mode
number literal	NUM	0	unboxed u32	
constructor arity 0	C00	ctor name	0	tag encodes arity
constructor arity N	C01-16	ctor name	node: [field0..fieldN-1]	N = tag - C00
pattern match	MAT	ctor name	node: [hit, miss]	match on constructor
number switch	SWI	number literal	node: [zero, succ]	parser/printer distinction from MAT
use (unbox)	USE	0	node: [fun]	forces argument, then applies
binary op	OP2	op code	node: [lhs, rhs]	EXT stores OP_ADD, OP_MUL, etc.
equality	EQL	0	node: [lhs, rhs]	structural equality
logical AND	AND	0	node: [lhs, rhs]	short-circuit AND
logical OR	OR	0	node: [lhs, rhs]	short-circuit OR
priority wrapper	INC	0	node: [term]	collapse priority wrapper
name literal	NAM	name id	0	literal ^name
stuck application	DRY	0	node: [fun, arg]	literal ^(f x)
reference	REF	name id	0	book reference @name
allocation	ALO	bind list length	direct or packed pair	len=0: VAL=book term; len>0: VAL->(low32=book term, high32=bind list head)
unscoped binding	UNS	0	node: [body]	helper to construct unscoped lams
wildcard	ANY	0	0	duplicates itself, equals anything
quoted lam var	BJV	0	de Bruijn level	quoted lam-bound var
quoted dup var 0	BJ0	label	de Bruijn level	quoted dup-bound var
quoted dup var 1	BJ1	label	de Bruijn level	quoted dup-bound var
dynamic superposition	DSU	0	node: [lab, tm0, tm1]	label computed dynamically
dynamic duplication	DDU	0	node: [lab, val, body]	label computed dynamically

SUBSTITUTION CELLS (SUB BIT)

When a linked binder interacts (APP-LAM, DUP-*), its body or expr slot is replaced by a substitution term with the SUB bit set. Any VAR/DP0/DP1 that points to that heap location must read the substitution instead of the original binder. This is how linked variables resolve their binders without extra maps.

DUP NODES VS DUP TERMS

A DUP term is the syntactic binder `[expr, body]` with tag `DUP`. A DUP node is the shared expr slot that DP0/DP1 point to; it has no body and no parent, and it is substituted when a duplication interaction occurs.

LINKED VS QUOTED BINDERS

Linked binders are dynamic lams/dups whose vars point to heap locations. Linked LAM uses `EXT = level | flags` and its VARs point to the body slot. Linked DUP uses DP0/DP1 vars that point to the shared expr slot.

Quoted binders are terms encoded with de Bruijn levels in `EXT` (and for BJ0/BJ1, the dup label). Their variables are BJV/BJ0/BJ1 indices, so there are no heap links and no interaction with APP-LAM or DUP-SUP.

During collapse, linked LAM/DUP binders are converted into quoted LAM/DUP binders, and free VAR/DP0/DP1 become BJV/BJ0/BJ1 at the current level. This is used to turn interaction nets into full lambda terms.

DYNAMIC VS STATIC TERMS

Dynamic terms live on the heap and are mutable: binders can be replaced by substitutions, and DUP nodes evolve as duplication is forced. Static terms live in the book and are immutable; they store de Bruijn levels and are never updated in place.

Allocation (ALO) terms bridge the two: they reference a static book term and a bind list, lazily expanding one layer into a dynamic term when forced. This keeps static definitions compact while still allowing dynamic sharing during execution.

ALO RUNTIME ENCODINGS

- `ALO` with `len == 0` (empty substitution list) is stored directly:
  - `ALO.val = book_term_loc`
  - no extra heap allocation for an ALO pair node.
- `ALO` with `len > 0` stores one packed ALO pair word at `ALO.val`:
  - `high 32 bits = bind list head location`
  - `low 32 bits = book term location`

This keeps `len > 0` ALO pairs at one heap word while preserving the full 32-bit location range for both bind-list heads and book terms.

ALO BIND-LIST NODES

For `len > 0`, the bind list is a linked list of 2-slot nodes:

- `node[0]`: bound term cell (the substitution target term lives here).
- `node[1]`: `NUM(next_node_loc)`; `0` means end of list.

This means the pointer used by variables/copiers is the bind-node location itself (slot `0`), not a separate allocation.

- In `ALO-LAM`, `node[0]` is the lambda body slot.
- In `ALO-DUP`, `node[0]` is the shared DUP expr slot.

LAM EXT FLAGS

- `LAM_ERA_MASK` (0x800000): binder is unused in lambda body (erasing lambda).

H.38 · bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/hvm-syntax.md

HVM Parser Syntax

Desugarings use the core surface syntax from `docs/hvm/core.md`. Core application is written there as `(f x)`, so this doc uses that notation for desugaring only; the parser itself does not accept whitespace application.

PARSING MODEL (HOW TERMS GROUP)

- Base unit: an atom.
- Postfix call: `f(...)` binds tightest.
- `<>` binds tighter than infix operators.
- `===`, `.&.`, `.|.` bind tighter than infix operators (same precedence class as `==`, `&&`, `||`, but separate nodes).
- Infix operators are left-associative, by the precedence table below.
- No whitespace application: `f x` is not an application; use `f(x)`.

LEXICAL RULES

- Whitespace: space, tab, newline, `\r`.
- Line comments: `// ...`.
- Names: `_a-zA-Z0-9$`.
  - A term starting with a digit is parsed as a number literal, not a name.

ATOMS

- Variable: `x`
- Dup variables: `x₀` or `x₁` (see "Dup variables")
- Grouping: `(term)`
- Constructor: `#Name` or `#Name{a,b,c}` (`,` and `;` are equivalent, trailing separator ok)
- Reference: `@Name`
- Name head: `^Name`
- Dry app: `^(f x)`
- Priority wrapper: `↑atom`
- Wildcard: `*`
- Lambda / dup / sup / match / literals (see below)

APPLICATION

- `f(a, b, c)` desugars to `((f a) b) c`.
  - `,` and `;` are equivalent, and separators are optional: `f(a b c)` is the same.
  - `f()` is allowed and desugars to `f`.

LITERALS

NUMBERS

- `123` is a numeric literal.

NATURALS (PEANO)

Using built-in constructors `#Zer` and `#Suc`:

- `Nn` desugars to `#Suc{...#Suc{#Zer}...}` (N times)
- `Nn+term` desugars to `#Suc{...#Suc{term}...}` (N times)
  - `n+` must be contiguous; `3n + x` is not a natural literal.

Examples:

```
3n    => #Suc{#Suc{#Suc{#Zer}}}
2n+xs => #Suc{#Suc{xs}}
```

CHARACTERS

- `'c'` desugars to `#Chr{#<codepoint>}`.

Escapes:

- Standard escapes: `\n`, `\t`, `\r`, `\0`, `\\`, `\'`, `\"`.

- Other backslashes escape the next byte verbatim.
- Otherwise UTF-8 sequences are decoded into a numeric codepoint.

STRINGS

Strings are lists of `#Chr`:

- `"str"` desugars to `#Con{#Chr{#s}, #Con{#Chr{#t}, #Con{#Chr{#r}, #Nil}}}`.

Same escape rules as characters.

LISTS

Lists use built-in `#Nil` and `#Con`:

- `[]` desugars to `#Nil`.
- `[a, b, c]` desugars to `#Con{a, #Con{b, #Con{c, #Nil}}}`.
  - `,` and `;` are equivalent; trailing separator allowed.
- `a <> b` desugars to `#Con{a, b}` (cons sugar).

LAMBDAS AND BINDERS

LAMBDA

- `λx. body` is core `Lam`.
- `λx,y,z. body` desugars to `λx. λy. λz. body`.
  - `,` and `;` are equivalent between binders.
- `λ$x. body` is an unscoped lambda binder (see "Unscoped lambda (UNS)").

CLONED BINDERS (AUTO-DUP)

Prefix `&` allows multiple uses; the parser inserts dup nodes.

- `λ&x. body` desugars to `λx. body` plus auto-duplication of `x`.
- Auto-duplication expands a cloned variable with N uses into N-1 nested dups using fresh labels. Example for three uses: `body[x,x,x]` becomes `!d0&=x; !d1&=d0₁; body[d0₀,d1₀,d1₁]`.

INLINE DUPLICATION ON LAMBDAS

Short form for inserting a dup on the bound variable:

- `λx&L. body` desugars to `λx. !x&L = x; body`.
- `λx&. body` uses a fresh label.
- `λx&(lab). body` desugars to `λx. !x&(lab) = x; body`.

All of these combine with cloning: `λ&x&L. ...`.

DUPLICATION AND LET (`!`)

LET SUGAR

- `!x = v; body` desugars to `((λx. body) v)`.
- `!!x = v; body` desugars to `((λ{λx. body}) v)` (strict let).
- Cloned forms `!&x = ...` and `!!&x = ...` allow multiple uses of `x` and insert auto-dup as needed.

DUPLICATION (STATIC LABEL)

- `!x&L = v; body` is core `Dup`.
- `!x& = v; body` uses a fresh label.
- Optional `,` or `;` after `v` is allowed.
- Cloned dup binder: `!&x&L = v; body` (or `!&x& = ...`).

DUP DESTRUCTURING SUGAR

Short form for binding each side of a duplicated value as a plain variable:

- `! &L{x,y} = v; body` desugars like `!x_y&L = v; body[x := x_y₀, y := x_y₁]`.
- `! &{x,y} = v; body` uses a fresh label.
- `! &(lab){x,y} = v; body` uses a dynamic label.
- Branch binders can be cloned independently: `! &L{&x,y} = v; body`.
- `,` and `;` are equivalent between branch binders; trailing separator allowed.

DUPLICATION (DYNAMIC LABEL)

- `!x&(lab) = v; body` is core `DDu`.
- `!&x&(lab) = v; body` is the cloned form.

UNSCOPED LAMBDA (UNS)

- `! f = λ x ; body` desugars to `!${f,x}; body`.
- `λ$x. body` desugars to `! f = λ x ; f(body)` with fresh `f`.
  - `,` and `;` are allowed between binders: `λ$x,y. body` (x unscoped, y scoped).
  - The unscoped variable is referenced as plain `x` inside the body.

SUPERPOSITION (&) AND FORK

SUPERPOSITION AND ERASURE

- `&L{a, b}` is core `Sup`.
- `&(lab){a, b}` is core `DSu`.
- `&{}` is core `Era`.
  - `,` and `;` are equivalent; trailing separator allowed.

AUTO-FORK SUGAR

```
&L!{A;B}
```

desugars to `&L[&x,&y,&z,...]{A;B}` where `x,y,z,...` are all variables currently in scope (lambda/let bindings and fork bindings, excluding raw dup bindings that require subscripts). All captured variables are automatically cloned. Dynamic labels are supported: `&(lab)!{A;B}`.

This is a convenience for the common pattern of forking the entire context into a superposition's two branches.

FORK SUGAR

```
&Lλx,y{A;B}
```

desugars to:

```
λx&L. λy&L. &L{A', B'}
```

where `A'` uses `x₀`, `y₀` and `B'` uses `x₁`, `y₁`.

Inside each branch you can omit `₀` / `₁` and the parser chooses the side automatically. You can also prefix branches with `&₀:` or `&₁:` to force the side for all un-subscripted dup variables in that branch. `,` and `;` are equivalent between branches; trailing separator allowed.

Dynamic labels are supported: `&(lab)λx,y{A;B}`.

MATCH / SWITCH / USE (λ{...})

ERASURE

- `λ{}` desugars to `&{}`.

USE / UNBOX

- `λ{term}` is core `Use`.
- `λ{_: term}` also desugars to `λ{term}` if it is the only entry.

PATTERN MATCH AND NUMERIC SWITCH

Syntax:

```
λ{
  pat1: t1;
  pat2: t2;
  ...
  _ : default
}
```

Patterns:

- `#Name:` match constructor `#Name`
- `[]:` match `#Nil`
- `<>:` match `#Con`
- `0n:` match `#Zer`
- `1n+:` match `#Suc` (digits before `n+` are ignored)
- `N:` numeric switch on literal `N`
- `_:` default branch

Desugaring is right-nested:

```
λ{#A: t1; #B: t2; d}
=> λ{#A: t1; λ{#B: t2; d}}

λ{0: t0; 1: t1; d}
=> λ{0: t0; λ{1: t1; d}}
```

`,` and `;` are equivalent between cases; trailing separator allowed, including after the final default. The final default can be written as `_ : d` or as a bare `d`.

NAMES AND REFERENCES

- `@Name` is a book reference.
- `^Name` is a stuck name head.
- `^(f x)` is a dry application.

PRIORITY WRAPPER AND WILDCARD

- `↑atom` is core `Inc` (binds to a single atom).
- `*` is core `Any`.

INFIX OPERATORS

All infix operators are left-associative.

Precedence (higher binds tighter):

- 8: `^`
- 7: `*` `/` `%`
- 6: `+` `-`
- 5: `<<` `>>`
- 4: `<` `<=` `>` `>=`

- 3: `==` `!=`
- 2: `&&`
- 1: `||`

Notes:

- `~` parses as `OP_NOT` but is infix. Evaluation ignores the left operand and uses only the right one (`~b`). For unary not, write `0 ~ x`.

STRUCTURAL EQUALITY AND SHORT-CIRCUIT OPS

These are separate nodes (not `Op2`):

- `a === b` structural equality
- `a .&. b` short-circuit AND
- `a .|. b` short-circuit OR

DUP VARIABLES AND SUBSCRIPTS

- Variables bound by `Dup` (or inline `λx&L`) must be referenced as x_0 or x_1.
- Inside fork branches, the parser can auto-select the side if you omit the subscript.
- Dynamic dup variables (`!x&(lab)=...`) always require a subscript.

NOT PARSED DIRECTLY

Some internal terms have print forms but are not parsed directly (for example `@{...}` for `ALO`). The list above reflects what the parser accepts today.

H.39 · bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/ic-theory.md

Interaction Calculus

The Interaction Calculus is a model of computation that extends the Lambda Calculus with **duplications** and **superpositions**, two core forms that enable optimal lazy evaluation.

CONTENTS

- Optimal Lazy Evaluation
- Duplications
- Superpositions
- Duplicating Lambdas
- Dup/Sup Labels
- The Four Core Interactions
- Relation to Interaction Combinators
- Complete Examples

OPTIMAL LAZY EVALUATION

Programs can be evaluated **strictly** (compute everything immediately) or **lazily** (compute only when needed). Lazy evaluation avoids unnecessary work, but has a flaw: when a value is used twice, its computation might happen twice. Haskell solves this with *memoization* (thunks), caching results so repeated access doesn't recompute. This works for data, but breaks down inside lambdas.

The Interaction Calculus achieves **optimal sharing**: work is never duplicated, even inside lambdas. Two dual forms make this possible:

- **Duplications**: allow a single value to exist in multiple locations
- **Superpositions**: allow multiple values to exist in a single location

These correspond to Lafont's "fan nodes" in Interaction Combinators (1997), with opposite polarities. Together, they provide the mechanics for lazy cloning that extends into lambda bodies.

DUPLICATIONS

Duplications allow a **single value** to exist in **two locations**. The `! x &= v; t` construct duplicates `v`, making it available as `x₀` and `x₁` in `t`:

```
! x &= 2;
(x₀ + x₁)
--------- DUP-NUM
(2 + 2)
--------- OP2-NUM
4
```

Here, the number `2` was duplicated, then added to itself.

Duplication is incremental - it happens layer by layer, on demand:

```
! x &= [1, 2, 3];
(x₀, x₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [2, 3];
(1 <> t₀, 1 <> t₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [3];
(1 <> 2 <> t₀, 1 <> 2 <> t₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [];
(1 <> 2 <> 3 <> t₀, 1 <> 2 <> 3 <> t₁)
------------------------------------ DUP-CTR
([1, 2, 3], [1, 2, 3])
```

The list was cloned element by element. Each step peels off one layer - duplicating the head and creating a duplication for the tail. The tail duplication only triggers when both copies are accessed.

SUPERPOSITIONS

Superpositions allow **two values** to exist in **one location**. The `&{a, b}` construct creates a superposition of `a` and `b`:

```
(&{1, 2} + 10)
--------------------- OP2-SUP
! x &= 10;
&{(1 + x₀), (2 + x₁)}
--------------------- DUP-NUM
&{(1 + 10), (2 + 10)}
--------------------- OP2-NUM, OP2-NUM
&{11, 12}
```

Here, we added `10` to a superposition of `1` and `2`. The addition applied to both values, producing a superposition of results. Notice that `10` had to be duplicated; SUPs generate DUP nodes (DP0/DP1) as byproducts, and vice-versa.

Superpositions and duplications are duals. When a DUP meets a SUP, they annihilate, extracting the two values:

```
! x &= &{1, 2};
(x₀ + x₁)
--------------- DUP-SUP
(1 + 2)
--------------- OP2-NUM
3
```

This is like a pair projection: `x₀` gets the first element, `x₁` gets the second. SUPs and DUP nodes create and eliminate each other, just like LAMs and APPs.

DUPLICATING LAMBDAS

Lambdas are also duplicated incrementally. When that happens, the bound variable becomes superposed, and temporarily escapes its scope:

```
! f &= λx.(x + 1);
(f₀(10), f₁(20))
---------------------------- DUP-LAM
! b &= (&{$x0, $x1} + 1);
((λ$x0.b₀)(10), (λ$x1.b₁)(20))
---------------------------- APP-LAM, APP-LAM
! b &= (&{10, 20} + 1);
(b₀, b₁)
---------------------------- OP2-SUP, DUP-NUM
! b &= &{(10 + 1), (20 + 1)};
(b₀, b₁)
---------------------------- OP2-NUM, OP2-NUM
! b &= &{11, 21};
(b₀, b₁)
---------------------------- DUP-SUP
(11, 21)
```

Notice how, on the first step, the variables `$x0` and `$x1` are bound *outside* the lambda's body. This is why the Interaction Calculus needs globally scoped variables. It's also what enables **optimal sharing inside lambdas**: the body is now shared by `b`, so any computation there only happens once.

DUP/SUP LABELS

Consider this example:

```
(&A{1, 2} + &A{10, 20})
---------------------- OP2-SUP
! x &A= &A{10, 20};
&A{(1 + x₀), (2 + x₁)}
---------------------- DUP-SUP (same label: annihilate)
&A{(1 + 10), (2 + 20)}
---------------------- OP2-NUM, OP2-NUM
&A{11, 22}
```

Here, the superpositions *annihilated*: the first element paired with the first, and the second with the second. But what if we wanted all combinations instead?

Labels control this behavior. When a DUP meets a SUP with the *same* label, they annihilate. With *different* labels, they commute:

```
(&A{1, 2} + &B{10, 20})
---------------------------------------------- OP2-SUP
! x &A= &B{10, 20};
&A{(1 + x₀), (2 + x₁)}
---------------------------------------------- DUP-SUP (different labels: commute)
! a &A= 10;
! b &A= 20;
&A{(1 + &B{a₀, b₀}), (2 + &B{a₁, b₁})}
---------------------------------------------- DUP-NUM, DUP-NUM
&A{(1 + &B{10, 20}), (2 + &B{10, 20})}
---------------------------------------------- OP2-SUP, OP2-SUP
&A{&B{(1 + 10), (1 + 20)}, &B{(2 + 10), (2 + 20)}}
---------------------------------------------- OP2-NUM (x4)
&A{&B{11, 21}, &B{12, 22}}
```

Now we get a nested superposition containing all four results. Labels let us control whether superpositions collapse together or stay independent.

THE FOUR CORE INTERACTIONS

The minimal Interaction Calculus has just four rules. Two create computation (APP-LAM, DUP-SUP) and two propagate it (APP-SUP, DUP-LAM):

APP-LAM - Application eliminates lambda:

```
(λx.body)(arg)
-------------- APP-LAM
x ← arg
body
```

DUP-SUP - Duplication eliminates superposition (same label):

```
! x &L= &L{a, b}; t
------------------- DUP-SUP
x₀ ← a
x₁ ← b
t
```

APP-SUP - Application propagates through superposition:

```
(&L{a, b})(c)
------------------- APP-SUP
! x &L= c;
&L{a(x₀), b(x₁)}
```

DUP-LAM - Duplication propagates through lambda:

```
! f &L= λx.body; t
------------------ DUP-LAM
f₀ ← λ$X0.B₀
f₁ ← λ$X1.B₁
x  ← &L{$X0, $X1}
! B &L= body;
t
```

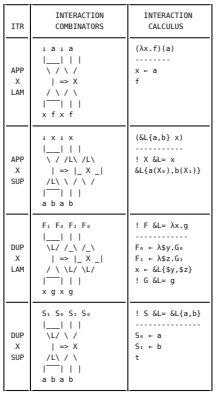
When labels differ, DUP-SUP commutes instead of annihilating:

```
! x &L= &R{a, b}; t
-------------------- DUP-SUP (L ≠ R)
! A &L= a;
! B &L= b;
x₀ ← &R{A₀, B₀}
x₁ ← &R{A₁, B₁}
t
```

These four rules form a complete system. Every other interaction in HVM is an extension for practical constructs: numbers, constructors, pattern-matching.

RELATION TO INTERACTION COMBINATORS

The Interaction Calculus is similar to Interaction Combinators, the graph rewriting model described by Lafont (1997). This similarity can be visualized as:



It can also be seen as a completion of the λ-Calculus, giving a computational meaning to previously undefined expressions: applying a pair and projecting a lambda. The Interaction Calculus provides sensible reduction rules for these, inspired by its Interaction Combinator equivalence: applying a pair (superposition) distributes over both elements (APP-SUP), and duplicating a lambda creates two lambdas with a superposed bound variable (DUP-LAM). This makes every possible interaction well-defined.

COMPLETE EXAMPLES

Here's an example demonstrating optimal sharing. We duplicate a lambda containing `(2 + 2)` and apply the copies to different arguments:

```
! F &= (λx. λy. ! z &= x; ((z₀ + z₁), y) 2);
(F₀(10), F₁(20))
---------------------------------------------- APP-LAM
! z &= 2;
! F &= λy.((z₀ + z₁), y);
(F₀(10), F₁(20))
---------------------------------------------- DUP-NUM
! F &= λy.((2 + 2), y);
(F₀(10), F₁(20))
---------------------------------------------- DUP-LAM
! B &= ((2 + 2), &{y0, y1});
((λy0.B₀ 10), (λy1.B₁ 20))
---------------------------------------------- APP-LAM (x2)
! B &= ((2 + 2), &{10, 20});
(B₀, B₁)
---------------------------------------------- DUP-CTR
! H &= (2 + 2);
! T &= &{10, 20};
((H₀, T₀), (H₁, T₁))
---------------------------------------------- OP2-NUM (shared!)
! H &= 4;
! T &= &{10, 20};
((H₀, T₀), (H₁, T₁))
---------------------------------------------- DUP-NUM, DUP-SUP
((4, 10), (4, 20))
```

Notice that `(2 + 2)` was computed only **once**, even though the lambda was duplicated and each copy was applied to a different argument. The result `4` flowed to both copies through the DUP-NUM interaction.

CHURCH 2^2

Here's a more complex example - computing 2^2 using Church numerals:

```
(λf. ! F &L= f; λx.(F₀ (F₁ x)) λg. ! G &K= g; λy.(G₀ (G₁ y)))
---------------------------------------------------------- APP-LAM
! F &L= λg. ! G &K= g; λy.(G₀ (G₁ y));
λx.(F₀ (F₁ x))
---------------------------------------------------------- DUP-LAM
! G &K= &L{g0, g1};
! F &L= λy.(G₀ (G₁ y));
λx.((λg0.F₀) (λg1.F₁ x))
---------------------------------------------------------- APP-LAM
! G &K= &L{(λg1.F₁ x), g1};
! F &L= λy.(G₀ (G₁ y));
λx.F₀
---------------------------------------------------------- DUP-LAM
! G &K= &L{(λg1.λy1.F₁ x), g1};
! F &L= (G₀ (G₁ &L{y0, y1}));
λx.λy0.F₀
---------------------------------------------------------- DUP-SUP (L ≠ K)
! A &K= (λg1.λy1.F₁ x);
! B &K= g1;
! F &L= (&L{A₀, B₀} (&L{A₁, B₁} &L{y0, y1}));
λx.λy0.F₀
---------------------------------------------------------- APP-SUP
! A &K= (λg1.λy1.F₁ x);
! B &K= g1;
! U &L= (&L{A₁, B₁} &L{y0, y1});
! F &L= &L{(A₀ U₀), (B₀ U₁)};
λx.λy0.F₀
---------------------------------------------------------- DUP-SUP (L = L)
! A &K= (λg1.λy1.(B₀ U₁) x);
! B &K= g1;
! U &L= (&L{A₁, B₁} &L{y0, y1});
λx.λy0.(A₀ U₀)
---------------------------------------------------------- APP-LAM
! A &K= λy1.(B₀ U₁);
! B &K= x;
! U &L= (&L{A₁, B₁} &L{y0, y1});
λx.λy0.(A₀ U₀)
---------------------------------------------------------- DUP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= (&L{λy11.A₁, B₁} &L{y0, &K{y10, y11}});
λx.λy0.((λy10.A₀) U₀)
---------------------------------------------------------- APP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= (&L{λy11.A₁, B₁} &L{y0, &K{U₀, y11}});
λx.λy0.A₀
---------------------------------------------------------- APP-SUP
! A &K= (B₀ U₁);
! B &K= x;
! V &L= &L{y0, &K{U₀, y11}};
! U &L= &L{((λy11.A₁) V₀), (B₁ V₁)};
λx.λy0.A₀
---------------------------------------------------------- DUP-SUP (L = L), APP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= &L{(A₁ y0), (B₁ &K{U₀, y11})};
λx.λy0.A₀
---------------------------------------------------------- DUP-SUP (L = L)
! A &K= (B₀ (B₁ &K{(A₁ y0), y11}));
! B &K= x;
λx.λy0.A₀
```

At this point, the computation is done. The result is a lambda `λx.λy0.A₀` connected to a small network of nodes. This compressed form represents the answer but is hard to read directly. To see the familiar Church numeral 4, we can **collapse** the Interaction Calculus term back into a proper λ-Term, by applying 2 extra interactions, DUP-VAR and DUP-APP:

```
...
-------------------------------- DUP-VAR
! A &K= (x (x &K{(A₁ y0), y11}));
λx.λy0.A₀
-------------------------------- DUP-APP, DUP-VAR
! X &K= (x &K{((x X₁) y0), y11});
λx.λy0.(x X₀)
-------------------------------- DUP-APP, DUP-VAR
! Y &K= &K{((x (x Y₁)) y0), y11};
λx.λy0.(x (x Y₀))
-------------------------------- DUP-SUP (K = K)
λx.λy0.(x (x ((x (x y11)) y0)))
-------------------------------- APP-LAM (x2)
λx.λy0.(x (x (x (x y0))))
```

The Church numeral 2 (`λf.λx.(f (f x))`) was applied to itself, yielding 4 (`λf.λx.(f (f (f (f x))))`). Despite temporarily escaping variables, the system correctly computes the result. This is the kind of symbolic computation where sharing matters most, and is often used as a benchmark in the literature. HVM completes it in 14 interactions, which is optimal.

H.40 · `bend-docs/openspec/changes/bend-docs-site/research/repos/hvm4-docs/primer.md`

HVM Primer

HVM is a runtime for the Interaction Calculus, a model of computation that extends lambda calculus to achieve **optimal reduction**: no work is ever duplicated, even when copying functions. It's lazy like Haskell, but sharing extends inside lambdas, not just at the data level.

RUNNING HVM

```
hvm file.hvm -s -C10 # run with collapse
```

Flags:

- `-s` : show performance stats
- `-D` : print intermediate reduction steps
- `-C10` : collapse superpositions (limit output to 10 lines)

Modes:

- *Normalize*: reduce to SNF. Keeps SUPs and DUPs in output.
- *Collapse*: eliminate SUPs and DUPs, enumerate as pure lambda terms.

GRAMMAR

```
File     ::= (Include | Def)*
Include  ::= "#include" '"' Path '"'
Def      ::= "@" Name "=" Term

Term ::=
  | Name                                  -- variable
  | Name "₀" | Name "₁"                   -- dup variable (first/second)
  | Integer                               -- number literal
  | "@" Name                              -- reference to definition
  | "*"                                   -- wildcard (erased)

  -- Lambda & Application
  | "λ" Params "." Term                   -- lambda: λx.body
  | "λ" "&" Name "." Term                 -- cloned lambda: λ&x.body
  | "λ" Name "&" Label "." Term           -- lambda with inline dup
  | Term "(" Args ")"                     -- application: f(x,y)

  -- Pattern Matching
  | "λ" "{" Cases "}"                     -- match lambda
  | "λ" "{" Term "}"                      -- use (strict unbox)

  -- Duplication & Superposition
  | "!" Name "&" Label "=" Term ";" Term  -- dup: !x&L=val;body
  | "!" Name "&" "(" Term ")" "=" Term ";" Term -- dynamic dup
  | "&" Label "{" Term "," Term "}"       -- superposition: &L{a,b}
  | "&" "(" Term ")" "{" Term "," Term "}" -- dynamic superposition
  | "&" "{" "}"                           -- erasure

  -- Let Bindings (sugar)
  | "!" Name "=" Term ";" Term            -- let: !x=val;body
  | "!" "&" Name "=" Term ";" Term        -- cloned let
  | "!!" Name "=" Term ";" Term           -- strict let

  -- Constructors
  | "#" Name "{" Args "}"                 -- constructor: #Pair{a,b}
  | "#" Name "{" "}"                      -- nullary: #ZER{} or #True{}

  -- Operators
  | Term Op Term                          -- binary: (a + b), (x == y)
  | Term "===" Term                       -- structural equality
  | Term ".&." Term                       -- short-circuit AND
  | Term ".|." Term                       -- short-circuit OR

  -- Special
  | "↑" Term                              -- priority (collapse order)
  | "&" Label "λ" Names "{" Term ";" Term "}"  -- fork
  | "[" Args "]"                          -- list: [a,b,c]
  | '"' Chars '"'                         -- string
  | "'" Char "'"                          -- char: 'x' -> #CHR{120}
  | Integer "n"                           -- nat: 2n -> #SUC{#SUC{#ZER}}

Cases    ::= (Pattern ":" Term ";")* [Default]
Pattern  ::= "#" Name | Integer | "[]" | "<>" | Integer "n" | Integer "n+"

Operators (low to high): || && == != < <= > >= << >> + - * / % ^
```

KEY CONCEPTS

AFFINE VARIABLES

Variables are **affine**: used at most once.

```
// WRONG: x used twice
@bad = λx. (x + x)

// RIGHT: use cloned binder
@square = λ&x. (x * x)
@main = @square(5)
//25
```

The `&` prefix tells the parser to auto-insert duplication nodes.

Common affinity errors:

1. Using a variable in multiple match branches:

```
// WRONG: x appears in both branches
@bad = λx. λ{
  #T: x
  #F: x
}

// RIGHT: match first, then bind x in each branch
@good = λ{
  #T: λx. x
  #F: λx. x
}
@main = @good(#T{}, 42)
//42
```

2. Using unnecessary clones:

```
// BAD: inline λ{...}(x) is poor style

@filter = λ&f. λ{
  []: []
  <>: λ&h. λ&t.
    λ{
      #T: h <> @filter(f, t)
      #F: @filter(f, t)
    }(f(h))
}

// GOOD: split into two functions

@keep = λ{
  #T: λh. λt. h <> t
  #F: λh. λt. t
}

@filter = λ&f. λ{
  []: []
  <>: λ&h. λt. @keep(f(h), h, @filter(f, t))
}
```

It is very important, on HVM, to avoid dups as much as possible. As such, it is considered a good style to format all top-level definitions as case-trees, rearranging arguments to avoid unecessary clones, and avoiding inline λ-match applications (such as `λ{...}(x)`).

SUPERPOSITION (SUP)

A SUP `&L{a,b}` represents two values in one location:

```
@main = (&X{10, 20} + 5)
//15
//25
```

When collapsed (`-C` flag), each branch prints separately.

LABEL MECHANICS

Labels control how DUPs interact with SUPs:

- **Same label**: DUP and SUP annihilate (pair extraction)
- **Different labels**: they commute (cross product)

To observe this, use explicit DUP syntax `!x&L = val; body` which binds x_0 and x_1:

```
// Same label: pairwise extraction
@main =
  !x&A = &A{1, 2};
  [x₀, x₁]
//[1,2]
```

```
// Different labels: the DUP stays inside each SUP branch
@main =
  !x&A = &B{1, 2};
  [x₀, x₁]
//[1,1]
//[2,2]
```

```
// Cross product from two SUPs with different labels (4 results)
@main = [&A{1, 2}, &B{10, 20}]
//[1,10]
//[1,20]
//[2,10]
//[2,20]
```

Prefer `λ&x` for general functions. Use explicit DUPs only when you need label control (pairwise extraction or explicit SUP/DUP interactions).

PATTERN MATCHING

Match on constructors with `λ{...}`:

```
@not = λ{
  #T: #F{}
  #F: #T{}
}
@main = @not(#T{})
//#F{}
```

```
@pred = λ{
  0n:  0n
  1n+: λp. p
}
@main = @pred(3n)
//2n
```

Match cases bind constructor fields as lambda arguments.

PEANO NATURALS

`Nn` is sugar for `#SUC{...#SUC{#ZER}...}`. Patterns: `0n:` matches `#ZER`, `1n+:` matches `#SUC` and binds predecessor:

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}
@main = @add(2n, 3n)
//5n
```

CHARACTERS

A char literal `'x'` desugars to `#CHR{120}` (the constructor `#CHR` wrapping the Unicode codepoint as a number). Escape sequences: `'\n'`, `'\t'`, `'\r'`, `'\0'`, `'\\'`, `'\''`, `'\"'`.

```
@main = 'A'
//'A'
```

To match on a character, match the `#CHR` constructor to extract the codepoint, then switch on the numeric value:

```
@is_a = λ{
  #CHR: λ{'A': #T; λn. #F}
}

@main = @is_a('A')
//#T{}
```

LISTS

Lists use the built-in constructors `#CON` and `#NIL` (all caps).

Sugar:

- `[]` desugars to `#NIL{}`
- `[a, b, c]` desugars to `#CON{a, #CON{b, #CON{c, #NIL{}}}}`
- `a <> b` desugars to `#CON{a, b}` (cons)

Pattern sugar:

- `[]:` matches `#NIL`
- `<>:` matches `#CON`

```
@len = λ{
  []: 0
  <>: λh. λt. (1 + @len(t))
}
@main = @len([1, 2, 3, 4, 5])
//5
```

STRINGS

Strings are lists of `#CHR`. `"hi"` desugars to:

#CON{#CHR{104}, #CON{#CHR{105}, #NIL{}}}

Backtick can be used instead of quotes.

```
@main = "hi"
//"hi"
```

Since strings are just lists, list functions work on them directly:

```
...
@main = @len("hello")
//5
```

NUMERIC PATTERNS

Switch on machine ints with `0:`, `1:`, etc., and `λn.` for default:

```
@fib = λ{
  0: 0
  1: 1
  λ&n. (@fib((n - 1)) + @fib((n - 2)))
}
@main = @fib(10)
//55
```

FORK NOTATION

`&Lλx,y{A;B}` is sugar for paired branching. A uses x_0, y_0; B uses x_1, y_1:

```
@main = (&Lλx,y{(x + y); (x * y)})(3, 4)
//7
//12
```

This desugars to:

```
@main =
  !x&L = 3;
  !y&L = 4;
  &L{(x₀ + y₀), (x₁ * y₁)}
//7
//12
```

UNSCOPED LAMBDAS

Unscoped lambdas allow a variable to be used outside its binder's lexical scope. You can write them directly as:

```
λ$x. body
```

This is sugar for:

```
! f = λ x ; body
```

This declares in `body`:

- `f = λy. λ$x. y` — a function that wraps its argument in an unscoped lambda
- `x = $x` — an unscoped variable usable anywhere

So `λ$x.body` is equivalent to constructing it as `f(body)`:

```
@main = (λ$x. (x + 1))(10)
//11
```

Applying the unscoped lambda to `10` binds `x` to `10`, yielding `11`. (Equivalent sugar: `! f = λ x ; f(x + 1)(10)`.)

Another example using the unscoped variable in a pair:

```
@main = ! f = λ v ; #P{f(1, 2), v}
//#P{1,2}
```

Here `f(1, 2)` is `f(1)(2)` which produces `λ$v.1` then applies it to `2`, binding `v` to `2`. The pair contains `1` and the captured value `2`.

This is useful for continuation-passing or "exporting" a binding out of a nested context.

EXAMPLES

FACTORIAL WITH PEANO NATURALS

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}

@mul = λ{
  0n:  λb. 0n
  1n+: λa. λ&b. @add(b, @mul(a, b))
}

@fac = λ{
  0n:  1n
  1n+: λ&p. @mul(1n+p, @fac(p))
}

@main = @fac(3n)
//6n
```

LIST MAP

```
@map = λ&f. λ{
  []: []
  <>: λh. λt. (f(h) <> @map(f, t))
}
@main = @map(λx. (x * 2), [1, 2, 3])
//[2,4,6]
```

ENUMERATING BINARY STRINGS

Superpositions can enumerate multiple results; each branch prints separately in collapse mode:

```
@bits = λ{
  0n:  #E{}
  1n+: λ&p.
    &B{#O{@bits(p)}, #I{@bits(p)}}
}
@main = @bits(2n)
//#O{#O{#E{}}}
//#O{#I{#E{}}}
//#I{#O{#E{}}}
//#I{#I{#E{}}}
```

CHURCH NUMERALS

Standard lambda encoding with a cloned binder for `s`:

```
@c2   = λ&s. λz. s(s(z))
@cmul = λa. λb. λs. λz. a(b(s), z)
@main = @cmul(@c2, @c2)
//λa.λb.a(a(a(a(b))))
```

SOLVING EQUATIONS VIA ENUMERATION

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}

@eq = λ{
  0n:  λ{
    0n:  #T{}
    1n+: λp. #F{}
  }
  1n+: λa. λ{
    0n:  #F{}
    1n+: λb. @eq(a, b)
  }
}

@if = λ{
  #T: λt. λf. t
  #F: λt. λf. f
}

// All naturals: 0, 1, 2, ...
@X = &N{0n, 1n+@X}

// Solve X + 2 = 4
@main = @if(@eq(@add(@X, 2n), 4n), @X, &{})
//2n
```

FILTER

```
@even = λ{
  0: #T{}
  1: #F{}
  λp. @even((p - 2))
}

@filter = λ&f. λ{
  []: []
  <>: λ&h. λt. @keep(f(h), h, @filter(f, t))
}

@keep = λ{
  #T: λh. λr. (h <> r)
  #F: λh. λr. r
}

@main = @filter(@even, [0, 1, 2, 3, 4, 5, 6])
//[0,2,4,6]
```

COMMON PATTERNS

- `λ&x.` : Cloned binder (auto-dup)
- `!x&L = v; body` : Explicit dup with label L
- `x₀` , `x₁` : Dup variable branches
- `&L{a,b}` : Superposition
- `&{}` : Erasure (discard value)
- `a <> b` : List cons
- `@name` : Reference to definition

Constructor match:

```
λ{
  #C: ...
}
```

Peano natural match:

```
λ{
  0n:  ...
  1n+: λp. ...
}
```

Numeric switch (machine ints):

```
λ{
  0:  ...
  1:  ...
  λn. ...
}
```

GOTCHAS

1. **No whitespace application**: use `f(x)` not `f x`
2. **Variables are affine**: use `&` for multiple uses
3. **Subscripts required**: dup-bound vars need `₀` or `₁`
4. **Labels matter**: same labels annihilate, different multiply branches
5. **Constructor syntax**: always `#Name{...}`, even nullary `#ZER{}`
6. **Pattern arms are lambdas**: `1n+: λpred. body` "binds" the field
7. **No inline match application**: avoid writting `λ{...}(x)`

REDUCTION MODES

- **WNF**: weak normal form (head only)
- **SNF**: strong normal form (full, keeps SUPs/DUPs)
- **CNF**: collapsed normal form (enumerate as pure lambda terms)

Use `-C` flag for collapse mode, omit for normalize mode.

H.41 · bend-docs/openspec/changes/bend-docs-site/research/victor/hoc-history.md

Higher Order Company: Complete Historical Overview - WIP

URL: https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1 (https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1)

FILE: HOC_HISTORICAL_OVERVIEW.MD

Higher-Order Company: Complete Historical Overview

This document is a complete historical overview of the Higher Order Company. If you want to learn anything about our background, a good way to do so is to feed this Gist into an AI (like Sonnet-3.5) and ask it any question!

MY SEARCH FOR A PERFECT LANGUAGE

It all started around 2015. I was an ambitious 21-year-old CS student who, somehow, had been programming for the last 10 years, and I had a clear goal:

> I want to become the greatest programmer alive

I wished for the ability to write a whole game engine over a weekend, and C wouldn't give me that power. This led me to survey all programming languages that existed, looking for one that would maximize my productivity. After learning dozens of common and esoteric options, I was convinced the functional paradigm was the way forward. Soon after the initial enthusiasm, I came to the realization that FP languages struggled with performance and practical issues, so I decided to build my own. And that was the beginning of a long journey.

THE LAMBDA CALCULUS

The Lambda Calculus the elegant model of computation behind functional programming languages. It, to my eyes, seemed canonical, in a very fundamental sense. Because of that, I expected it to be "perfect", just like the Euler formula, or Tau constant. So, it was surprising to me to learn that all λ-Calculus evaluators - including the purest languages in the domain, like Haskell and Lisp - were *sub-optimal*. Why would that be the case?

Trying to implement my own functional language, I quickly realized why: **substitution isn't atomic**. Instead, it is a quite convoluted operation that clones arbitrarily large structures, and performs complex renamings. It isn't lightweight, in the way I expect the most fundamental operation of the most fundamental model to be. But it HAD to be!

LAMPING'S OPTIMAL ALGORITHM

I then started looking for the fastest way to evaluate λ-terms, and someone linked me to Lamping's 1990 algorithm `(https://dl.acm.org/doi/10.1145/96709.96711)`. Turns out there IS a lightweight way to perform substitution, but it involves a new, graphical representation, rather than the syntax-based trees we're used to. Yet, in that new point of view, substitution becomes atomic, fast and absent of renaming schemes. It felt *canonical*.

I quickly implemented that algorithm as a small JavaScript project called Optlam `(https://github.com/VictorTaelin/optlam)`, and tested it on a bunch of terms that, on my previous implementations, took a long time to compute. And, in these cases, it was... fast. No, not just fast. It was miraculously fast. Impossibly so. I could actually not believe it, so I made this `(https://stackoverflow.com/questions/31707614/why-are-%CE%BB-calculus-optimal-evaluators-able-to-compute-big-modular-exponentiation)` Stack Overflow post. Turns out my implementation was correct, and the algorithm was, indeed, that good. So... **why the hell was nobody using it** `(https://www.reddit.com/r/haskell/comments/2zqtfk/why_isnt_anyone_talking_about_optimal_lambda/)`

After quick research, I identified 2 key reasons:

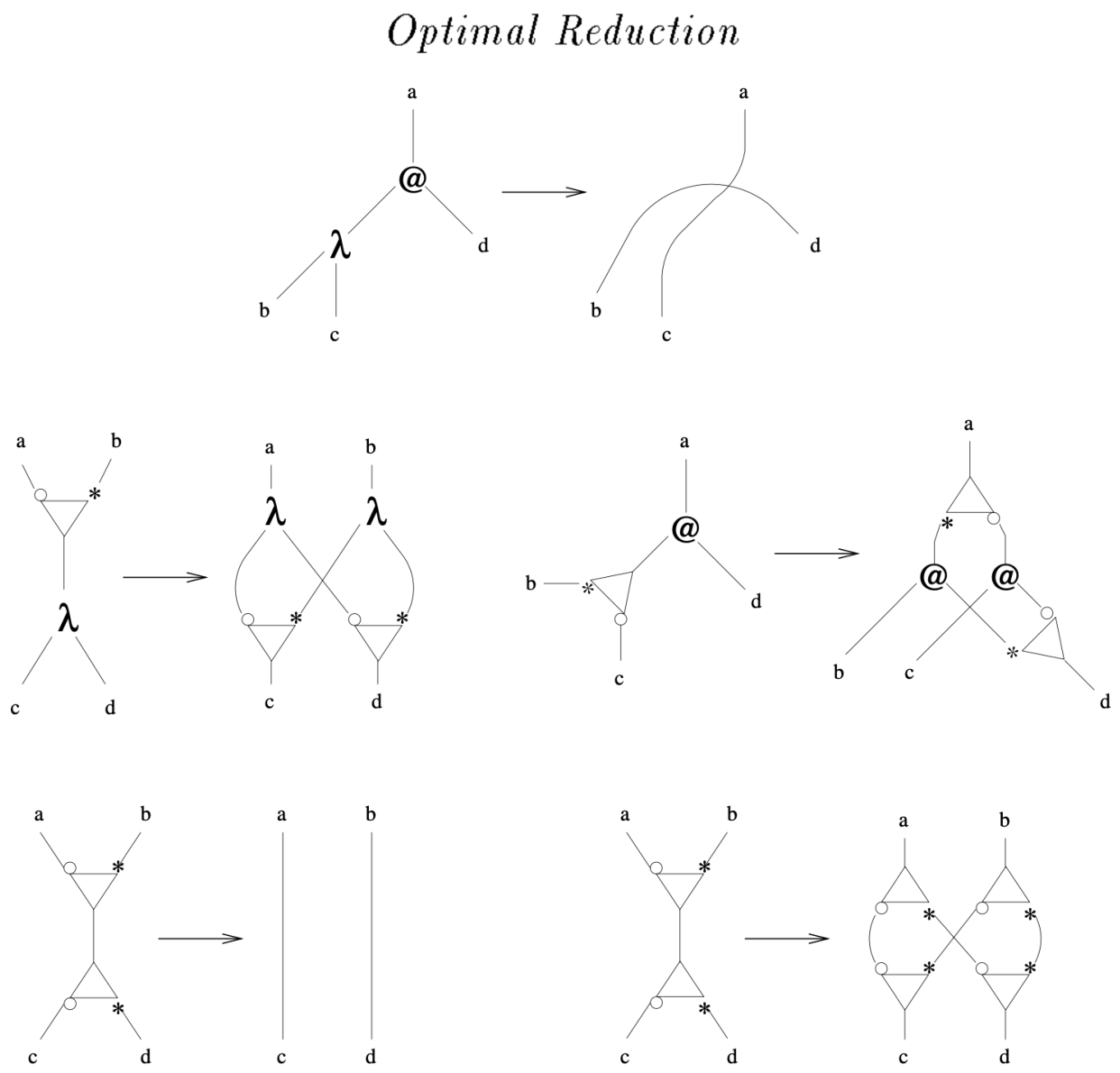
1. A *negative result* that was misinterpreted by researchers back then
2. The theoretical optimality didn't translate to real-world performance

A MISINTERPRETED THEORETICAL RESULT

To explain the negative result, a little bit of context is needed. Lamping's algorithm is actually divided into two parts: the **Abstract Algorithm** and the **Oracle**.

1. THE ABSTRACT ALGORITHM

The first part breaks down **substitution** into two operations: **application** and **cloning**. It looks like this:



These simple graph-rewrite rules form the core algorithm to evaluate λ-terms! `λ` nodes represent lambdas, `@` nodes represent applications, and the triangles represent cloning. As the "optimal" word implies, there is no way to reduce lambda terms faster than this approach, because it ensures no work is needlessly duplicated (which is the main issue with tree-based evaluators). In other words...

If:

- Substitution is now an atomic, O(1) operation.
- The number of substitutions to compute a term is the minimum possible.
- Structural cloning is also kept at the bare theoretical minimum.

Then:

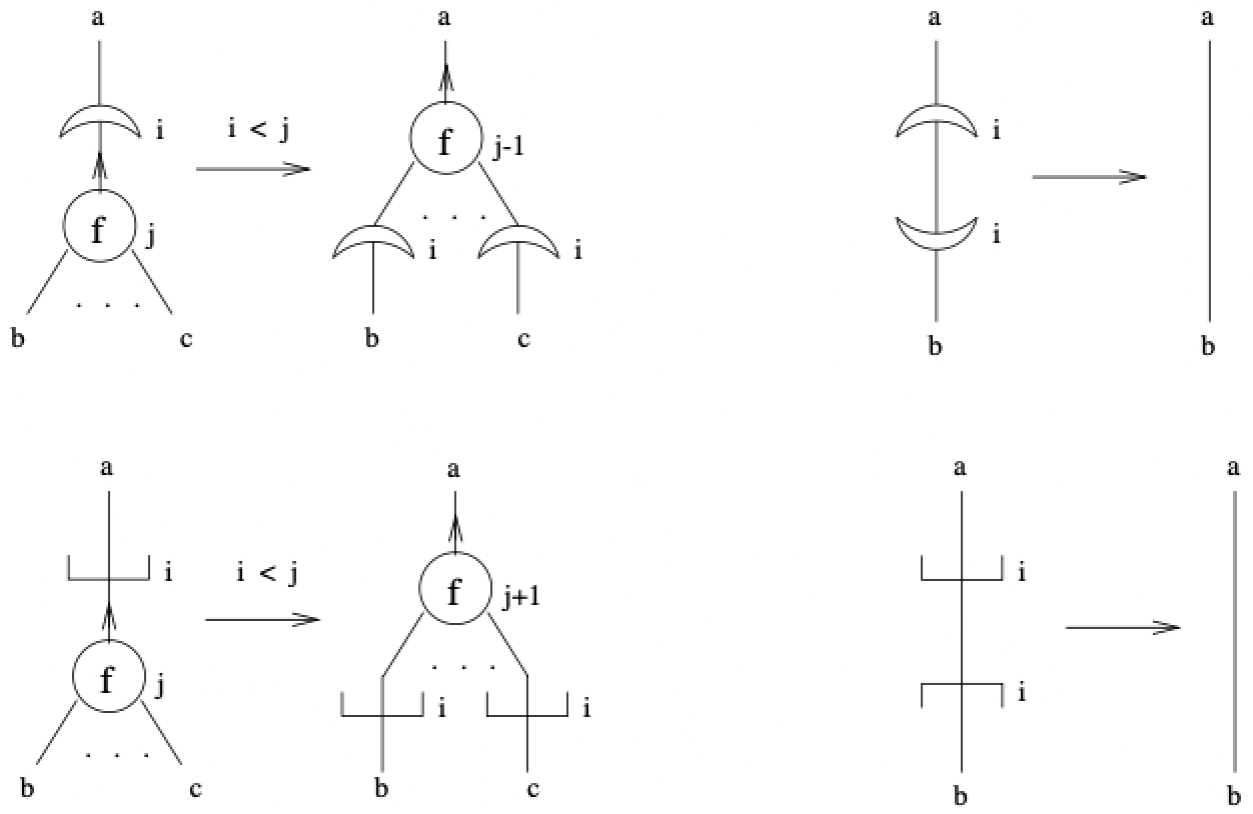
- The Abstract Algorithm is **optimal**!

That conclusion is sensible, and it implied, to many, that this simple algorithm could be used to evaluate λ-terms efficiently and, thus, serve as the backbone of functional programming languages. Amazing! So, what was the problem?

2. THE ORACLE

Turns out the Abstract Algorithm, alone, isn't capable of evaluating all λ-terms correctly. There are some terms for which it will, simply, produce an incorrect result. But Lamping himself provided a solution: the **Oracle**. It is a set of additional graph nodes that, in simple terms, do some scope tracking wizardry that makes the algorithm provably correct. Here is

how it looks:



The addition of these moon-bracket-looking nodes allows Lamping's algorithm to evaluate arbitrary lambda terms to normal form correctly and soundly. But they also introduce a problem: the nodes accumulate, greatly increasing the overall cost... sometimes, exponentially. This led to the perception that the "optimal" algorithm, perhaps, wasn't actually optimal: after all, while it minimizes the amount of substitutions, it seemed like the cost was just being pushed into the Oracle.

For some time, researchers were busy trying to better understand that aspect. Many alternatives to Lamping's algorithm were developed. They all kept the same "abstract" fragment, but proposed a different oracle. The overall hope was that a "better oracle" would get rid of that little problem, proving, once and for all, that the optimal algorithm is, indeed, optimal.

3. LAMPING'S ALGORITHM IS NOT ACTUALLY OPTIMAL?

Sadly, that hope was shattered in 1996, when Asperti, Lawall and Mairson published a paper `(https://dl.acm.org/doi/10.1145/232627.232639)` proving that the cost of evaluating a λ-term is NOT bounded by the total amount of substitutions. Now, let's think.

If:

- Lamping's algorithm minimizes the "substitution count".

Yet...

- "Substitution count" does NOT represent the "actual total cost".

Then:

- Lamping's algorithm **may** not be optimal in "actual total cost"!

And also:

- The Oracle was, indeed, "hiding" a very significant cost.

This is all true, and, as you can imagine, this realization shattered the hype around optimal evaluation. But, here's the catch: this result doesn't say anything about Lamping's algorithm. It just provides an insight about the Lambda Calculus itself: that minimizing substitution isn't enough for optimality, as everyone believed. In other words, it does not prove

Lamping's algorithm isn't optimal; it just shows that *we don't know*. But it could still be. So, is it?

4. LAMPING'S ALGORITHM IS ACTUALLY OPTIMAL!

Later on, Asperti himself discovered another fact: the *extra cost* (presumably pushed to the Oracle) is actually already present in the abstract fragment, because the amount of *cloning* that a λ-term may present is not bounded by the amount of *substitutions* either! But, wait...

If:

- The "extra cost" is actually caused by "cloning", not by the Oracle
- Lamping's algorithm, indeed, minimizes cloning

Then:

- Lamping's algorithm is *actually* optimal!

That's the sensible, correct take. To recap:

1. Lamping devised an optimal algorithm for λ-Calculus evaluation, in terms of "substitution count"
2. People believed the algorithm was, indeed, optimal in terms of "total work done"
3. Asperti discovered an "extra cost", implying "substitution count" isn't enough to define "optimality"
4. People concluded the algorithm may, thus, not actually be optimal, since it minimizes a now-irrelevant metric
5. Asperti has shown that the "extra cost" is actually attributable to "cloning"
6. Lamping's algorithm also minimizes "cloning", so, it is optimal on that "extra cost" too!

So, essentially, as far as theory is concerned, Lamping's algorithm IS, indeed, an optimal λ-Calculus evaluator, w.r.t all investigated measures. Sadly, the misconceptual reception of the original "negative result" - which was not even related to Lamping's algorithm - was so strong that it distanced many researchers from this field. Years later (in a response to a question posted on Reddit!) Asperti himself published a follow-up paper `(https://arxiv.org/pdf/1701.04240)` trying to clear up that misconception, but it was already too late to fix the accidental damage.

FROM THEORETICAL TO PRACTICAL PERFORMANCE

Even if Lamping's algorithm is, indeed, optimal in theory, this still leaves an open question: is it optimal **in practice**? Both notions aren't necessarily equivalent. For example, the theoretically optimal multiplication algorithm is `O(N*log(N))`. Yet, it has constant costs so high that, for most practical purposes, it is worth using some known sub-optimal algorithms instead. So, is that the case for Lamping's algorithm?

Asperti himself attempted to answer that question by implementing The Bologna Optimal Higher-order Machine `(https://www.cambridge.org/core/services/aop-cambridge-core/content/view/1F2763B0F931680F9B15BDC750BEB343/S0956796800001994a.pdf/bologna_optimal_higherorder_machine.pdf)`. This work was the main inspiration behind HVM, and the first serious attempt at implementing Lamping's algorithm efficiently in practice. It is a marvelous piece of software, and a big achievement. Yet, it had two key problems:

1. Memory consumption was way too high
2. Total interaction count was way too high

Due to both of these problems, BOHM couldn't outperform classical evaluators such as Haskell's STG, or Scheme's closures, in practice. The benchmarks were brutally clear in that regard. So, how could we improve it?

1. ABSAL: OPTIMIZING OPTIMAL REDUCTION

Looking at BOHM's implementation, all my depraved engineer mind could think of was this massive, overlooked optimization opportunity:

> Implement just the Abstract Algorithm!

Remember Lamping's algorithm had two parts? The second one, the Oracle, is necessary to ensure sound λ-Calculus evaluation **in all cases**. Yet, it adds a 10x overhead, even in cases where it isn't needed at all! So, my question was: when is it *actually* needed? I'm an Engineer, not a Scientist. If I see a 10x speedup opportunity, hell I'm not discounting it.

To my surprise, this (https://dl.acm.org/doi/abs/10.1145/1131313.1131315) paper identified that there is an entire class of λ-terms on which the oracle isn't needed at all; specifically, terms typeable in Elementary Affine Logic. That sounded perfect, so, I quickly reimplemented the algorithm in a more polished repository, Absal (https://github.com/VictorTaelin/abstract-algorithm), with the goal of exploring this subset. And, over the next few years, I implemented basically ALL algorithms you can think of, in this little subset of the untyped lambda calculus. And I really mean it. It was months and months non-stop implementing all sorts of algorithms you can think of, in raw λ-terms. At a point, I was visiting my own mother in the hospital - a very dark time - and, in the waiting room, there I was, with my little notebook, drafting lambda terms. Back then, I had no big intentions with all of this. Just curiosity and, in a way, escapism.

So, after this long experience, I came up with one fundamental conclusion:

> The Abstract Algorithm is all you need

And that is just true: every program can be computed by the Abstract Algorithm (it is Turing Complete!), every λ-term can be "indirectly" computed (by interpretation), and specific λ-terms that aren't "directly compatible" can easily changed to compatible terms, or rejected by a type-system, just like Rust and Haskell reject many λ-terms too. It is a fact that most practical languages do not support the full λ-Calculus either, yet, for some reason, people take this as a negative point exclusive to the Abstract Algorithm. It isn't! And, if I could have a 10x speedup by just rejecting some defective λ-terms, hell I'd have it!

2. INTERACTION COMBINATORS

After spending long enough time with the Abstract Algorithm, some quite eerie thoughts started occurring in my mind. Remember: the entire reason I got into this whole thing, to begin with, was because the Lambda Calculus seemed, to me, like "the one true Model of Computation". But, now, I had a direct counterpoint to this hypothesis. After all, if the λ-Calculus was so perfect... why would it have "defective terms" at all? Also, why does Lamping's algorithm have two parts... one that feels beautiful/canonical, and another that feels engineered/arbitrary?

I know that's not a very technical argument, but, still, it seemed very real to me. This prompted me to dig deeper into the nature of Lamping's Algorithm itself, which - and I don't remember exactly how - eventually got me to the most important paper of my life: Lafont's 1997 Interaction Combinator.

That paper was a distillation of a whole new model of computation, called Interaction Nets - and it was kind of framed as a "perfect" alternative to the Lambda Calculus and Turing Machines. A new model that was optimal, in a deep theoretical sense. Specifically, Lafont proved that his model was capable of emulating any other without increasing complexity; which is a property stronger than mere Turing completeness. Moreover, it was massively parallel, which, back then, wasn't something I was even looking for, but, hey, that's a nice bonus, I guess?

Here is how it looks like:

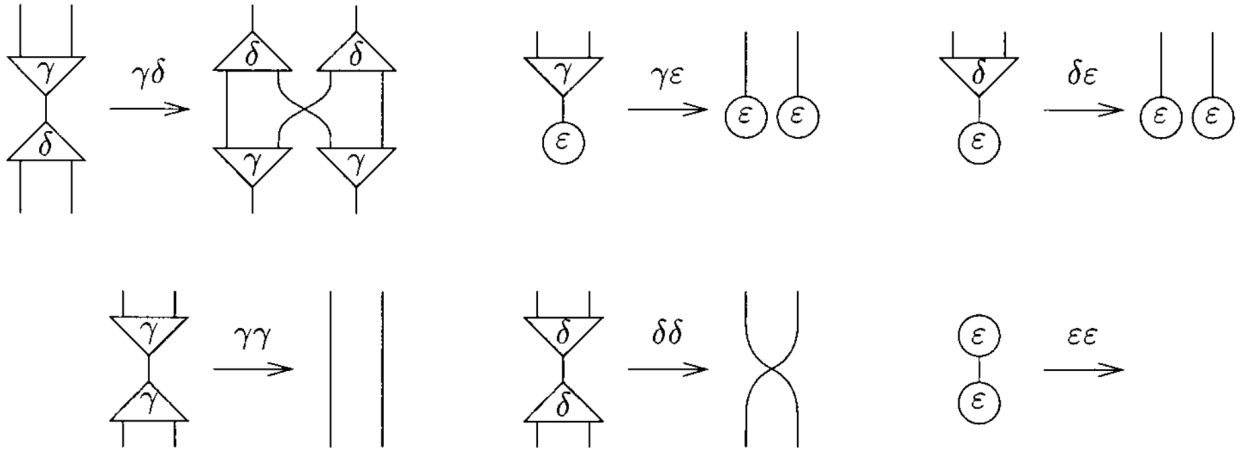


FIG. 2. Interaction rules for the combinators.

Can you see it? That's right: if you just draw λ's as triangles, and @'s as triangles... then, Interaction Combinators are **exactly** the same thing as Lamping's Abstract Algorithm. Yves's work isn't just a universal distillation of Interaction Nets. It is also the heart of the Lambda Calculus. AND it also subsumes Turing Machines (which can be seen as just 1D

Interaction Combinators) and Cellular Automata (which can be seen as just grid-aligned Interaction Combinators).

As you can probably already imagine - I now hold the very strong conviction that Interaction Combinators are THE optimal model of computation, and that everything else is just a bloated version of that tiny, ultra-efficient system; just like the Lambda Calculus was. And, if that's the case... that implies every programming language, every runtime, every processor, could be made faster - or, rather, less slow - by removing bloat through the lens of Interaction Combinators.

As usual, I was very enthusiastic about this new model, and asked on Reddit why the hell nobody was talking about it (https://www.reddit.com/r/haskell/comments/568gtk/why_the_overall_lack_of_interest_in_interaction/).

3. INTERACTION COMBINATOR ∩ LAMBDA CALCULUS = INTERACTION CALCULUS

At that point, I was convinced ICs are the ultimate model of computation, independent of the λ-Calculus, and, as such, I started asking the obvious questions: can it do things that the λ-Calculus fundamentally cannot? There are IC nets that don't directly correspond to λ-terms. What do they do? What can be done with them? That was so interesting, I started abandoning the λ-Calculus and playing with ICs directly. There was a problem, though: visualizing ICs on computers was quite hard, and, even though I implemented a bunch of renderers, ultimately, drawing was still the easiest way to explore the system. At a point, my floor was covered by mountains of paper sheets (https://imgur.com/a/Zb0W7FQ) with hundreds of IC nets drawn manually.

The key insight is that, instead of drawing "triangles", you can actually write "CON nodes" as lambdas and application, as long as you accept the following departion from the λ-Calculus:

> Lambdas have global scopes

That is, the variable bound by a lambda can, as insane as that may seem, occur outside of its body. Once you accept that, then you can actually draw Interaction Combinators in a more familiar notation. From that point of view, the Interaction Calculus (https://github.com/VictorTaelin/Interaction-Calculus) was born. Initially, this was meant to be merely a textual syntax for exploring the graphs, allowing me to write and debug its nets without having to make drawings, and it really worked well for that purpose. Nowadays, I think there is something more fundamental to it.

> **NOTE: I went on a tangent here. Feel free to skip to the next part.**

Several concepts that are "strange" in conventional functional languages suddenly gain a new light once you think about them in terms of "lambdas with global scopes". For example, it is impossible to write a monadic function in Haskell that returns a HOAS λ-Term directly:

```
-- Here, $x is a "globally bound variable"
make_lam : Context -> Term
make_lam ctx = do
  ...
  body <- make_lam ($x : ctx)
  ...
  return $ Lam (\$x -> body)
```

It is impossible to do so in one pass, because the solution requires the body to be constructed before the lambda, but the lambda is needed to construct the body. Similarly, fundamental features like call-cc, O(1) double-ended queues, mutable references(!) aren't expressive in the λ-Calculus, but can be done easily in the Interaction Calculus. Also, in a massivelly parallel setup, the λ-Calculus alone isn't sufficient to represent all concurrent algorithms; global λ's are needed to, say, fully emulate Erlang's work scheduler. (Amb nodes are too, but I won't get into that point.)

Also, just like lambdas and applications are duals, the dual to a duplication node was a superposition node, which turned out to be a very intuitive and massively impactful feature that I now apply intuitively to solve problems. For example, a simple SAT (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615) brute-forcer gains an asymptotical speedup in certain cases, if you just use superpositions, instead of loops, to search through the space. This concept is now so familiar to me that I use it naturally on Bend, to the point I actually miss it when I'm coding in Haskell. I've been exploring it extensively on unpublished AI research, and its presence is absolutely key in some parts of the algorithms. Or, as I see it, its absence introduces tons of unecessary slowdowns!

Later on, when thinking about a type system for Interaction Combinators (https://github.com/victortaelin/interaction-calculus-of-constructions), Franchu (on HOC's Discord) realized that a very elegant primitive, that we now call "The Bridge", could be used to derive the core axioms of type theory (such as Pi types and Sigma types). In other words, Π and Σ can be further broken down, and I strongly believe that may have strong applications in the study of mathematical foundations. For example, Bridges are sufficient to express Aaron Stump's self types (https://homepage.divms.uiowa.edu/~astump/papers/fu-stump-rta-tlca-14.pdf), which, in turn, can be used to represent dependent λ-encodings, which allow small core theories like the Calculus of Constructions to express arbitrary inductive families (and even quotients) without a complex native datatype system (as currently implemented in all proof assistants).

Anyway, I'm going on a tangent here, so, back on track.

3. FORMALITY-NET: THE FIRST REAL ATTEMPT

Fast forward a few years, I was working at the Ethereum Foundation, just hired as the new member of the Mist team: a browser designed as the entry point of the Ethereum network. Back then, the network was in a complete state of immaturity and chaos, and, as you can already imagine, ALL issues that occurred were obviously Mist's fault. People would literally have dogs eat their private keys, and blame Mist for it. But I was fine with that; what terrified me was when I learned about how Solidity worked, and how it was compiled to EVM. That language was being used to upload "unstoppable" contracts that held millions of dollars. Except they stopped working all the time, due to bugs in the code. That was completely insane to me, so, I had to do something.

During these times, I started heavily studying the one silver bullet for software bugs: mathematical proofs. As far as I could tell, it was the only technology capable of ensuring that given software is 100% correct on its first version - which is exactly what Ethereum needed. I made a lot of internal pressure inside EF, until they eventually gave up and just let me lead my own team to implement a proof language for it. And, just like that, the Formality language was born! But I also needed to compile it to something... and that thing had to be efficient enough to run inside the blockchain, an environment where every operation costs a lot of money. Idris had already attempted, and it was unviably expensive. Thus, as you can guess... I used Lamping's algorithm, and attempted to make it as fast as I could.

Formality's runtime was called FM-Net (https://github.com/VictorTaelin/Formality-Net-legacy). It was now written in C, and optimized as much as I possibly could, using that idea of dispensing the Oracle entirely. Sadly, even with that "10x" speedup... it was still not so fast in practice. Haskell was still much faster, and there wasn't much I could do about it. Perhaps I was wrong, after all...

4. HVM: AN EFFICIENT IMPLEMENTATION

After Formality failed to meet the performance I expected it to, I just gave up on the problem for a while. During my time at the Ethereum Foundation, I converted most of my salaries to ETH, which appreciated enough to give me financial independence and allow me to retire early when I was 25. I proceeded to spend the next few years funding and working on gamedev projects that gave me joy, playing games, teaching friends how to program - but I always had this specific problem in the back of my mind.

One day, it randomly occurred to me an insight that should probably have been obvious: I could use the Interaction Calculus as a memory format to represent Interaction Combinator nodes with reduced memory footprint, by omitting pointers on lambda, application and superposition nodes. Moreover, by including tags directly on pointers (rather than on nodes), I can further cut it down. That meant a lambda could be represented with 2x less memory, from 256 to 128 bits (in a 64-bit architecture).

This naturally and almost simultaneously led me to realize another huge (now obvious) optimization opportunity: rather than using fan nodes to clone global definitions, I can actually do that with a specialized "deref" operation, in a single memcpy. And, while we're there, why not just go ahead and add native constructors? Then, we can actually dispatch pattern-matching equations with a switch statement, just like Haskell works! In essence, that setup would give me the same memory/compute footprint of GHC (in low-order computations), yet the exact same asymptotics of BOHM (in higher-order computations).

There was no reason it wouldn't work, so, I stopped everything I was doing, locked myself in a room for a few days, and, just like that, the first version of HVM was born, written 100% in Rust. And, to my actual surprise, on my very first benchmarks, it was computing some real-world algorithms at about 30% of the performance of Haskell GHC. Considering that all previous attempts were orders of magnitude slower, that, to me, was a clear indication that I finally got the right architecture.

But, wait... so, I had 1/3 the speed of Haskell, and 8 cores on my computer. And I know Interaction Combinators, as a paradigm, are inherently parallel. So... could I...? Yes, I could! A few parallelism annotations later and HVM was now using all the cores of my computer, and it actually surpassed GHC in real benchmarks. Okay, it wasn't THAT simple, and, unfortunately, the original parallelism on HVM1 had major bugs that I couldn't fix. But, hey, for some cases it did work, and surpassing Haskell in some cases was not something I was expecting ICs to do anytime soon.

HIGHER ORDER COMPANY

So, let's recap what the HVM represented, to me:

1. A new runtime capable of computing the functional paradigm as fast as Haskell/GHC in all low-order programs (because it is low-footprint in practice), yet, exponentially faster in some higher-order program (because it is optimal in theory).
2. A new paradigm capable of automatic massive parallelism, maximizing Amdahl's law (https://en.wikipedia.org/wiki/Amdahl%27s_law) (because interactions are small atomic operations, which minimize the longest sequential fragment), and, thus, potentially scaling in performance with near-ideal speedup up to millions of concurrent threads.
3. New powerful primitives (global lambdas, superpositions, bridges) that could, one day, bring advancements in number theory, type theory and foundations.

Of course, perhaps, I was wrong. Perhaps all these things were just fruits of my imagination, and the Interaction Combinator paradigm could never bring any real progress to humanity whatsoever. But what if it could? At that point, this whole thing was kinda on my hands? If I just stopped working on it, nobody would. It is as if Newton had invented the Calculus, sent to me in a letter and disappeared. I felt a huge sense of responsibility in bringing this technology to the world, because I believed great things could come out of it. So, I had to do something, and take it to the next level. But how?

TAKING INTERACTION COMBINATORS SERIOUSLY

Progress demands funding. Initially, I thought I could seek grants and get closer to academia. But, honestly, I'm terrified of the economics of academia. I don't think the incentives are right and I don't think the money goes to where it has to be. I, sadly, don't trust the process. (Should I?)

My second plan was to create an alternative to Ethereum, called Kindelia. This seemed like such a low-hanging fruit: we could replace EVM by the HVM, have massively increased layer-1 throughput, greatly improve on the security problem (because HVM runs proof languages natively!) and, once the project grows to the top 10 of CoinMarketCap - which I'm fairly sure it would - we could fund HVM research out of the appreciation of the token, alone. And I really think that could work, so, we spent a few months researching the idea.

Yet, Kindelia is such a cool concept, that I'd love to see it existing as a core component of the internet, like Unix and DNS. A fair network, just like Bitcoin. Not a semi-private chain, with a team behind it taking a huge chunk of the total supply. Because that sucks. Oh, wait, that owner would be us? No, that would still suck. I love Kindelia, and I want it to be shipped on its best form. And its best form doesn't include my keys on the genesis block. There are other ways to fund a project.

The third plan was to just create a normal business, make my own profit out of the value of the technology, sustain its research from that and gracefuly give parts of it to the world - as much as it would be viable to, without compromising the business's survival. That made sense to me, because, if the technology was truly capable of improving upon so many areas of computing, then, it should not be hard to extract business opportunities from it, right? And if it was not, well, then better it fail before it gets too big! Win-win. From that point of view, the Higher Order Company was founded, with the goal of:

1. Further developing the "Interactional Paradigm" and HVM technology
2. Using private parts of this development to create profitable products
3. Launching parts of this development to the public (ex: HVM, Bend, Kindelia)

Now, it needed funds.

THE SEED ROUND

Of course, nobody makes money from just selling runtimes and programming languages, even when these are already massively used and deployed (Unity 3D kinda tried, and we know how that went). So, it was, obviously, really really hard to come up with a reasonable business plan for a prototypal runtime that barely managed to outperform public solutions in several cases. So, what now?

For our Seed Round, we made a highly technical pitch deck, with this clear message stamped on it: ICs are a core fundamental technology that could, one day, result in highly valuable products. But, for now, it still needs a lot of development and polish, as it would be immature and unfruitiful to attempt to create any product given the current state of things. So, the plan was to mature the tech, develop a correct version of HVM's mass parallelizer, and then choose one direction to focus, with Kindelia, Kind and Bend proposed as possible paths.

To be honest, I assumed - and kinda hoped, I guess? - nobody would buy that pitch. There was still so much to do in the techn real before we could have a proper business plan. And, initially, it did feel like that. Now, though, I realize the reception was actually much better than I expected. In about 1 week of meetings (which felt infinite to me, but actually isn't) we closed our $5m round. Sadly, one of the investors pulled out, bringing it down to $4m. But that was still enough to start, so, we did.

THE SEED STAGE

Unfortunatelly, our seed stage wasn't as productive as I wish it would. As you can probably imagine, neither I or my friends are very experienced with running a business, hiring and managing people. We spent A LOT of money on professionals that weren't productive. And, in most cases, it wasn't even their fault, as we didn't provide the right conditions for progress to take place. Ultimately, most of the results we got during this stage still came from the work of a few people.

Fortunatelly, even though the $ efficiency was not stellar, the results were. During this period, we delivered on the promises of the seed pitch, with Bend and its ecosystem chosen as the direction to focus, AND we pushed some other directions that, in my view, are promising. We only used ~60% of the funds so far, so, we still have a ~2 year runway. Key milestones:

1. Deeply researched how to make HVM's automatic parallelism **correct** (https://github.com/HigherOrderCO/HVM/blob/main/paper/PAPER.pdf), to the point we now have a very good understanding of the problem, and a collection of simple, elegant algorithms that solve the issue for good.
2. Implemented HVM2 (https://github.com/higherorderco/hvm,) which, based on these parallel primitives, can now scale up to thousands (and, potentially, millions or billions) of threads, with near ideal speedup. Running the full λ-calculus on GPUs was a huge milestone.
3. Developed and shipped Bend (https://github.com/higherorderco/hvm), an ultra-simplified, Python-like/Haskell-like language that runs natively on GPUs, while needing af fraction of the expertise necessary to develop parallel algorithms on C or CUDA.
4. Further developed the theoretical aspects, and better understood fundamental primitives like global lambdas, superpositions (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615), bridges (https://github.com/victortaelin/interaction-calculus-of-constructions), that I believe could have profound impact on algorithm design, type theory and AI.
5. Progressed on some other fronts like the Kind (https://github.com/higherorderco/kind) proof assistant and Kindelia (https://github.com/kindelia/Kindelia), which we plan to release somewhere during the Series A stage.

The main highlight is, of course, the correctness of HVM's new parallelizer. Yet, Bend's reception surprised us. Before the announcement, we joked it would barely reach 100 likes on Twitter, because my technical posts usually flop, while the meme/AI ones tend to viralize. The plan was to make a silent release, and focus on organically growing a community later this year, once Bend had a stdlib and HVM supported more targets. Yet, the announcement somehow got 13K likes on Twitter (https://x.com/VictorTaelin/status/1791213162525524076), 25k+ stars on GitHub, and was featured on some major YouTube channels. In a way, that made me realize how many people care about my work and want us to succeed, which gives me determination. In another, such popularity in a moment where the language is still not so mature may not give the best perception, but that's something we'll have to keep in mind and work with.

H.42 · bend-docs/openspec/changes/bend-docs-site/research/victor/hoc-optimal-computer.md

Higher-Order Company: Towards an Optimal Computer

URL: https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b)

FILE: TOWARDS_AN_OPTIMAL_COMPUTER.MD

Higher-Order Company: Towards an Optimal Computer ==========

WHAT IS THE TRUE NATURE OF COMPUTATION?

A hundred years ago, humanity answered that very question, twice. In 1936, Alan invented the Turing Machine, which, highly inspired by the mechanical trend of the 20th century, distillated the common components of early computers into a single universal machine that, despite its simplicity, was capable of performing every computation conceivable. From simple numerical calculations to entire operating systems, this small machine could compute anything. Thanks to

its elegance and simplicity, the Turing Machine became the most popular model of computation, and served as the main inspiration behind every modern processor and programming language. C, Fortran, Java, Python are languages based on a procedural mindset, which is highly inspired by Turing's invention.

Yet, the Turing Machine wasn't the only model of computation that humanity invented. Albeit less known, also in 1936, and in a completely independent way, Alonzo Church invented the Lambda Calculus, which distillated the common components - not of machines, but of different branches of math - into a single universal language that was capable of modeling every mathematical theory. What was surprising, though, is that this language, unexpectedly, could also perform universal computations. The same algorithms that could be computed by Turing Machines procedurally, could also be computed by the Lambda Calculus, through symbolic manipulations. The idea of using the Lambda Calculus for computing inspired the creation of an entire new branch of programming, now known as the functional paradigm. Haskell, Clojure, Elixir, Agda are languages based on that mindset, which is highly inspired by Church's invention - or discovery.

BEYOND TURING COMPLETENESS

If both Turing Machines (and procedural languages), and the Lambda Calculus (and functional languages), are capable of computation, which mindset is the "right one"? When it comes to raw capabilities, neither. Still on the 20th century, it was proven that, when it comes to computability, Turing Machines and the Lambda Calculus are equivalent. Every problem that one can solve, can also be solved by the other. That insight is known as the Church-Turing thesis, which essentially states that computers are capable of emulating each-other. If that was completely true, then the choice wouldn't matter. After all, if, for example, every programming language is capable of solving the same set of problems, then what is the point in making a choice?

Yet, while the Church-Turing hypothesis makes a statement about *computability*, it says nothing about *computation*. In other words, **a model can be inherently less efficient than other**. Historically, procedural languages such as C and Fortran have have consistently outperformed the fastest functional languages, such as Haskell and Ocaml. Yet, languages like Haskell and Agda provide abstractions that make entire classes of bugs unrepresentable. Historically, the functional paradigm has been more secure. Of course, these factors can vary greatly, but the point is that Turing's notion of equivalence is limited.

In 1983, Stephen Wolfram introduced the Rule 110, an elementary cellular automaton that has been shown to be as capable as both. Wolfram argues that this model is of fundamental importance for math and physics, and that a new kind of science should emerge from its study. These claims were met with harsh scepticism; after all, if all models are equivalent, what is the point? Yet, we've just stablished that, while equal in capacity, different models result in different practical outcomes. Perhaps there isn't a new branch of science to emerge from the study of alternative models of computation, but what about the design of processors and programming languages?

MODELS INFLUENCE HOW WE THINK, CREATE AND DESIGN

A model of computation has a significant impact on the way we design our languages, and several of their drawbacks can be traced down to limitations of the underlying model. For example, the parallel primitives of the procedural paradigm, mutexes and atomics, provide a contrived, complex solution to multi-threaded synchronization. As a result, parallelism is generally considered hard, and programmers still write sequential code by default. Similarly, security isn't natural to the procedural paradigm. Global state, mutable arrays and loops generate an explosion of possible execution paths, edge cases and off-by-one errors that make absolute security all but impossible. Even highly audited code, such as OpenSSL, is often compromised by out-of-bounds exploits. The functional paradigm handles both issues much better: there is an enourmous amount of inherent parallelism on functional programs, and, as mentioned, logic-based type systems make entire classes of bugs unrepresentable. But if that is the case, then why functional programs are still mostly single-threaded, and bug-ridden? And why isn't the functional paradigm more prevalent?

A strong factor, we argue, is performance. As much as the Turing Machine, and, thus, the procedural paradigm as a whole, is inadequate for parallelism and security, the Lambda Calculus, and the functional paradigm as a whole, is inadequate for efficient computations. The fundamental operation of the Lambda Calculus, substitution, may trigger an unbounded amount of copies of an unboundedly large argument. Because of that, it can not be performed in a bounded amount of steps, and, thus, there isn't a physical mapping of substitution to real-world physical processes. In other words, substitution isn't an atomic operation, which prevents us from creating efficient functional processors and runtimes. Attempts to solve the issue only pushed it into other directions, such as the need of shared references, which inhibit parallelism, or garbage collection, which isn't atomic. The failure of the functional paradigm to achieve satisfactory efficiency impacted its popularity, which, in turn, lead to tools like formal proofs to never catch up.

A PERFECT MODEL OF COMPUTATION

This raises the question: is there a model of computation which, like Turing Machine, has a reasonable physical implementation, yet, like the Lambda Calculus, has a robust logical interpretation? In 1997, Yves Lafont proposed a new alternative, the Interaction Combinators, on which substitution is broken down into 2 fundamental laws: commutation, which creates and copies information, and annihilation, which observes and destroys information. In a sense, this may resemble SKI combinators, but that isn't a good analogy, since SKI combinators still include non-atomic operations: K may erase an unboundedly large structure, and S may copy an unboundedly large structure. The charm, and elegance, of the Interaction Combinators is that its reduction laws are truly atomic: each operation can be completed in a constant amount of steps, and has a clear physical mapping. Not only that, they're inherently parallel, in the same sense that the Lambda Calculus has been claimed to be, in theory, but without the issues that let it to be, in practice.

Interestingly, every aspect which is considered good in other models of computation is present on Interaction Combinators, while negative aspects are almost entirely absent. Moreover, both the Lambda Calculus and the Turing Machine can be efficiently emulated by the Interaction Combinators, while the opposite isn't true. This suggests that, while the 3 systems are equivalent in terms of computability, the Interaction Combinators are more capable in terms of computation. Under certain point of view, one could argue that both the Turing Machine and the Lambda Calculus are slight distortions of this fundamental model, caused by human creativity, due to our historical intuitions regarding machines and mathematics. Perhaps machines and substitutions aren't as fundamental as we think, and some alien civilization has developed all its mathematical theories and computers based on annihilation and commutation, with no references to the Lambda Calculus, or the Turing Machine.

FROM INTERACTION COMBINATORS, TOWARDS A NEW ERA OF COMPUTING

We, at the Higher-Order Company, hold the view that Interaction Combinators are a more fundamental model of computation, and, consequently, that computers, processors and programming languages inspired by them would offer tangible benefits compared to the ones we built based on Turing Machines and the Lambda Calculus. The Higher-Order Company was created to research this new model of computation, and, through the different mindset it brings, catch insights that will let us produce groundbreaking technology that will push humanity towards the next level of computational maturity.

H.43 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-10min.md

Optimal Evaluation in 1 Minute or 10 Minutes or 10 Years

URL: https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6)

FILE: OPTIMAL_EVALUATION_IN_1_OR_10_OR_10_YEARS.MD

Optimal Evaluation in 1 Minute (or 10 Minutes) (or 10 Years)

SHORT VERSION (1 MINUTE)

A prerequisite to intelligence is the ability to find a program that explains a pattern. Programs are functions. To test a candidate program, we need to implement a "function evaluator". The problem is: all modern programming languages are sub-optimal "function evaluators", which, in the context of search, leads to massive slowdowns. To implement an optimal interpreter, we need to: 1. postpone the execution of expressions, to avoid wasted work, 2. cache the result of postponed expressions, to avoid duplicated work, 3. incrementally copy cached structures, to ensure 1 and 2 don't interfere. Haskell *almost* achieves this, but falls short on 3, because it is unable to incrementally copy lambdas. To solve this, we introduce the concept of a "superposition", which allows multiple versions of a term to exist simultaneously. This ensures that no work is ever wasted or duplicated, allowing us to optimally interpret (or compile) a given language.

LONG VERSION (10 MINUTES)

A prerequisite to intelligence is the ability to find a program that explains a pattern. Programs are functions. To test a candidate program, we need to implement a "function evaluator". The problem is: it turns out all modern programming languages are sub-optimal "function evaluators"! For example, consider the following Python expression:

```
arr = [9999 ** 2, 42]
print(arr[1])
```

When running this program, Python will gladly compute `9999**2`, even though that expression isn't ever used. That's wasted work. In a large space of random programs, the number of unnecessary computations is massive - thus, the amount of wasted work in naive program search is colossal, making it very slow. But what if we could prevent all that waste?

Here's the plan: let's "postpone" computing expressions for only when they're needed. So, for example, in the program above, we'd only compute `9999**2` if we printed it to the screen - which we don't, so it is never computed. By designing an interpreter around that idea, we'd never evaluate unnecessary expressions, greatly speeding up our computation. Except for a small problem: postponing, turns out, can accidentally cause *duplicated work*. For example, consider:

```
mul2 = lambda x: x + x
print(mul2(9999**2))
```

If we blindly postpone the execution of `9999**2`, the program above will evaluate to:

```
(9999**2) + (9999**2)
```

Which will compute the same expression twice. That's terrible! Postponing let us avoid unnecessary expressions, at the cost of computing necessary expressions twice, which is equally bad. Is all hope lost? Not yet. Here's an idea: what if we also *cached* the result of a postponed expression? For example, in the same program:

```
mul2 = lambda x: x + x
print(mul2(9999**2))
```

We could first apply `mul2`, postponing `9999**2` in a "cached thunk":

```
cache x = 9999**2
print(x + x)
```

Now, since `x` is clearly used, we could then proceed to compute `9999**2`, copy the result, and perform the addition. That wastes no work! So, is that all? Are we optimal yet? Well, almost... but now we need to decide how we evaluate cached expressions. For example, consider:

```
cache xs = [10 ** 2, 20 ** 2, 30 ** 2]
print(xs[0] + xs[0])
```

In this case, the cached expression has many sub-expressions that also need to be evaluated. How do we proceed? If we just evaluate the whole expression and "memcpy" the entire result, we'll compute `20 ** 2` and `30 ** 2`, which aren't used. That's not good. If we, instead, copy first, and evaluate later, then, we'll have:

```
cache xs = [10 ** 2, 20 ** 2, 30 ** 2]
print(xs[0] + xs[0])
-----------------------------------------------------------------------
print([10 ** 2, 20 ** 2, 30 ** 2][0] + [10 ** 2, 20 ** 2, 30 ** 2][0])
-----------------------------------------------------------------------
print((10 ** 2) + (10 ** 2))
---------------------------
print(100 + 100)
```

Which not only copies the whole array unnecessarily, but also computes `10 ** 2` twice. Seems like we're back to the same issue! Fortunately, it turns out there IS a proper way to do it: just **copy incrementally**. Like this:

```
cache xs = [10 ** 2, 20 ** 2, 30 ** 2]
print(xs[0] + xs[0])
-------------------------------------
cache xs = [100, 20 ** 2, 30 ** 2]
print(xs[0] + xs[0])
-------------------------------------
cache xs = [20 ** 2, 30 ** 2]
print(100 + 100)
-------------------------------------
print(200)
```

Notice how we never computed `20 ** 2` or `30 ** 2`, and only computed `10 ** 2` once. That's perfect!

This interleaved "postponed expression + incremental cloning" approach is provably **optimal**. That is, an interpreter designed that way is guaranteed to never 1. compute unused expressions nor 2. compute the same expression twice. And that's fast! This is what "optimal evaluation" is all about. But there's still a last catch... what if the cached expression is, itself, a lambda? For example, consider:

```
cache fn = lambda x: x + 10 ** 2
print(fn(1) + fn(2))
```

Take a moment to reflect about the program above. How do we avoid computing `10 ** 2` twice? Since we're postponing computations, we can't execute `10 ** 2` yet (we don't know if it will be used). Thus, the best we can do is just copy the function, and proceed like:

```
print((lambda x: x + 10 ** 2)(1) + (lambda x: x + 10 ** 2)(2))
-------------------------------------------------------------
print(1 + 10 ** 2 + 2 + 10 ** 2)
--------------------------------
print(1 + 100 + 2 + 100)
------------------------
print(203)
```

Here, clearly, `10 ** 2` was evaluated twice, wasting compute. So, how do we avoid that in a classical programming language? It turns out we can't, and this is the exact moment when all languages in the world - even Haskell, which does all the tricks above - become sub-optimal. That's because there is no way to "incrementally copy" a function without expanding our universe, just like we need to expand real numbers with imaginary units to solve certain equations. I call that extension "superposition". Here's how it goes:

```
cache {fn0,fn1} = lambda x: x + 10 ** 2
print(fn0(1) + fn1(2))
-----------------------------------------------
cache {fn0,fn1} = {x0,x1} + 10 ** 2
print((lambda x0: fn0)(1) + (lambda x1: fn1)(2))
-----------------------------------------------
cache {fn0,fn1} = {1,2} + 10 ** 2
print(fn0 + fn1)
-----------------------------------------------
cache {fn0,fn1} = {1,2} + 100
print(fn0 + fn1)
-------------------------------------------
cache {fn0,fn1} = {1 + 100, 2 + 100}
print(fn0 + fn1)
-----------------------------------------------
cache {fn0,fn1} = {101, 102}
print(fn0 + fn1)
-----------------------------------------------
print(101 + 102)
-----------------------------------------------
print(203)
```

Here, instead of caching "fn" directly, we cached a superposition of its future copies: "fn0" and "fn1". Then, we proceeded to incrementally copy its binder (`lambda x:`), while keeping its body (`x + 10 ** 2`) intact. As a result, its variable, `x`, became a superposed variable, `{x0,x1}`, which was bound to its two partial copies (`lambda x0:` and `lambda x1:`). This has the unintuitive consequence that some lambdas are now bound to variables that occur outside of their bodies! That looks forbidden, but is actually fine: in the end, just like imaginary numbers, everything cancels out, and the final result is correct: `203`. And the good news is: we arrived at that result without ever computing `10 ** 2` twice.

A lazy interpreter that implements caching and superpositions this way is guaranteed to never compute any unnecessary expression, and never duplicate any necessary expression. In other words, it is an **optimal evaluator** for the language it interprets (or compiles). This has some crazy, yet beautiful consequences. For example, in an optimal evaluator, we can implement addition by repeated increment:

```
def add(x, y):
  for i in range(x):
    y += 1
  return y
```

And, despite looking embarrassingly inefficient, it performs as well as the "add-carry" algorithm (!!!). Regarding real-world applications, in situations where there is an exponential amount of redundant computations that are generated by some simple procedure - for example, church-nat exponentiation, discrete program search, and certain forms of ray tracing - optimal evaluation can yield potentially massive speedups. I personally believe clever usage of superposition can result in the design of algorithms that solve hard number theory problems like the DLP - but, of course, that's my speculation.

EXPERT VERSION (10 YEARS)

If you're interested in the underlying theory behind optimal evaluation, start by reading The Optimal Implementation of Functional Programming Languages (https://www.amazon.com/Implementation-Functional-Programming-Languages-Theoretical/dp/0521621127), by Andrea Asperti and Stefano Guerrini. If you want to explore this model of evaluation, Bend (https://github.com/HigherOrderCO/Bend) is a practical programming language that will be optimally evaluated, but note that, in its current version, it isn't lazy yet - this will arrive later this year (2024). For now, you can use HVM1 (https://github.com/HigherOrderCO/HVM1) directly.

H.44 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-collapse-monad.md

The Collapse Monad

URL: https://gist.github.com/VictorTaelin/60d3bc72fb4edefecd42095e44138b41 (https://gist.github.com/VictorTaelin/60d3bc72fb4edefecd42095e44138b41)

FILE: COLLAPSE_MONAD.HS

import Control.Monad (ap, forM_) import qualified Data.Map as M

-- The Collapse Monad

<hr>

-- The Collapse Monad allows collapsing a value with labelled superpositions -- into a flat list of superposition-free values. It is like the List Monad, -- except that, instead of always doing a cartesian product, it will perform -- pairwise merges of distant parts of your program that are "entangled" -- under the same label. Examples: -- - collapse (&0{1 2}, 3) == [(1,3), (2,3)] -- - collapse (&0{1 2}, &0{3 4}) == [(1,3), (2,4)] -- - collapse (&0{1 2}, &1{3 4}) == [(1,3), (1,4), (2,3), (2,4)] -- - collapse (&0{1 2}, &1{3 4}, 5) == [(1,3,5), (1,4,5), (2,3,5), (2,4,5)] -- Note how the second line above doesn't return the full cartesian product of -- {1 2} and {3 4}; instead, it combines elements pairwise, because both -- superpositions have the same label. I'm posting this file as a template to -- recall later. This algorithm is used to flatten HVM3's results, which can be -- superposed, back to a list of λ-terms without superpositions.

-- A bit-string data Bin = O Bin | I Bin | E deriving Show

-- A Collapse is a tree of superposed values data Collapse a = Sup Int (Collapse a) (Collapse a) | Val a

-- The Collapse Monad -- Note: could be optimized using an IntMap instead of a List bind :: Collapse a -> (a -> Collapse b) -> Collapse b bind a f = fork a (repeat (\x -> x)) where fork (Val v) paths = pass (f v) (map (\x -> x E) paths) fork (Sup k x y) paths = Sup k (fork x (mut k put0 paths)) (fork y (mut k put1 paths)) pass (Val v) paths = Val v pass (Sup k x y) paths = case paths !! k of E -> Sup k x y O p -> pass x (mut k (_->p) paths) I p -> pass y (mut k (_->p) paths) put0 bs = \x -> bs (O x) put1 bs = \x -> bs (I x)

-- Collapses a Term and flattens into a list flatten :: Collapse a -> [a] flatten (Val x) = [x] flatten (Sup _ x y) = flatten x ++ flatten y

-- Mutates an element at given index in a list mut :: Int -> (a -> a) -> [a] -> [a] mut 0 f (x:xs) = f x : xs mut n f (x:xs) = x : mut (n-1) f xs mut _ _ [] = []

instance Functor Collapse where fmap f (Val v) = Val (f v) fmap f (Sup k x y) = Sup k (fmap f x) (fmap f y)

instance Applicative Collapse where pure = Val (<*>) = ap

instance Monad Collapse where return = pure (>>=) = bind

-- Example

<hr>

-- Collapsing a simple DSL with Superpositions

-- A simple DSL with tuples, numbers and superpositions data Term = TNum Int -- number | TTup Term Term -- tuple | TSup Int Term Term -- superposition

-- Shows a term with superpositions showTerm :: Term -> String showTerm (TNum n) = show n showTerm (TTup x y) = "(" ++ showTerm x ++ "," ++ showTerm y ++ ")" showTerm (TSup k a b) = "&" ++ show k ++ "{" ++ showTerm a ++ " " ++ showTerm b ++ "}"

-- A Collapser for our DSL -- - On normal values, we just use `<-` -- - On superpositons, we return a Sup collapse :: Term -> Collapse Term collapse (TNum x) = do return $ TNum x collapse (TTup l r) = do l <- collapse l r <- collapse r return $ TTup l r collapse (TSup k a b) = Sup k (collapse a) (collapse b)

-- Test program main :: IO () main = do -- Builds the term ({&0{1 2},&1{3 4}},{&0{5 6},&1{7 8}}) let tup0 = TTup (TSup 0 (TNum 1) (TNum 2)) (TSup 1 (TNum 3) (TNum 4)) let tup1 = TTup (TSup 0 (TNum 5) (TNum 6)) (TSup 1 (TNum 7) (TNum 8)) let term = TTup tup0 tup1 -- Collapses it let coll = flatten (collapse term) -- Prints the sup-free term list forM_ coll $ \ term -> putStrLn $ showTerm term

H.45 · `bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-deforestation.md`

That's a great question. Sadly, no, HVM won't do that, by design. That's because we follow the Linux philosophy of doing just one thing, and doing it well: it is a runtime, not an optimizing compiler. GHC does both at once, but HVM separates these responsibilities. As such, it will just run the program you feed as fast as it can, but it won't alter it in any way. It is the compiler's job (i.e., the language that is targeting the HVM) to apply optimizing transformations. This is good because it gives the source language full control over what will actually run.

That said, it is worth noting that, due to optimal reduction, we already have "runtime deforestation"! Yes, that's right. That's because optimal reduction can be seen as as if HVM was doing aggressive inlining at runtime, which in all other languages used to be a compile-time thing. Let me give you a concrete example. Consider the following program:

```
(Gen 0 r) = r
(Gen n r) = (Gen (- n 1) (List.cons (- n 1) r))

(Map f List.nil)         = List.nil
(Map f (List.cons x xs)) = (List.cons (f x) (Map f xs))

(Main n) =
  let list = (Gen 100 Nil)
  let list = (Map λx(x) list)
  let list = (Map λx(x) list)
  ...
  let list = (Map λx(x) list)
  let list = (Map λx(x) list)
  list
```

It just applies `Map` to a list of 100 elements, many times in a row. We can measure the complexity of this program by querying the number of graph rewrite rules with `hvm run -c true`. Here are the results, based on the number of times we called `Map`:

```
calls | rewrites
----- | --------
0     | 402
1     | 802
2     | 1202
3     | 1602
4     | 2002
5     | 2402
6     | 2802
7     | 3202
8     | 3602
9     | 4002
```

As you can see, each call to Map adds exactly 400 to the total cost. Now, as you may be aware, a popular deforestation technique employed by Haskell is `build/foldr`, which works (using List as an example) by 1. folding over the list, 2. implementing the functions to operate on the folded-version, 3. rebuilding the list. The reason this works is that the compiler is able to inline algorithms implemented for folded lists, removing all the intermediate allocations. Clever. Now, on HVM, we don't need the compiler to do that. **The runtime itself is capable of deforesting!** So, let's reimplement `Map` using the foldr/build approach:

```
(Gen 0 r) = r
(Gen n r) = (Gen (- n 1) (List.cons (- n 1) r))

(Fold List.nil)         = λcons λnil nil
(Fold (List.cons x xs)) = λcons λnil (cons x ((Fold xs) cons nil))

(Build fold) = (fold λhλt(List.cons h t) List.nil)

(Map f fold) = λcons λnil (fold λh (cons (f h)) nil)

(Main n) =
  let list = (Gen 100 Nil)
  let list = (Fold list)
  let list = (Map λx(x) list)
  let list = (Map λx(x) list)
  ...
  let list = (Map λx(x) list)
  let list = (Map λx(x) list)
  (Build list)
```

Here is the number of rewrites in function of the number of calls to Map:

```
calls | rewrites
----- | --------
0     | 1403
1     | 1408
2     | 1413
3     | 1418
4     | 1423
5     | 1428
6     | 1433
7     | 1438
8     | 1443
9     | 1448
```

As you can see, each call to `Map` only added `5` rewrite rules to the total count. Since the list has 100 elements, it would be impossible to allocate a whole list with just `5` graph rewrites, indicating that, indeed, HVM removed the need for allocating intermediate structures! Of course, the function used is just `id`. If you use anything else, the cost will increase proportional to the cost of the function itself, but no intermediate structures will be allocated. Even cooler, if the function applied itself fuses, then it will also be optimized that way. For example, mapping 10x `λx (Add 1 x)` will turn it into a single map of `λx (Add 10 x)`, as long as `Add` is implemented with λ-encodings too. This approach generalizes to all cases, so more advanced approaches (like Haskell's stream fusion) can also be done on HVM with no change to the compiler. Just implement it as a lib, and it will work as expected.

H.46 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-how.md



Note: this is a public draft. It contains a lot of errors and may be too meme-ish and handholding in some parts. I know it needs improvements. I'll review and finish in a future. Corrections and feedbacks are welcome!

How? ====

- TL;DR
- Core Language Overview
- What makes it fast
- Rewrite Rules
- Low-level Implementation
- Bonus: Copatterns
- Bonus: Abusing Beta-Optimality
- Bonus: Abusing Parallelism

TL;DR =====

Since this became a long, in-depth overview, here is the TL;DR for the lazy:

HVM doesn't need a global, stop-the-world garbage collector because every "object" only exists in one place, **exactly like in Rust**; i.e., HVM is *linear*. The catch is that, when an object needs to be referenced in multiple places, instead of a complex borrow system, HVM has an elegant, pervasive **lazy clone primitive** that works very similarly to Haskell's evaluation model. This makes cloning essentially free, because the copy of any object isn't made in a single, expensive pass, but in a layer-by-layer, on-demand fashion. And the nicest part is that this clone primitive works for not only data, but also for lambdas, which explains why HVM has better asymptotics than GHC: it is capable of **sharing computations inside lambdas, which GHC can't**. That was only possible due to a key insight that comes from Lamping's Abstract Algorithm for optimal evaluation of λ-calculus terms. Finally, the fact that objects only exist in one place greatly simplifies parallelism. Notice how there is only one use of atomics in the entire runtime.c.

This was all known and possible since years ago (see other implementations of optimal reduction), but all implementations of this algorithm, until now, represented terms as graphs. This demanded a lot of pointer indirection, making it slow in practice. A new memory format, based on SIC `(https://github.com/VictorTaelin/Symmetric-Interaction-Calculus)`, takes advantage of the fact that inputs are known to be λ-terms, allowing for a 50% lower memory usage, and letting us avoid several impossible cases. This made the runtime 50x (!) faster, which finally allowed it to compete with GHC and similar. And this is just a prototype I wrote in about a month. I don't even consider myself proficient in C, so I have expectations for the long-term potential of HVM.

HVM's optimality and complexity reasoning comes from the vast literature on the optimal evaluation of functional programming languages. This book (https://www.amazon.com/Implementation-Functional-Programming-Languages-Theoretical/dp/0521621127), by Andrea Asperti and Stefano Guerrini, has a great overview. HVM is merely a practical, efficient implementation of the bookkeeping-free reduction machine depicted on the book (pages 14-39). Its high-order machinery has a 1-to-1 relationship to the theoretical model, and the same complexity bounds, and respective proofs (chapter 10) apply. HVM has additional features (machine integers, datatypes) that do not affect complexity.

That's about it. Now, onto the long, in-depth explanation.

HVM's Core Language Overview ============================

HVM is, in essence, just a virtual machine that evaluates terms in its core language. So, before we dig deeper, let's review that language. HVM's Core is a very simple language that resembles untyped Haskell. It features lambdas (eliminated by applications), constructors (eliminated by user-defined rewrite rules) and machine integers (eliminated by operators).

```
term ::=
  | λvar. term               # a lambda
  | (term term)              # an application
  | (ctr term term ...)      # a constructor
  | num                      # a machine int
  | (op2 term term)          # an operator
  | let var = term; term     # a local definition

rule ::=
  | term = term

file ::= list<rule>
```

A constructor begins with a name and is followed by up to 16 fields. Constructor names must start uppercased. For example, below is a pair with 2 numbers:

```
(Pair 42 7)
```

HVM files consist of a series of `rules`, each with a left-hand term and a right-hand term. These rules enact a computational behavior where every occurrence of the left-hand term is replaced by its corresponding right-hand term. For example, below is a rule that gets the first element of a pair:

```
(Fst (Pair x y)) = x
```

Once that rule is enacted, the `(Fst (Pair 42 7))` term will be reduced to `42`. Note that `(Fst ...)` is itself just a constructor, even though it is used like a function. This has important consequences which will be elaborated later on. From this, the remaining syntax should be pretty easy to guess. As an example, we define and use the `Map` function on `List` as follows:

```
(Map f Nil)         = Nil
(Map f (Cons x xs)) = (Cons (f x) (Map f xs))

(Main) =
  let list = (Cons 1 (Cons 2 Nil))
  let add1 = λx (+ x 1)
  (Map add1 list)
```

By running this file (with `hvm r main`), HVM outputs `(Cons 2 (Cons 3 Nil))`, having incremented each number in `list` by 1. Notes:

- Application is distinguised from constructors by case (`(f x)` vs `(F x)`).
- The parentheses can be omitted from unary constructors (`Nil` == `(Nil)`)
- You can abbreviate applications (`(((f a) b) c ...)` == `(f a b c ...)`).
- You may write `@` instead of `λ`.
- Check this (https://github.com/Kindelia/HVM/issues/64#issuecomment-1030688993) issue about how constructors, applications and currying work.

What makes it fast ==========

What makes HVM special, though, is **how** it evaluates its programs. HVM has one simple trick that hackers don't want you to know. This trick is responsible for HVM's major features: beta-optimality, parallelism, and no garbage collection. But before we get too technical, we must first talk about **clones**, and how their work obsession ruins everything, for everyone. This section should provide more context and a better intuition about why things are the way they are.

CLONES RUIN EVERYTHING

By clones, I mean when a value is copied, shared, replicated, or whatever else you call it. For example, consider the JavaScript program below:

```
function foo(x, y) {
  return x + x;
}
```

To compute `foo(2, 3)`, the number `2` must be **cloned** before adding it to itself. This seemingly innocent operation has made a lot of people very confused and has been widely regarded as the hardest problem of the 21st century.

The main issue with clones is how they interact with the **order of evaluation**. For example, consider the expression `foo(2 * 2, 3 * 3)`. In a **strict** language, it is evaluated as such:

```
foo(2 * 2, 3 * 3) // the input
foo(4    , 9    ) // arguments are reduced
4 + 4              // foo is applied
8                  // the output
```

But this computation has a silly issue: the `3 * 3` value is not necessary to produce the output, so the `3 * 3` multiplication was wasted work. This led to the idea of **lazy** evaluation. An initial implementation of this idea would operate as follows:

```
foo(2 * 2, 3 * 3) // the input
(2 * 2) + (2 * 2) // foo is applied
4       + 4       // arguments are reduced
8                 // the output
```

Notice how the `3 * 3` expression was never computed, thereby saving work? Instead, the other `2 * 2` expression has been computed twice, yet again resulting in **wasted work**! That's because `x` was used two times in the body of `foo`, which caused the `2 * 2` expression to be **cloned** and, thus, computed twice. In other words, clones ruined the virtue of laziness.

EVERYONE'S SOLUTION: BAN THE CLONES

Imagine a language without clones. Such a language would be computationally perfect. Lazy evaluators wouldn't waste work, since expressions can't be cloned. Garbage collection would be cheap, as every object would only have one reference. Parallelism would be trivial, because there would be no simultaneous accesses to the same object. Sadly, such a language wouldn't be practical. Imagine never being able to copy anything! Therefore, real languages must find a way to let their users replicate values, without impacting the features they desire, all while avoiding these expensive clones. In previous languages, the solution has almost always been the use of references of some sort.

For example, Haskell is lazy. To avoid "cloning computations", it implements thunks, which are nothing but *memoized references* to shared expressions, allowing the `(2 * 2)` example above to be cached. This solution, though, breaks down when there are lambdas. Similarly, Rust is GC-free, so every object has only one "owner". To avoid too much cloning, it implements a complex *borrowed references* system, allowing the same object to be accessed from multiple places, when the compiler can prove it is safe. Finally, parallel languages require mutexes and atomics to synchronize accesses to *shared references*. In other words, references saved the world by letting us avoid these clones, and that's great... right?

> clone wasn't the impostor

References. **They** ruin everything. They're the reason Rust is so hard to use. They're the reason parallel programming is so complex. They're the reason Haskell isn't optimal, since thunks can't share computations that have free variables (i.e., any expression inside lambdas). They're why a 1-month-old prototype beats GHC in the same class of programs it should thrive in. It isn't GHC's fault. References are the real culprits.

HVM'S SOLUTION: MAKE CLONES CHEAP

Clones aren't bad. They just need to relax.

Once you understand the context above, grasping how HVM can be optimally lazy, non-GC'd and inherently parallel is easy. At its base, it has the same "linear" core that both Haskell and Rust share in common (which, as we've just established, is already all these things). The difference is that, instead of adding some kind of clever reference system to circumvent the cost of cloning... **HVM introduces a pervasive, lazy clone primitive**.

HVM's runtime has no references. Instead, it features a `.clone()` primitive that has zero cost, until the cloned value needs to be read. Once it does, instead of being copied whole, it's done layer by layer, on-demand.

For the purpose of lazy evaluation, HVM's lazy clones works like Haskell's thunks, except they do not break down on lambdas. For the context of garbage collection, since the data is actually copied, there are no shared references, so memory can be freed when values go out of scope. For the same reason, parallelism becomes trivial, and the runtime's `reduce()` procedure is almost entirely thread safe, requiring minimal synchronization.

In other words, think of HVM as Rust, except replacing the borrow system by a very cheap `.clone()` operation that can be used and abused with no mercy. This is the secret sauce! Easy, right? Well, no. There is still a big problem to be solved: **how do we incrementally clone a lambda?** There is a beautiful answer to this problem that made this all possible. Let's get technical!

HVM's Rewrite Rules ==========

HVM is, in essence, a graph-rewrite system, which means that all it does is repeatedly rewrite terms in memory until there is no more work left to do. These rewrites come in two forms: user-defined and primitive rules.

User-defined Rules ----------

User-defined rules are generated from equations in a file. For example, the following equation:

```
(Foo (Tic a) (Tac b)) = (+ a b)
```

Generates the following rewrite rule:

```
(Foo (Tic a) (Tac b))
-------------------- Foo-Rule-0
(+ a b)
```

It should be read as "the expression above reduces to the expression below". So, for example, `Foo-rule-0` dictates that `(Foo (Tic 42) (Tac 7))` reduces to `(+ 42 7)`. As for the primitive rules, they deal with lambdas, native numbers and the duplication primitive. Let's start with numeric operations.

Operations ----------

```
(<op> x y)
------- Op2-U32
x +  y if <op> is +
x -  y if <op> is -
x *  y if <op> is *
x /  y if <op> is /
x %  y if <op> is %
x &  y if <op> is &
x |  y if <op> is |
x ^  y if <op> is ^
x >> y if <op> is >>
x << y if <op> is <<
x <  y if <op> is <
x <= y if <op> is <=
x == y if <op> is ==
x >= y if <op> is >=
x >  y if <op> is >
x != y if <op> is !=
```

This should be read as: *"the addition of `x` and `y` reduces to `x + y`"*. This just says that we can perform numeric operations on HVM. For example, `(+ 2 3)` is reduced to `5`, `(* 5 10)` is reduced to `50`, and so on. HVM numbers are 32-bit unsigned integers, but more numeric types will be added in the future.

Number Duplication ----------

```
dup x y = N
-----------
x <- N
y <- N
```

This should be read as: *"the duplication of the number `N` as `x` and `y` reduces to the substitution of `x` by a copy of `N`, and of `y` by another copy of `N`"*. Before explaining what is going on here, let me also present the constructor duplication rule below.

Constructor Duplication ----------

```
dup x y = (Foo a b ...)
---------------------- Dup-Ctr
dup a0 a1 = a
dup b0 b1 = b
...
x <- (Foo a0 b0 ...)
y <- (Foo a1 b1 ...)
```

This should be read as: *"the duplication of the constructor `(Foo a b ...)` as `x` and `y` reduces to the duplication of `a` as `a0` and `a1`, the duplication of `b` as `b0` and `b1`, and the substitution of `x` by `(Foo a0 b0 ...)` and the substitution of `y` by `(Foo a1 b1 ...)`"*.

There is a lot of new information here, so, before moving on, let's dissect it all one by one.

1. What the hell is `dup`? That is an **internal duplication node**. You can't write it directly on the user-facing language; instead, it is inserted by the pre-processor whenever you use a variable more than once. For example, at compile time, the equation below:

```
(Foo a) = (+ a a)
```

Is actually replaced by:

```
(Foo a) =
  dup x y = a
  (+ x y)
```

Because of that transformation, **every runtime variable only occurs once**. The effect of `dup` is that of cloning an expression, and moving it to two locations. For example, the program below:

```
dup x y = 42
(Pair x y)
```

Is reduced to:

```
(Pair 42 42)
```

2. By "substitution", we mean "replacing a variable by a value". For example, the substitution of `x` by `7` in `[1, x, 8]` would be `[1, 7, 8]`. Since every variable only occurs once in the runtime, substitution is a fast, constant time operation that performs either 1 or 2 array writes.

3. `dup`s aren't stored inside the expressions. Instead, they "float" on the global scope. That's why they're always written on top.

4. Remember that `dup` (like all other rules) is only triggered when it is needed, due to lazy evaluation. That's what makes it ultra-cheap. In a way, it is as if HVM has added a `.clone()` to every variable used more than once. And that's fine.

5. Even though the user-facing language makes no distinction between constructors and functions, the runtime does, for optimality purposes. Specifically, a duplication is only applied for constructors that are not used as functions. This equal treatment means we can write Copatterns easily in HVM; see the bonus section.

EXAMPLE

Now that you know all that, let's watch `dup` in action, by visualizing how the `[1 + 1, 2 + 2, 3 + 3]` list is cloned. Lines separate reduction steps.

```
dup x y = (Cons (+ 1 1) (Cons (+ 2 2) (Cons (+ 3 3) Nil)))
(Pair x y)
---------------------------------------- Dup-Ctr
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
dup a b = (+ 1 1)
(Pair
  (Cons a x)
  (Cons b y)
)
---------------------------------------- Op2-U32
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
dup a b = 2
(Pair
  (Cons a x)
  (Cons b y)
)
---------------------------------------- Dup-U32
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
(Pair
  (Cons 2 x)
  (Cons 2 y)
)
---------------------------------------- Dup-Ctr
dup x y = (Cons (+ 3 3) Nil)
dup a b = (+ 2 2)
(Pair
  (Cons 2 (Cons a x))
  (Cons 2 (Cons b y))
)
---------------------------------------- Op2-U32
dup x y = (Cons (+ 3 3) Nil)
dup a b = 4
(Pair
  (Cons 2 (Cons a x))
  (Cons 2 (Cons b y))
)
---------------------------------------- Dup-U32
dup x y = (Cons (+ 3 3) Nil)
(Pair
  (Cons 2 (Cons 4 x))
  (Cons 2 (Cons 4 y))
)
---------------------------------------- Dup-Ctr
dup x y = Nil
dup a b = (+ 3 3)
(Pair
  (Cons 2 (Cons 4 (Cons a x)))
  (Cons 2 (Cons 4 (Cons b y)))
)
---------------------------------------- Op2-U32
dup x y = Nil
dup a b = 6
(Pair
  (Cons 2 (Cons 4 (Cons a x)))
  (Cons 2 (Cons 4 (Cons b y)))
)
---------------------------------------- Dup-U32
dup x y = Nil
(Pair
  (Cons 2 (Cons 4 (Cons 6 x)))
  (Cons 2 (Cons 4 (Cons 6 y)))
)
---------------------------------------- Dup-Ctr
(Pair
  (Cons 2 (Cons 4 (Cons 6 Nil)))
  (Cons 2 (Cons 4 (Cons 6 Nil)))
)
```

In the end, we made two copies of the list. Note how the `(+ 1 1)` expression, was NOT ”cloned”. It only happened once, even though we evaluated the program lazily. And, of course, since the cloning itself is lazy, if we only needed parts of the list, we wouldn’t need to make two full copies. For example, consider the following program instead:

```
dup x y = (Cons (+ 1 1) (Cons (+ 2 2) (Cons (+ 3 3) Nil)))
(Pair (Head x) (Head (Tail y)))
---------------------------------------- Dup-Ctr
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
dup a b = (+ 1 1)
(Pair (Head (Cons a x)) (Head (Tail (Cons b y))))
---------------------------------------- Head
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
dup a b = (+ 1 1)
(Pair a (Head (Tail (Cons b y))))
---------------------------------------- Op2-U32
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
dup a b = 2
(Pair a (Head (Tail (Cons b y))))
---------------------------------------- Dup-U32
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
(Pair 2 (Head (Tail (Cons 2 y))))
---------------------------------------- Tail
dup x y = (Cons (+ 2 2) (Cons (+ 3 3) Nil))
(Pair 2 (Head y))
---------------------------------------- Dup-Ctr
dup x y = (Cons (+ 3 3) Nil)
dup a b = (+ 2 2)
(Pair 2 (Head (Cons b y)))
---------------------------------------- Head
dup x y = (Cons (+ 3 3) Nil)
dup a b = (+ 2 2)
(Pair 2 b)
---------------------------------------- Op2-U32
dup x y = (Cons (+ 3 3) Nil)
dup a b = 4
(Pair 2 b)
---------------------------------------- Dup-U32
dup x y = (Cons (+ 3 3) Nil)
(Pair 2 4)
---------------------------------------- Collect
(Pair 2 4)
```

Notice how only the minimal amount of copying was performed. The first part of the list (`(Cons (+ 1 1) ...)`) was copied twice, the second part (`(Cons (+ 2 2) ...)`) was copied once, and the rest (`(Cons (+ 3 3) Nil)`) was collected without even touching it. Collection is orchestrated by variables that go out of scope. For example, in the last lines, `x` and `y` both aren't referenced anywhere. That triggers the collection of the remaining list.

That was a lot of info. Hopefully, by now, you have an intuition about how the lazy duplication primitive works. Moving on.

LAMBDA APPLICATION

```
(λx(body) arg)
-------------- App-Lam
x <- arg
body
```

This is the famous beta-reduction rule. This must be read as: *"the application of the lambda `λx(body)` to the argument `arg` reduces to `body`, and the substitution of `x` by `arg`"*. For example, `(λx(Single x) 42)` is reduced to `(Single 42)`. Remember that variables only occur once. Because of that, beta-reduction is a very fast operation. A modern CPU can perform more than 200 million beta-reductions per second, in a single core. As an example:

```
(λxλy(Pair x y) 2 3)
------------------- App-Lam
(λy(Pair 2 y) 3)
------------------- App-Lam
(Pair 2 3)
```

Simple, right? This rule is beautiful, but the next one is special, as it is responsible for making all of HVM possible.

LAMBDA DUPLICATION

Incrementally cloning datatypes is a neat idea. But there is nothing special to it. In fact, that **is** exactly how Haskell's thunks behave! But, now, take a moment and ask yourself: **how the hell do we incrementally clone a lambda**?

```
dup a b = λx(body)
------------------ Dup-Lam
a <- λx0(b0)
b <- λx1(b1)
x <- {x0 x1}
dup b0 b1 = body

dup a b = {r s}
--------------- Dup-Sup
a <- r
b <- s
```

Here is how. This may be a bit overwhelming. A good place to start is by writing this in plain English. It reads as: "the duplication of a lambda `λx(body)` as `a` and `b` reduces in the duplication of its `body` as `b0` and `b1`, and the substitution of `a` by `λx0(b0)`, `b` by `λx1(b1)` and `x` by the superposition `{x0 x1}`".

What this is saying is that, in order to duplicate a lambda, we must duplicate its body; then we must create two lambdas. Then, weird things happen with its variable. And then there is a brand new construct, the superposition, that I haven't explained yet. But, this is fine. Let's try to do it with an example:

```
dup a b = λx λy (Pair x y)
(Pair a b)
```

This program just makes two copies of the `λx λy (Pair x y)` lambda. But, to get there, we are not allowed to copy the entire lambda whole. Instead, we must go through a series of incremental lazy steps. Let's try it, and copy the outermost lambda (`λx`):

```
dup a b = λy (Pair x y)
(Pair λx(a) λx(b))
```

Can you spot the issue? As soon as the lambda is copied, it is moved to another location of the program, which means it gets detached from its own body. Because of that, the variable `x` gets unbound on the first line, and the body of each copied `λx` has no reference to `x`. That makes no sense at all! How do we solve this?

First, we must let go of material goods and accept a cruel reality of HVM: **lambdas don't have scopes**. That is, a variable bound by a lambda can occur outside of its body. So, for example, `(Pair x (λx(8) 7))` would reduce to `(Pair 7 8)`. Please, take a moment to make sense out of this... even if looks like it doesn't.

Once you accept that in your heart, you'll find that the program above will make a little more sense, because we can say that the `λx` binder on the second line is "connected" to the `x` variable on the first line, even if it's outside. But there is still a problem: there are **two** lambdas bound to the same variable. If the left lambda gets applied to an argument, it should NOT affect the second one. But with the way it is written, that's what would happen. To work around this issue, we need a new construct: the **superposition**. Written as `{r s}`, a superposition stands for an expression that is part of two partially copied lambdas. So, for example, `(Pair {1 2} 3)` can represent either `(Pair 1 3)` or `(Pair 2 3)`, depending on the context.

This gives us the tools we need to incrementally copy these lambdas. Here is how that would work:

```
dup a b = λx(λy(Pair x y))
(Pair a b)
------------------------------------------------ Dup-Lam
dup a b = λy(Pair {x0 x1} y)
(Pair λx0(a) λx1(b))
------------------------------------------------ Dup-Lam
dup a b = (Pair {x0 x1} {y0 y1})
(Pair λx0(λy0(a)) λx1(λy1(b)))
------------------------------------------------ Dup-Ctr
dup a b = {x0 x1}
dup c d = {y0 y1}
(Pair λx0(λy0(Pair a c)) λx1(λy1(Pair b d)))
------------------------------------------------ Dup-Sup
dup c d = {y0 y1}
(Pair λx0(λy0(Pair x0 c)) λx1(λy1(Pair x1 d)))
------------------------------------------------ Dup-Sup
(Pair λx0(λy0(Pair x0 y0)) λx1(λy1(Pair x1 y1)))
```

Wow, did it actually work? Yes, it did. Notice that, despite the fact that "weird things" happened during the intermediate steps (specifically, variables got out of their own lambda bodies, and parts of the program got temporarily superposed), in the end, it all worked out, and the result was proper copies of the original lambdas. This allows us to share computations inside lambdas, something that GHC isn't capable of. For example, consider the following reduction:

```
dup f g = ((λx λy (Pair (+ x x) y)) 2)
(Pair (f 10) (g 20))
---------------------------------- App-Lam
dup f g = λy (Pair (+ 2 2) y)
(Pair (f 10) (g 20))
---------------------------------- Dup-Lam
dup f g = (Pair (+ 2 2) {y0 y1})
(Pair (λy0(f) 10) (λy1(g) 20))
---------------------------------- App-Lam
dup f g = (Pair (+ 2 2) {10 y1})
(Pair f (λy1(g) 20))
---------------------------------- App-Lam
dup f g = (Pair (+ 2 2) {10 20})
(Pair f g)
---------------------------------- Dup-Ctr
dup a b = (+ 2 2)
dup c d = {10 20}
(Pair (Pair a c) (Pair b d))
---------------------------------- Op2-U32
dup a b = 4
dup c d = {10 20}
(Pair (Pair a c) (Pair b d))
---------------------------------- Dup-U32
dup c d = {10 20}
(Pair (Pair 4 c) (Pair 4 d))
---------------------------------- Dup-sup
(Pair (Pair 4 10) (Pair 4 20))
```

Notice that the `(+ 2 2)` addition only happened once, even though it was nested inside two copied lambda binders! In Haskell, this situation would lead to the un-sharing of the lambdas, and `(+ 2 2)` would happen twice. Notice also how, in some steps, lambdas were applied to arguments that appeared outside of their bodies. This is all fine, and, in the end, the result is correct.

Uff. That was hard, wasn't it? The good news is the worst part is done. From now on, nothing too surprising will happen.

Superposed Application ----------

Since duplications are pervasive, it may happen that a superposition will end up in the function position of an application. For example, the situation below can happen at runtime:

```
({λx(x) λy(y)} 10)
```

This represents two superposed lambdas, applied to an argument `10`. If we leave this expression as is, certain programs would be stuck, and we wouldn't be able to evaluate them. We need a way out. Because of that, there is a superposed application rule that deals with that situation:

```
({a b} c)
---------- App-Sup
dup x0 x1 = c
{(a x0) (b x1)}
```

In English, this rule says that: "the application of a superposition `{a b}` to `c` is the superposition of the application of `a` and `b` to copies of `c`". Makes sense, doesn't it? That rule also applies to user-defined functions. The logic is the same, only adapted depending on the arity. I won't show it here.

Superposed Operation ----------

```
(+ {a0 a1} b)
------------- Op2-Sup-A
dup b0 b1 = b
{(+ a0 b0) (+ a1 b1)}


(+ a {b0 b1})
------------- Op2-Sup-B
dup a0 a1 = a
{(+ a0 b0) (+ a1 b1)}
```

This, too, follows the same logic of superposed application, except operators are strict on both arguments.

Superposed Duplication ----------

There is one last rule that is worth discussing.

```
dup x y = {a b}
--------------- DUP-SUP (different)
x <- {xA xB}
y <- {yA yB}
dup xA yA = a
dup xB yB = b
```

This rule handles the duplication of a superposition. In English, it says that: *"the duplication of a superposition `{a b}` as `x` and `y` reduces to the duplication of `a` as `xA` and `yA`, `b` as `xB` and `tB`, and the substitution of `x` by the superposition `{xA xB}`, and the substitution of `y` by `{yA tB}`"*. At that point, the formal notation is probably doing a better job than English at conveying this information.

If you've paid close attention, though, you may have noticed the DUP-SUP has already been defined, on the *Lambda Application* section. So, what is going on here? Well, it turns out that DUP-SUP is a special case that has two different reduction rules. If this DUP-SUP represents the end of a duplication process, it must go with the former rule. However, if you're duplicating a term, which itself duplicates something, then this rule must be used. Due to the extremely local nature of HVM reductions though, determining when each rule should be used in general would require an expensive book-keeping machinery. To avoid that extra cost, HVM instead placed a limitation that allowed for a much faster decision procedure. That limitation is:

If a lambda that clones its argument is itself cloned, then its clones aren't allowed to clone each-other.

For example, this term is **not** allowed:

```
let g = λf(λx(f (f x)))
(g g)
```

That's because `g` is a function that clones its argument (since `f` is used twice). It is then cloned, so each `g` on the second line is a clone. Then the first clone attempts to clone the second clone. That is considered undefined behavior, and a typed language that compiles to HVM must check that this kind of situation won't happen.

How common is this? Well, unless you like multiplying Church-Encoded natural numbers in a loop, you've probably never seen a program that reaches this limitation in your entire career. Even if you're a fan of λ-encodings, you're fine. For example, the program above can be fixed by just avoiding one clone:

```
let g = λf(λx(f (f x)))
let h = λf(λx(f (f x)))
(g h)
```

And all the other "hardcore" functional programming tools are compatible. Y-Combinators, Church-Encodings, nested maps of maps, all work just fine. If you think you'll reach this limitation in practice, you're probably misunderstanding how esotheric a program must be for that to happen. It is a common (and annoying) misconception that this limit is any relevant in practice. C programmers survived without closures, for decades. Rust programmers live well with far more restrictive limitations on what shapes of programs they're allowed to write. HVM has all sorts of extremely high-level closures you can think of. You just can't have a clone clone its own clone. Without this limitation, which is almost irrelevant in practice, it wouldn't be possible for HVM to achieve its current performance, so we believe it is justified.

As a last note, HVM's current implementation is slightly more restrictive than it should be, since each occurrence of a global definition counts as a clone of itself. That is not necessary, and will soon be patched. Regardless, even in this version, it is very unlikely you'll find this in practice.

HVM's Low-level Implementation ==========

TODO: in this section, explain how HVM nodes are stored in memory, how rewrites and reduction works, etc. Since this isn't done yet, feel free to explore it yourself by reading runtime.c (https://github.com/Kindelia/HVM/blob/master/src/runtime.c).

[TODO]

Bonus: Copatterns ==========

Since functions and constructors are treated the same, this means there is nothing preventing us from writing copatterns, by just swapping the roles of eliminators and introducers. That is, for example, consider the program below:

```
// List Map function
(Map f Nil)         = Nil
(Map f (Cons x xs)) = (Cons (f x) (Map f xs))

// List projectors
(Head (Cons x xs)) = x
(Tail (Cons x xs)) = xs

// The infinite list: 0, 1, 2, 3 ...
Nats = (Cons 0 (Map λx(+ x 1) Nats))

// Just a test (returns 2)
Main = (Head (Tail (Tail Nats)))
```

It is an the usual recursive `Map` applied to an infinite `List`. Here, `Map` is used in the function position, and the List constructors (`Nil` and `Cons`) are matched. The same program can be written in a corecursive fashion, by inverting everything: the `List` destructors (`Head` / `Tail`) are used in the function position, and the function `Map` is matched:

```
// CoList Map function
(Head (Map f xs)) = (f (Head xs))
(Tail (Map f xs)) = (Map f (Tail xs))

// The infinite colist: 0, 1, 2, 3 ...
(Head Nats) = 0
(Tail Nats) = (Map λx(+ x 1) Nats)

// Just a test (returns 2)
(Main n) = (Head (Tail (Tail Nats)))
```

Bonus: Abusing Beta-Optimality ==========

By abusing beta-optimality, we're able to turn some exponential-time algorithms in linear-time ones. That is why we're able to implement `Add` on `BitStrings` as repeated increment:

```
// Addition is just "increment N times"
(Add xs ys) = (App xs λx(Inc x) ys)
```

This small, elegant mathematical one-liner is as efficient as the manually-crafted add-with-carry operation, which is an 8-cases, low-level, error-prone definition. In order for this to be possible, we must apply some techniques to make sure the self-composition (`λx (f (f x))`) of the function remais as small as possible. First, we must use λ-encoded algorithms. It we don't, then the normal form will not be small. For example:

```
(Not True)  = False
(Not False) = True
```

This is easy to read, but then `λx (Not (Not x))` will not have a small normal form. If we use λ-encodings, we can write `not` as:

```
True  = λt λf t
False = λt λf f
Not   = λb (b False True)
```

This correctly negates an λ-encoded boolean. But `λx (Not (Not x))` still has a large normal form: `λx (x λtλf(f) λtλf(t) λtλf(f) λtλf(t))`. Now, if we inline the definition of `Not`, we get:

```
True  = λt λf t
False = λt λf f
Not   = λb (b λtλf(f) λtλf(t))
```

Notice how both branches start with the same lambdas? We can lift them up and **share** them:

```
True  = λt λf t
False = λt λf f
Not   = λb λt λf (b f t)
```

This will make the normal form of `λx (Not (Not x))` small: i.e., it becomes `λx λt λf (b t f)`. This makes `Not^(2^N)` linear time in `N`!

The same technique also applies for `Inc`. We start with the usual definition:

```
(Inc E)     = E
(Inc (O x)) = (I x)
(Inc (I x)) = (O (Inc x))
```

Then we make it λ-encoded:

```
(Inc x) =
  let case_e = λe λo λi e
  let case_o = λx λe λo λi (i x)
  let case_i = λx λe λo λi (o (Inc x))
  (x case_e case_o case_i)
```

Then we lift the shared lambdas up:

```
(Inc x) = λe λo λi
  let case_e = e
  let case_o = λx (i x)
  let case_i = λx (o (Inc x))
  (x case_e case_o case_i)
```

This makes `λx (Inc (Inc x))` have a constant-space normal form, which in turn makes the composition of `Inc` fast, allowing `Add` to be efficiently implemented as repeated increment.

Similar uses of this idea can greatly speed-up functional algorithms. For example, a clever way to implement a `Data.List` would be to let all algorithms operate on λ-encoded Church Lists under the hoods, converting as needed. This has the same "deforestation" effect of Haskell's rewrite pragmas, without any hard-coded compile-time rewrite, and in a more flexible way. For example, using `map` in a loop is "deforested" in HVM. GHC can't do that, because the number of applications is not known statically.

Note that too much cloning will often make your normal forms large, so avoid these by keeping your programs linear. For example, instead of:

```
Add = λa λb
  let case_zero = b
  let case_succ = λa_pred (Add a_pred b)
  (a case_succ case_zero)
```

Write:

```
Add = λa
  let case_zero = λb b
  let case_succ = λa_pred λb (Add a_pred b)
  (a case_succ case_zero b)
```

Notice how the later avoids cloning `b` entirely.

Abusing Parallelism ----------

[TODO]

H.47 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-linker.md

HVM3's Optimal Polarized Atomic Linker

URL: https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b (https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b)

FILE: HVM3_ATOMIC_LINKER.MD

HVM3's Optimal Atomic Linker (with Polarization)

Atomic linking is at the heart of HVM's implementation: it is what allows threads to collaborate towards massive parallelism. All major HVM versions started with a better atomic linker. From slow, buggy locks (HVM1), to AtomicCAS (HVM1.5), to AtomicSwap (HVM2), the algorithm became simpler and faster over the years.

On the initial HVM3 implementation, I noticed that one of the cases on the atomic linker never happened. After some reasoning, I now understand why, and that information can be used to greatly simplify the algorithm. The result, I believe, is, for the first time, an "Optimal Atomic Linker", in the sense it can be performed with the minimal amount of CPU instructions. While the changes look small, I believe this optimal shape will result in significant improvements in performance, as it is the most frequent operation when evaluating HVM.

The main insight is that we now treat negative and positive terms differently. On interactions, instead of taking both (i.e., swapping by 0), we take only the positive term. Then, we 'move' it into the *location* of the negative term, which will then call 'link' if needed. I'll document the algorithm below, and provide a very informal proof.

MEMORY LAYOUT

Before we start, let's explain the polarized memory layout. It is basically the same as HVM2 (https://docs.google.com/viewer?url=https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf), except terms must always obey polarization. For example, the following Bend program:

```
Main = λt(t (λx(x) λy(y)))
```

Would be represented as:

```
@main
  = +(-(+c -r) +r)
  & -(+(-y +y) -c) ~ +(-x +x)
```

Note that the root is always a positive term, redexes are always stored as `-negative ~ +positive`, and two ends of a wire always have opposing polarities. Interaction Calculus (https://github.com/VictorTaelin/Interaction-Calculus) terms are represented as follows:

- Lam: `+(-var +bod)`
- App: `-(+arg -ret)`
- Sup: `+{+tm1 +tm2}`
- Dup: `-{-dp1 -dp2}`

This also means that variables are always paired. The negative variable is placed at the binder's port, and the positive variable is placed at any other location of the program. For example, if we were to add polarity annotations to vars in λ-terms, the CNat 1 would look like this: `λ-f. λ-x. (+f +x)`.

OPTIMAL POLARIZED ATOMIC LINKER

Let's first define a simple Net representation in pseudo-Haskell, using a map of terms, similar to HVM2:

```
-- A Net is just a map of Locations to Terms
type Net = Map Location Term

-- A Location is just an integer
type Location = U64

-- A Term holds a tag and the location of its children
type Term = (Tag, Location)

-- A Tag specifies the Term's node type
data Tag
  = VAR | SUB -- pos/neg Var (aux wire)
  | NUL | ERA -- pos/neg Eraser Node
  | LAM | APP -- pos/neg Constructor Node
  | SUP | DUP -- pos/neg Duplicator Node
```

Let's also define a **valid net** as one that is structured with correct polarities. That means every variable has a positive and negative occurrence in the net. To implement the atomic linker, we must store, on every positive variable, the location of the negative variable. The negative variable doesn't need to store anything (its 'loc' field can be just 0).

An interaction is performed as before, except we 'take' only positive terms:

```
-- The beta-reduction (APP-LAM interaction)
-- - neg_loc: the location of the negative node
-- - pos_loc: the location of the positive node
applam :: Location -> Location -> HVM ()
applam neg_loc pos_loc = do

  -- 1. Compute the location of each child term
  let arg_loc = neg_loc + 1
  let ret_loc = neg_loc + 2
  let var_loc = pos_loc + 1
  let bod_loc = pos_loc + 2

  -- 2. Take the positive terms
  arg_val <- swap arg_loc 0
  bod_val <- swap bod_loc 0

  -- 3. Move positive terms into negative locations
  move var_loc arg_val
  move ret_loc bod_val
```

That's important: if we took the negative terms, we would need to handle their absence, making the logic more complex. To perform a link, we then 'move' the positive term (which can be either a variable or a node) into the negative term's location:

```
-- Moves a positive term (var or node) into a negative location
move : Location -> Term -> HVM ()
move neg_loc pos = do
  neg <- swap neg_loc pos
  if getTag neg == SUB
  then do
    return ()
  else do
    link neg pos
```

We start by atomically swapping the negative term's location by the positive term, retrieving a negative term (since the net is valid). Then, there are two options:

If the retrieved term is a negative node, we *link* it to the positive term. (See below.)

If the retrieved term is a negative var, we do nothing. Note that this means the positive term will be left in the negative location, temporarily violating the valid-net condition. That's fine: we consider that this location has been casted to a "substitution map entry" - i.e., it is not technically part of the net anymore. We are always able to disambiguate, because the only other part of the net that can point to this location is the positive counterpart of that var. Yet, positive vars can't point to positive terms. So, when that happens, it means that location is a substitution map entry.

Now, the atomic linker itself looks like this:

```
-- Links a negative node with a positive term (var or node)
link : Term -> Term -> HVM ()
link neg pos = do
  if getTag pos == VAR
  then do
    neg_var <- swap (getLoc pos) neg
    if getTag neg_var == SUB
    then do
      return ()
    else do
      move (getLoc pos) neg_var
  else do
    pushRedex neg pos
    return ()
```

Note this function should only be called with a negative *node*, never a negative *var*.

It starts by checking if the positive term is a var. If so, we swap that positive var's target (i.e., the location where its negative var counterpart is supposed to be) by the negative node. Then, there are two cases:

If the term we retrieved is, indeed, the corresponding negative var, then, good job: we just performed a cross-thread substitution without conflicts. The negative node is now stored on the binding λ's location, and both occurrences of that var are completely gone. We don't need to do anything else!

Now, if the retrieved term is NOT the corresponding negative var, then, what could it be? Remember: vars always point to each-other, but there is one exception: when a location has been "casted" to a substition map entry. That only happens when we place a positive term in a negative location. That means the term we retrieved is that positive term (var or node). We need to do something with it, otherwise it will disappear! Thankfully, we know the negative node will be now placed on that location. So, to complete the linking, all we need to do is *move* the positive into that negative location. Interestingly, this means information has been exchanged between two threads. That single, innocent line of code elegantly handles every synchronization conflict that could ever occur, liberating us from race conditions once for all!

Finally, if the positive term was a node, that means we have formed a new active pair. We push it to the bag and return.

And that's it! This completes the algorithm and proof.

CONCLUSION AND NOTES

Making it that simple took years of efforts, from versions using lock (HVM1: slow, buggy, infeasible on GPUs), to better versions using atomic compare-and-swap (HVM1.5: worked on GPUs, but slow), to better, but still imperfect versions using just atomic swap (HVM2: worked on GPUs, fast, but has some limitations), to this one exploiting polarities, which, as far as I can tell, is optimal, in the sense it requires the least amount of CPU instructions, and has maximum expressivity.

For example, this new format allows for lazy evaluation, since positive vars point to the negative variable location, allowing us to traverse the graph with the good-old lazy evaluation algorithm; this wasn't possible on HVM2. It also allows us to dispense the "subst map" that we had on HVM2, since we reuse the negative binder locations to perform substitutions, effectively using 1/3 less memory. It also allows other nice savings, like not needing to store anything on negative vars, which simplifies the injection/readback algorithms, and so on.

I also like how polarity clears up the connection between Interaction Combinators and the Interaction Calculus I proposed: the distinction between lams/dups and sups/dups is just their polarity, and there *is* a reason to use it on implementations. Note that we could omit the polarity tags (i.e., have just CON instead of LAM/APP) since it is inferrable (as long as we keep the memory organized as to respect polarization).

Finally, keep in mind all this polarity stuff has been known and discussed for a long time, but, for whatever reason, I never realized it could be relevant to fast implementations. Also, as T6 pointed in our Discord, perhaps polarization isn't even needed to implement the algorithm above. He might be right, but it did help me to arrive at it. He also pointed how polarity-based linking might result in longest redirection chains, but I believe this can be addressed outside.

For completion, here is `duplam` in HVM3's setup:

```
duplam :: Location -> Location -> HVM ()
duplam neg_loc pos_loc = do

  -- 1. Compute the location of each child term
  let dp1_loc = neg_loc + 1
  let dp2_loc = neg_loc + 2
  let var_loc = pos_loc + 1
  let bod_loc = pos_loc + 2

  -- 2. Take the positive terms
  bod_val <- swap bod_loc 0

  -- 3. Allocate new nodes
  co1_loc <- alloc 3
  co2_loc <- alloc 3
  du1_loc <- alloc 3
  du2_loc <- alloc 3

  -- 4. Fill the new nodes
  set (co1_loc + 1) (SUB, co1_loc + 1)
  set (co1_loc + 2) (VAR, du2_loc + 1)
  set (co2_loc + 1) (SUB, co2_loc + 1)
  set (co2_loc + 2) (VAR, du2_loc + 2)
  set (du1_loc + 1) (VAR, co1_loc + 1)
  set (du1_loc + 2) (VAR, co2_loc + 1)
  set (du2_loc + 1) (SUB, du2_loc + 1)
  set (du2_loc + 2) (SUB, du2_loc + 2)

  -- 5. Move positive terms into negative locations
  move dp1_loc (LAM, co1_loc)
  move dp2_loc (LAM, co2_loc)
  move var_loc (SUP, du1_loc)
  link (DUP, du2) bod_val
```

Again, it is almost the same as the HVM2 version, with the subtle difference that we only take positive terms, and move them into negative locations.

Here's (https://gist.github.com/VictorTaelin/997260bc68dc35acb82960051176daf4) a snapshot of the complete implementation using Kind-Lang (https://github.com/HigherOrderCO/Kind). To run it, you need to download the KindBook (https://github.com/HigherOrderCO/kindbook) library. There is also a standalone C implementation (https://gist.github.com/VictorTaelin/27e1bce6f5c8d2f89aafef75025b4765).

H.48 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-mystery.md

Solving the mystery behind Abstract Algorithm's magical optimizations

URL: https://gist.github.com/VictorTaelin/85b94b5ba5b8b5440ded64bba8c89ac2 (https://gist.github.com/VictorTaelin/85b94b5ba5b8b5440ded64bba8c89ac2)

FILE: SOLVING_THE_MYSTERY.MD

Note: This is an old post from back when I was trying to make sense of why inets are so fast for evaluating some λ-terms. It has some silly bits, I learned a lot since and could probably write a better article today, but I think this can still be insightful for these getting started, so I'll leave it here.

Yesterday, I reported (https://medium.com/@maiavictor/some-functions-may-have-negative-complexity-im-worried-for-my-crypto-c53f6e7343d3) the bizarre observation that certain functions can behave as if they had negative complexity. If you haven't checked that article yet, it isn't necessary, but you should, as it may blow your mind. In short, the λ-term `f(bits) = copy(comp(inc,n,bits))`, when given to optimal λ-calculus evaluator, is asymptotically faster than `g(bits) = comp(inc,n,bits)`; i.e., `copy` (a `O(1)` operation for a fixed size) behaves as if it had a `O(1/n)` complexity, causing the program to run faster by doing more things (!?).

That's not the only bizarre complexity result I had when experimenting with the Abstract Algorithm. Years ago, I noticed (https://stackoverflow.com/questions/31707614/why-are-%CE%BB-calculus-optimal-evaluators-able-to-compute-big-modular-exponentiation) that it is capable of computing programs that wouldn't fit in a computer the size of the universe. Recently, I posted how it can apply a `O(1)` operation `N` times in `[O(log(N))` time (https://github.com/VictorTaelin/articles/tree/master/0004-supercompilation-for-free) (!?). Despite insightful answers being provided (https://stackoverflow.com/questions/31707614/why-are-%CE%BB-calculus-optimal-evaluators-able-to-compute-big-modular-exponentiation), it was hard to make sense of such unintuitive behaviors.

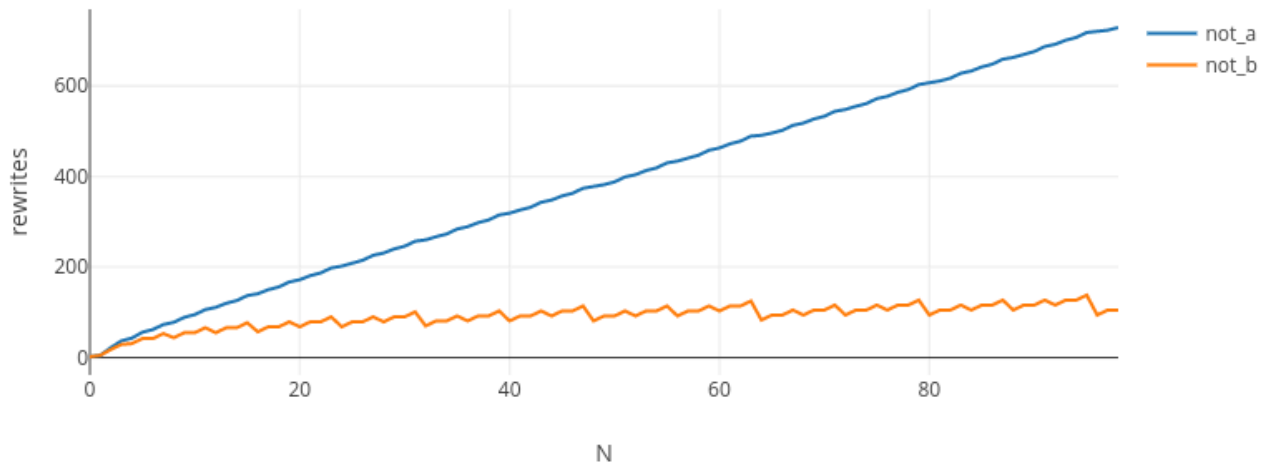
After a lot of research and some recent insights, it finally hit me. Turns out the explanation for all of that is pretty simple, and, in this article, I'll explain what is going on carefully (not jokingly this time). If you're in a hurry, feel free to skip to the tl;dr at the end.

1. Optimizable functions are those that fuse when self-composed ----------

First, let's start with the simplest example on which something interesting happens: repeated `not`. Let's implement it in two different ways:

```
True  = λt. λf. t
False = λt. λf. f
not\_a = λb. (b False True)
not\_b = λb. λt. λf. (b f t)
```

If you don't understand those definitions, you can read about lambda encodings (https://www.cs.umd.edu/class/fall2017/cmsc330/lectures/02-lambda-calc-encodings.pdf), but that's not too important. Now, how would consecutive applications of `not_a` and `not_b` perform in Absal (https://github.com/VictorTaelin/abstract-algorithm) (an implementation of the **Abs**tract **Al**gorithm)? Here is a chart:



Complexity (total rewrites) of applying "not" N times to True

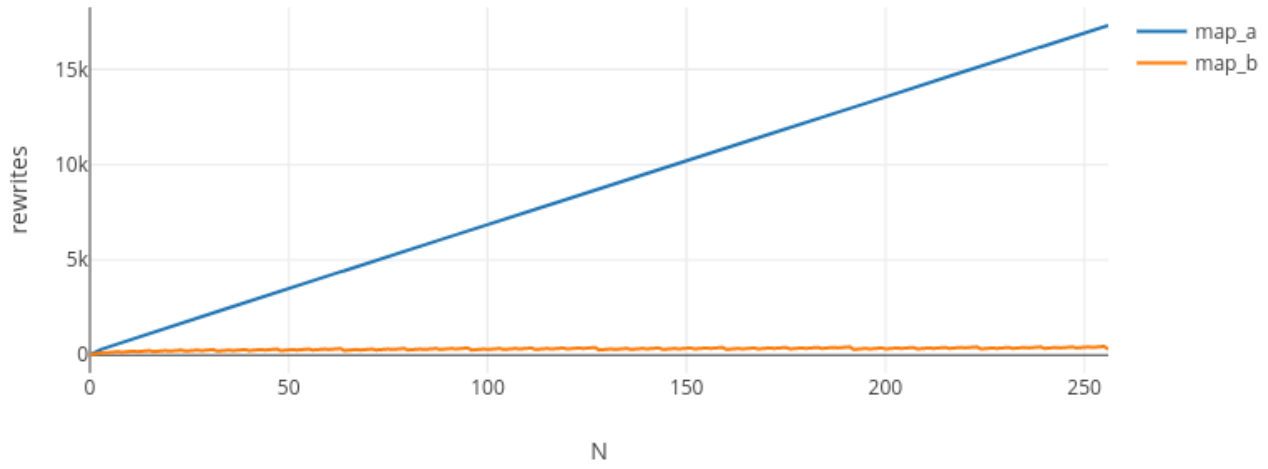
Here, complexity is measured as graph rewrites. Don't worry about that, though, just assume the time it takes to run a λ-term is proportional to the number of graph rewrites. As you can see, **`not_a` performs linearly, while `not_b` performs logarithmically**. That's a huge difference, for such a small change! And is also very unintuitive, because you shouldn't be able to apply `N` `not`s in `O(log(N))`. Then why that happens? Before answering, let's first make an experiment. Let's compose both functions with themselves and check their normal forms.

- norm(`not_a.not_a`): `λb. (b (λt.λf.f) (λt.λf.t) (λt.λf.f) (λt.λf.t))`
- norm(`not_b.not_b`): `λb. λt. λf. (b t f)`

Interesting: once self-composed, `not_a` grows larger, but `not_b` stays the same size. Why? Because the later function **fused**. That's the same effect that allows Haskell to turn several passes over a list in a single pass, eliminating intermediate structures. This leads us to hypothesize that the magical speedup could have something to do with that property. Let's check that hypothesis by testing the `map` operation on lists:

```
Nil   =         λc. λn. n
Cons  = λh. λt. λc. λn. c h (t c n)
map\_a = λf. λl.         l (λh. λt. Cons (f h) t) Nil
map\_b = λf. λl. λc. λn. l (λh. λt. c    (f h) t) n
list  = Cons True (Cons False (Cons True (Cons False Nil)))
```

Here, we once again defined it in two different ways: `map_b` fuses and `map_a` doesn't. Let's measure the complexity of applying `map` repeatedly to a list:



Complexity (total rewrites) of applying "map" N times to a list of 4 elements

As you can see, the same behavior is observed here: the version that fuses performs logarithmically, while the version that doesn't performs linearly, supporting our hypothesis. Woa! But why would that happen?

(Here `(https://gist.github.com/VictorTaelin/2a1914bf827ae4c20b9e26d0c708ed99)` is the code for reproducing that experiment and generate the charts.)

2. Exponentiation by squaring = composition by sharing ----------

To understand that, let's first talk about integer exponentiation. If you're a developer, chances are you've heard about exponentiation by squaring `(https://en.wikipedia.org/wiki/Exponentiation_by_squaring)`. It is an ancient, simple algorithm capable of computing `A^B` in much less steps than it'd take to multiply `A` repeatedly, `B` times. To refresh our minds, let's visualize how it works:

```
\# Computing 13^8 with repeated multiplication
13^2 = 13 \* 13 = 169
13^3 = 13 \* 169 = 2197
13^4 = 13 \* 2197 = 28561
13^5 = 13 \* 28561 = 371293
13^6 = 13 \* 371293 = 4826809
13^7 = 13 \* 4826809 = 62748517
13^8 = 13 \* 62748517 = 815730721\

# Computing 13^8 with exponentiation by squaring
13^2 = 13 \* 13 = 169
13^4 = 169 \* 169 = 28561
13^8 = 28561 \* 28561 = 815730721
```

As you can see, the first approach multiplies 13 one-by-one, needing `N` multiplications to reach `13^N` (linear time), while the second repeatedly squares the first number instead, allowing it to jump from `13^1` to `13^2` to `13^4` to `13^8` in a single multiplication for each jump (logarithmic time). To compute values that aren't perfect squares, you just multiply sub-terms, i.e., `13^11 = 13^8 * 13^2 * 13^1`.

Neat, right? But how is that relevant? Due to a less known fact, for any function that can be fused by composition, we can also compute its repeated application in `O(log(N))` steps using that same trick! Let's visualize it too:

```
\# Computing not^8(t) with repeated application
not^2(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b f t) = (λb.λf.λt. b t f)
not^3(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b t f) = (λb.λf.λt. b f t)
not^4(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b f t) = (λb.λf.λt. b t f)
not^5(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b t f) = (λb.λf.λt. b f t)
not^6(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b f t) = (λb.λf.λt. b t f)
not^7(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b t f) = (λb.λf.λt. b f t)
not^8(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b f t) = (λb.λf.λt. b t f)

\# Computing not^8(t) with composition by sharing
not^2(t) = (λb.λf.λt. b f t) . (λb.λf.λt. b f t) = (λb.λf.λt. b t f)
not^4(t) = (λb.λf.λt. b t f) . (λb.λf.λt. b t f) = (λb.λf.λt. b t f)
not^8(t) = (λb.λf.λt. b t f) . (λb.λf.λt. b t f) = (λb.λf.λt. b t f)
```

Ok, that is way too hard to read. Let's replace lambdas by their names:

```
\# Computing not^8(x) with repeated application
not^2(x) = not . not = id
not^3(x) = not . id  = not
not^4(x) = not . not = id
not^5(x) = not . id  = not
not^6(x) = not . not = id
not^7(x) = not . id  = not
not^8(x) = not . not = id

\# Computing not^8(x) with composition by sharing
not^2(x) = not . not = id
not^4(x) = id  . id  = id
not^8(x) = id  . id  = id
```

Can you see it? The first approach applies `not` s one by one, needing `N` applications to reach the final result (linear time), while the second repeatedly self-composes the initial function, jumping from `not^2` to `not^4` to `not^8` with just a single composition for each jump (logarithmic time). In other words, **as long as the self-composition of `**f**` fuses, we can implement repeated application in logarithmic time, using the same trick behind exponentiation by squaring**!

3. Absal performs runtime fusion ---------------------------------

Sadly, knowing that trick isn't sufficient to speed-up fusible algorithms: we also need a functional language capable of performing short-cut fusion during runtime. Haskell, for one, isn't capable of doing that for native functions. To illustrate, let's implement "composition by sharing" in it:

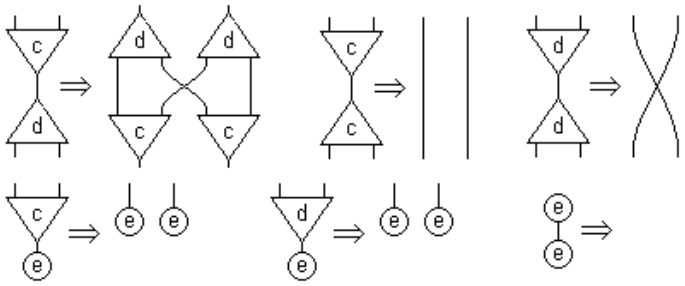
```
comp :: Int -> (a -> a) -> (a -> a)
comp 0 f x = x
comp n f x = comp (n `div` 2) g (r x)
  where  g = f . f
         r = if even n then id else f
```

This function takes a number `n`, a function `f`, and returns a new function that applies `f` repeatedly to `x`, `n` times, using the "composition by sharing" method. As an example, (`comp(not,10,x)`) evaluates to the equivalent of:

```
not_10_times :: Bool -> Bool
not_10_times x =
  let f1 = not in
  let f2 = f1 . f1 in
  let f4 = f2 . f2 in
  let f8 = f4 . f4 in
  f8 (f2 x)
```

Notice how this function applies `not` 10 times to its input, even though it only uses composition 3 times? That's why it should be efficient. So, what is the complexity of `comp(not, N, x)` in Haskell? Not logarithmic as we'd expect, but linear! You can test that fact by running this (https://gist.github.com/VictorTaelin/bc0e8b38f93130a083b8de293d9cc7f5) Gist. If you change `2^20` to something higher such as `2^40`, it will run out of memory. Why? Because it is too lazy and never evaluates inside lambdas, which would be necessary to fuse those intermediate expressions. And, even if it did, it'd still not work. This (https://gist.github.com/VictorTaelin/053b3a608e633b826cfd79bc749d9cdf) separate posts investigates why, examining Haskell and JavaScript's evaluation model and why they're not optimal.

Now we're real close to solving our puzzle. But, first, we need to talk about parallel universes; I mean, parallel reductions. The main insight behind Absal is the use of interaction nets instead of simpler textual representations. Interaction nets are a model of computation based on graphs with labelled ports (one being the "main port") and a very peculiar caveat: whenever two "main ports" met, the nodes connected by them are rewritten.

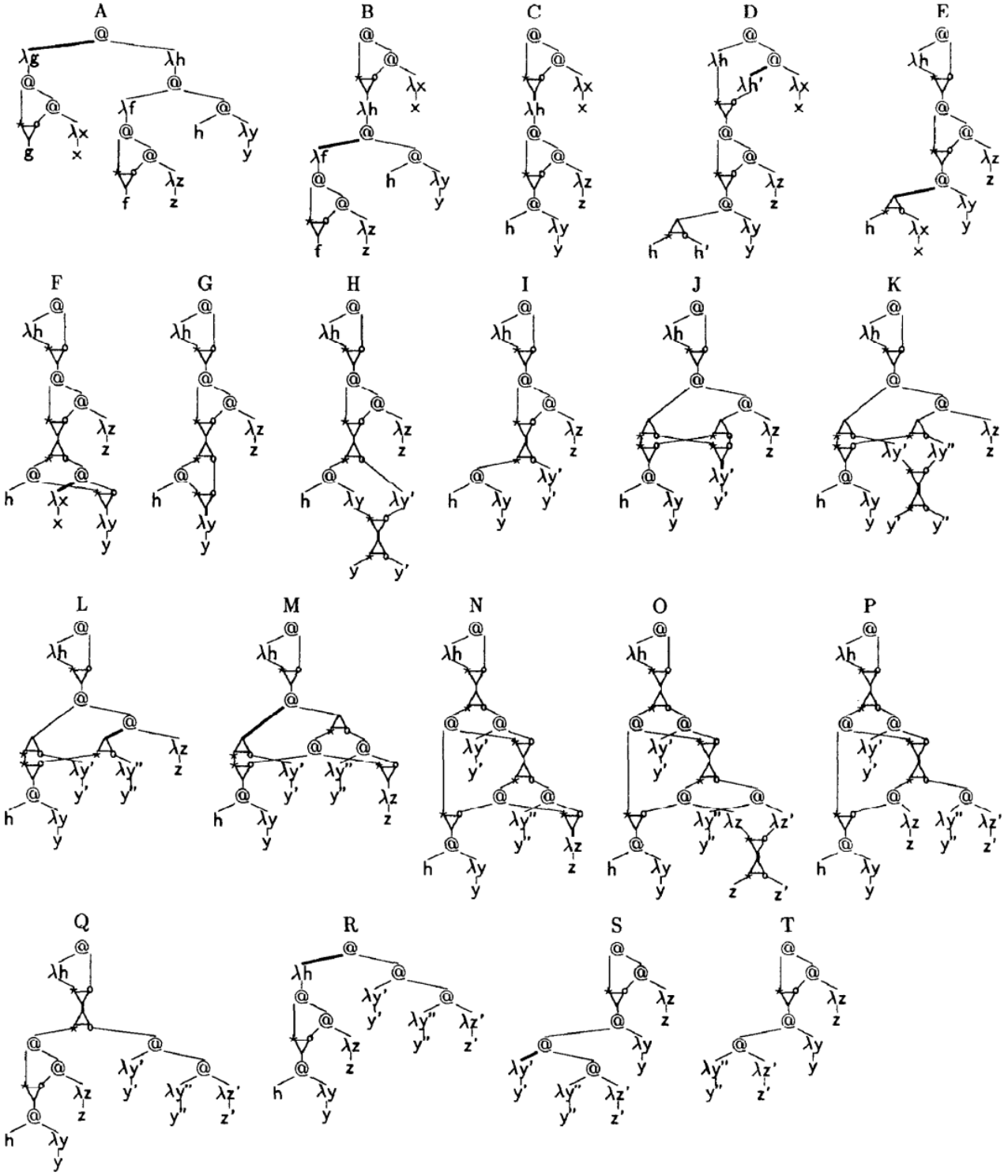


Rewrite rules of interaction combinators, a type of interaction net

That causes a very interesting effect: no matter what order you rewrite your nodes, the total number of steps stays constant. If, thus, you compile λ-terms to interaction nets, then you can reduce expressions in any order: functions first, arguments first, **even in parallel**. In the end, the performed work will be the same!

The trick, of course, is how to properly translate λ-terms to interaction nets. Sadly, there is no solution to that covers all λ-terms efficiently. But, for a huge class of them (probably whatever you'd write in practice), there is a simple and elegant way to do it: that's the Abstract Algorithm. To explain it shortly, each lambda and each application of a λ-term becomes a ternary node on the net, and each repeated occurrence of a variable becomes a "fan node". Also, the main port of an application node points to the function port, which is why redexes can never be duplicated.

If you're overwhelmed, don't worry. The main insight is that, by being free to reduce terms in whatever order it wants, **Absal is capable of being as eager as possible without any performance penalty, which, in turn, allows it to fuse intermediate compositions. This is exactly what is needed to exploit the "composition by sharing" trick**. If you do want to have a better understanding of the Abstract Algorithm, I recommend reading this `(https://www.amazon.com/Implementation-Functional-Programming-Languages-Theoretical/dp/0521621127)` book, it is extremely pedagogical. Alternatively, you can just contemplate the image below:



It depicts the evaluation of the `(λg. g (g λx. x)) (λh. (λf. f (f λz. z)) (h λy. y))` term on the abstract algorithm. Notice how sub-expressions can be shared by several terms, how we can evaluate inside lambdas without ever risking duplicating work (and while still being able to be lazy), and how a single constant-time beta-reduction in this format is equivalent to several beta-reductions on more traditional representations. Beautiful, no?

But what about the negative complexity stuff? ==========

Let's recap all that we have learned so far. First, some λ-calculus functions fuse when they are self composed. By "fuse", I mean the size of their normal forms stay constant. Second, those functions can be applied `N` times with `O(log(N))` complexity using the "composition by sharing" strategy, as long as the host language can eagerly normalize sub-expressions inside lambdas. Third, Absal is capable of evaluating inside lambdas, which allows it to do exactly that, fusing inner compositions during runtime. That leads us to one logical conclusion: `copy` must somehow cause `inc` to fuse! Wow! Could it be? Let's test that hypothesis by looking normal forms!

```
-- Normal form of inc:
λa. λb. λc. λd. (((a c) λe. (b λf. λg. λh. (((e g) λi. (f λj.
λk. λl. (((i k) λm. (j m)) l))) h))) d)

-- Normal form of inc . inc:
λa. λb. λc. λd. (((a λe. (b λf. λg. λh. (((e g) λi. (f λj. λk
. λl. (((i k) λm. (j m)) l))) h))) λe. (c λf. λg. λh. (((e g)
λi. (f λj. λk. λl. (((i k) λm. (j m)) l))) h))) d)

-- Normal form of inc . inc . inc:
λa. λb. λc. λd. (((a λe. (c λf. λg. λh. (((e g) λi. (f λj. λk
. λl. (((i k) λm. (j m)) l))) h))) λe. (b λf. λg. λh. (((e λi
. (f λj. λk. λl. (((i k) λm. (j m)) l))) λi. (g λj. λk. λl. (
((i k) λm. (j m)) l))) h))) d)

------------------------------------------------------------

-- Normal form of copy . inc:
λa. (((a λb. λc. λd. λe. (d (((b λf. λg. λh. λi. (g (((f λj.
λk. λl. λm. (k j)) λj. λk. λl. λm. (l j)) λj. λk. λl. l))) λf
. λg. λh. λi. (h (((f λj. λk. λl. λm. (k j)) λj. λk. λl. λm.
(l j)) λj. λk. λl. l))) λf. λg. λh. h))) λb. λc. λd. λe. (c (
((b λf. λg. λh. λi. (h (((f λj. λk. λl. λm. (k j)) λj. λk. λl
. λm. (l j)) λj. λk. λl. l))) λf. λg. λh. λi. (g (((f λj. λk.
λl. λm. (l j)) λj. λk. λl. λm. (k j)) λj. λk. λl. l))) λf. λg
. λh. h))) λb. λc. λd. d)

-- Normal form of copy . inc . inc:
λa. (((a λb. λc. λd. λe. (c (((b λf. λg. λh. λi. (h (((f λj.
λk. λl. λm. (k j)) λj. λk. λl. λm. (l j)) λj. λk. λl. l))) λf
. λg. λh. λi. (g (((f λj. λk. λl. λm. (l j)) λj. λk. λl. λm.
(k j)) λj. λk. λl. l))) λf. λg. λh. h))) λb. λc. λd. λe. (d (
((b λf. λg. λh. λi. (h (((f λj. λk. λl. λm. (k j)) λj. λk. λl
. λm. (l j)) λj. λk. λl. l))) λf. λg. λh. λi. (g (((f λj. λk.
λl. λm. (l j)) λj. λk. λl. λm. (k j)) λj. λk. λl. l))) λf. λg
. λh. h))) λb. λc. λd. d)

-- Normal form of copy . inc . inc . inc:
λa. (((a λb. λc. λd. λe. (d (((b λf. λg. λh. λi. (h (((f λj.
λk. λl. λm. (k j)) λj. λk. λl. λm. (l j)) λj. λk. λl. l))) λf
. λg. λh. λi. (g (((f λj. λk. λl. λm. (l j)) λj. λk. λl. λm.
(k j)) λj. λk. λl. l))) λf. λg. λh. h))) λb. λc. λd. λe. (c (
((b λf. λg. λh. λi. (g (((f λj. λk. λl. λm. (l j)) λj. λk. λl
. λm. (k j)) λj. λk. λl. l))) λf. λg. λh. λi. (h (((f λj. λk.
λl. λm. (l j)) λj. λk. λl. λm. (k j)) λj. λk. λl. l))) λf. λg
. λh. h))) λb. λc. λd. d)

-- Note: using a fixed depth of 3 for better visualization
```

There it is! That's exactly what happens. Don't worry about the contents of those terms, we're just considering their sizes. Here, the composition of `inc` with itself is small, but as we keep appending more `inc.` s, the normal form grows bigger, linearly. In other words, `inc` does not fuse with itself. If we just append `copy.` to the sequence of `inc.` s, then the normal form of the resulting function is larger, but stays constant, regardless of the number of compositions. In other words, its self-composition merely permutes variables and, thus, fuses. That, in turn, allows the "composition by sharing" effect to kick off, which causes the `N` th composition of `inc` to have logarithmic complexity. Feels good, `inc.` !

And that's it. Mystery solved. Here `(https://gist.github.com/VictorTaelin/25da5d76f4be0c9ccf7e9c9c3e5693d7)` is the code for replicating this experiment with Absal (see my previous `(https://medium.com/@maiavictor/some-functions-may-have-negative-complexity-im-worried-for-my-crypto-c53f6e7343d3)` post for install instructions).

A last pending question would be: why `copy` causes `inc` to fuse? But that's not hard to understand. First, let's visualize the definitions of `copy` and `inc` :

```
inc = λd. λbits.

  -- Folds over d (the recursion depth)
  d

    -- Succ branch of fold
    λrec. λbits.

      -- Return constructors in advance (important for fusion)
      λB0. λB1. λEnd.

        -- Pattern matches bits
        bits

          -- B0 case
          B1

          -- B1 case
          λxs. B0 (rec xs)

          -- End caase
          End

    -- Zero branch of fold
    λxs. xs

    -- Applies resulting function to bits
    bits

copy = λd.

  -- Folds over d (the recursion depth)
  d

    -- Succ branch of fold
    λrec. λxs.

      -- Pattern matches xs
      xs

        -- B0 case
        λxs. B0 (R xs)

        -- B1 case
        λxs. B1 (R xs)

        -- End case
        End

    -- Zero branch of fold
    λxs. xs
```

Now, let's fix a depth `d=2` and visualize what `inc` becomes when composed with `copy`:

```
inc = λbits.
 -- Pattern matches bits
 bits

   -- B0 case
   λbits. λB0. λB1. λEnd.

     -- Returns B1
     B1

       -- Pattern matches bits
       (bits

         -- B0 case (returns B0)
         λbits. λB0. λB1. λEnd. (B0 bs)

         -- B1 case (returns B1)
         λbits. λB0. λB1. λEnd. (B1 bs)

         -- End case
         λB0. λB1. λEnd. End)

   -- B1 case
   λbits. λB0. λB1. λEnd.

     -- Returns B0
     B0

       -- Pattern matches bits
       (bits

         -- B0 case (returns B0)
         λbits. λB0. λB1. λEnd. (B0 bs)

         -- B1 case (returns B1)
         λbits. λB0. λB1. λEnd. (B1 bs)

         -- End case
         λB0. λB1. λEnd. End)

   -- End case
   λB0. λB1. λEnd. End
```

Turns out `copy` sucked `inc` into itself, feeding it a bunch of constructors, which transformed it into a function that exhaustively pattern-matches all branches of its input to the maximum depth, instead of prematurely returning when no more work is needed, as it did before. Notice how line 16 on the first snippet was replaced by lines 8-21 on the second one. In other words, `inc` essentially became a big permutation / lookup table instead of a normal recursive function. That sounds like a bad idea, but turns out this complete coverage is exactly what was needed for the function to be able to fuse with itself. Without it, the symbolic variables that are prematurely returned in certain branches couldn't be reduced further, accumulating in self-composed terms, causing them to grow larger and larger. Or, alternatively, the additional pattern matches self-destroy under self-composition, keeping the function's body small.

Conclusion ----------

Well, that was quite a trip. Turns out that the Abstract Algorithm had simple explanations behind its magical results. I'm glad I finally figured those things out, and thankful to the Redditors who commented with insights and speculations; in special, /u/matt-noonan `(https://www.reddit.com/user/matt-noonan)`, who mentioned the composition by sharing trick which put all pieces together in my head.

tl;dr -----

In Absal, an optimal implementation of functional languages, repeated application is performed with church-numbers, which are represented by functions with a high amount of internal sharing. For example, 21 is represented as:

```
λf1. λx.
 let f2  = f1 . f1 in
 let f4  = f2 . f2 in
 let f8  = f4 . f4 in
 let f16 = f8 . f8 in
 f16 (f4 (f1 x))
```

That format is very similar to “exponentiation by squaring”, a classic algorithm capable of computing the exponentiation of `A^B` in `O(log(N))` instead of `O(N)`. That alone isn’t sufficient to explain the phenomena; in Haskell, for example, using that function gives no speedups. Absal, though, is extremely eager (despite suffering no slowdowns for it), which makes it capable of computing the normal form of intermediate `f . f` compositions, essentially performing fusion at runtime. That’s only useful when `f` fuses when self-composed, otherwise the growth of its normal form would cause an exponential slowdown due to the cost of copying increasingly larger terms.

If `f` fuses under self-composition, though, then Absal manifests a “composition by sharing” effect, which is similar to “exponentiation by squaring”, except with functions and composition instead of integers and multiplication. This is what allows it to compute `N` repeated applications of `f` in `O(log(N))` time. Finally, turns out `copy . inc` transforms `inc` into a function that self-fuses, by replacing symbols that accumulate by lambdas that self-destroy under self-composition, which is why it is asymptotically faster than `inc` alone.

tl;dr tl;dr ----------

Magic.

H.49 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-sat.md

Simple SAT Solver via superpositions

URL: https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615 (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615)

FILE: SAT.MD

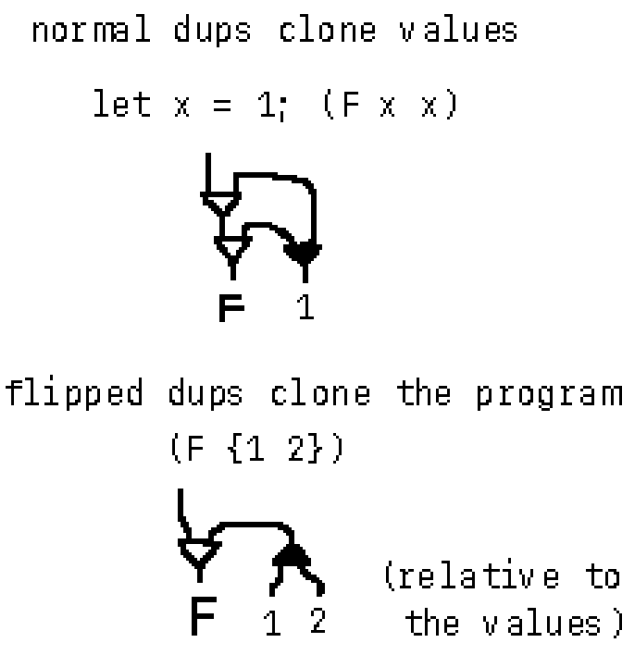
Solving SAT via interaction net superpositions

I’ve recently been amazed, if not mind-blown, by how a very simple, ”one-line” SAT solver on Interaction Nets can outperform brute-force by orders of magnitude by exploiting ”superposed booleans” and optimal evaluation of λ-expressions. In this brief note, I’ll provide some background for you to understand how this works, and then I’ll present a simple code you can run in your own computer to observe and replicate this effect. Note this is a new observation, so I know little about how this algorithm behaves asymptotically, but I find it quite interesting that this simple approach performs much better than it should.

WHAT ARE INTERACTION NET SUPERPOSITIONS?

Interaction Nets are a new model of computation capable of evaluating lambdas optimally. HVM (https://github.com/HigherOrderCO) is a practical implementation of that system. Since INets are fully linear, variables must be ”cloned” via ”dup-nodes”, which can be seen as ”lazy cloners”. Usually, these can only be linked to variables, and are ”invisible” to the user. But what happens if we expose dup nodes as a first-class feature of the user-facing language?

Turns out that they behave like ”superposed values”, which can be seen as a new programming feature that semantically means ”this expression has two values at the same time”. In HVM, the superposition of `X` and `Y` is written as `{X Y}`. When you apply a function to a superposed value, as in, `F({X Y})`, `F` will be executed for each input, `X` and `Y`. So, for example, `double({2 3})` computes to `{4 6}`. What makes this interesting, though, is that these calls are ”overlapped”, in a way that *shares computations*, giving you a huge performance boost.



WHAT ARE ITS APPLICATIONS?

1. FAST FUNCTION CALLS

When we apply a function to a tree of superposed values, it will behave like a "map", and be applied to each value. For example, if `F(x) = x * 10`, then `F({1 2 3 4})` is seen as `{F(1) F(2) F(3) F(4)}`, which reduces to `{10 20 30 40}`, with the advantage of computations being shared optimally across different calls. For example, suppose that `F` has a heavy computation inside it:

```
def F(x):
    return x + slow(999999)
```

Here, `slow(999999)` would take a long time to compute. Yet, when `F` is applied to superposed values, that sub-expression would only be computed once, saving a lot of work. This is a very general behavior. For example, sub-expressions that depend on the inputs (and, thus, can't be computed once) will still be "partly computed only once", as much as possible, making your program much faster.

2. AUTOMATED VECTORIZATION

Another application is automated vectorization. If we compile eliminators and destructors of datatypes to lambdas with different labels, these behave like superpositions across different types. As such, if we evaluate, say, `(Not (Pair True False))`, it will reduce to `(Pair False True)`. The `Not` eliminator "passes through" the `Pair` constructor, and only interacts with the bools.

3. SEARCH

Now, for the most interesting part; and what made me hyped enough to make this post; we can also use superposition to **search**. For example, if we evaluate `(And {True False} {True False})`, it will output `{{True False} {False False}}`, which is the truth-table for the And function. In other words, applying a function to a superposition of its complete domain results in a superposition of its complete image.

So, what happens if we're searching for a specific output? Turns out we can use superpositions to perform this "brute-force" search without loops. In other words, instead of:

```
for x0 in [T, F]:
    for x1 in [T, F]:
        for x2 in [T,F]:
            ...
            print F(x0, x1, x2, ...)
```

We do this instead:

```
print F({T,F}, {T,F}, {T,F}, ...)
```

As you may have expected, the later will share a lot of computation across ”guesses”, resulting in a search that is much faster than a naive brute-force search would be. Below, we exploit this technique to find the solution of a SAT problem in HVM. The ”brute-force” algorithm takes 3 minutes to solve a random instance with 32 variables in Rust. In HVM, the same instance is solved in 1 second by using superpositions. Below is an example SAT solver in HVM, for a random instance with 16 variables:

```
// Bool/List constructors and functions
(Cons h t)  = λcλn(c h (t c n))
Nil         = λcλn(n)
T           = λt λf t
F           = λt λf f
(Id a)      = a
(Not a)     = λt λf (a f t)
(And a b)   = (a λx(x) λx(F) b)
(Or a b)    = (a λx(T) λx(x) b)
(Or3 a b c) = (Or a (Or b c))

// Pretty prints a bool
(Bool x) = (x 1 0)

// Converts a solution to a pretty-printed church-list singleton
(Log x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF x) = (x (Cons λt(t
  (Bool x0) (Bool x1) (Bool x2) (Bool x3)
  (Bool x4) (Bool x5) (Bool x6) (Bool x7)
  (Bool x8) (Bool x9) (Bool xA) (Bool xB)
  (Bool xC) (Bool xD) (Bool xE) (Bool xF)
) Nil) Nil)

// A random 3-SAT instance with 16 variables
(Foo x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF) =
  (Log x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
  (And (Or3 xC x8 (Not xB))
  (And (Or3 (Not xA) (Not x2) (Not x8))
  (And (Or3 xB (Not x9) x4)
  (And (Or3 xA x1 (Not x8))
  (And (Or3 (Not x9) x0 x5)
  (And (Or3 (Not x4) (Not x4) xD)
  (And (Or3 (Not x1) (Not x4) xB)
  (And (Or3 (Not x2) xE (Not xD))
  (And (Or3 xD (Not x8) (Not x7))
  (And (Or3 xD (Not x1) x7)
  (And (Or3 (Not x8) x4 x8)
  (And (Or3 x5 (Not x0) xC)
  (And (Or3 x1 x0 (Not x3))
  (And (Or3 x3 xE (Not xD))
  (And (Or3 x9 (Not xC) (Not xB))
  (And (Or3 x8 (Not xE) (Not x5))
  (And (Or3 (Not x7) (Not x9) (Not xF))
  (And (Or3 xF xB x2)
  (And (Or3 (Not x2) (Not xE) (Not x7))
  (And (Or3 (Not xE) x3 (Not x3))
  (And (Or3 x3 (Not xF) x1)
  (And (Or3 (Not x0) x0 xD)
  (And (Or3 (Not x8) (Not x3) xC)
  (And (Or3 xC (Not x1) x7)
  (And (Or3 x2 (Not xE) (Not x0))
  (And (Or3 x6 (Not x5) x1)
  (And (Or3 (Not xC) x3 (Not x3))
  (And (Or3 (Not x1) (Not xC) (Not x5))
  (And (Or3 xB xA x6)
  (And (Or3 xF x6 x9)
  (And (Or3 (Not xF) x3 (Not x9))
  (And (Or3 (Not xB) x1 x8)
  (And (Or3 x9 (Not xE) xE)
  (And (Or3 (Not xA) (Not x2) x2)
  (And (Or3 x5 xE (Not x2))
  (And (Or3 (Not xB) xC x2)
  (And (Or3 x1 xB (Not x2))
  (And (Or3 x8 (Not x6) (Not x7))
  (And (Or3 xD x9 (Not xA))
  (And (Or3 x0 x8 (Not x8))
  (And (Or3 (Not xA) x9 (Not x0))
  (And (Or3 (Not xE) (Not x7) (Not xC))
  (And (Or3 x9 xC (Not xA))
  (And (Or3 (Not x7) (Not xF) xD)
  (And (Or3 x0 (Not x2) xD)
  (And (Or3 xC (Not x9) (Not x0))
  (And (Or3 (Not xA) x8 (Not x1))
  (And (Or3 x4 x5 (Not xE))
  (And (Or3 (Not x0) x5 (Not x7))
  (And (Or3 (Not x1) (Not xC) xA)
  (And (Or3 x0 xF x9)
  (And (Or3 (Not xA) x3 (Not x2))
  (And (Or3 (Not x7) (Not xF) x6)
  (And (Or3 (Not x1) x4 (Not x5))
  (And (Or3 xC (Not x8) x2)
  (And (Or3 x4 x7 (Not x5))
```

```
  (And (Or3 (Not x9) (Not x0) (Not xA))
  (And (Or3 xC x1 (Not x2))
  (And (Or3 xB xE (Not xB))
  (And (Or3 xF x5 xA)
  (And (Or3 (Not x6) xE x3)
  (And (Or3 (Not xB) xB x1)
  (And (Or3 (Not xF) x0 x7)
       (Or3 xE (Not x3) (Not x2))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))

// Collapsers (to retrieve solutions from superpositions) - credit to @Franchu
Join = λaλbλcλn(a c (b c n))
Col0 = λx let #A0{x0 x1} = x; (Join x0 x1)
Col1 = λx let #A1{x0 x1} = x; (Join x0 x1)
Col2 = λx let #A2{x0 x1} = x; (Join x0 x1)
Col3 = λx let #A3{x0 x1} = x; (Join x0 x1)
Col4 = λx let #A4{x0 x1} = x; (Join x0 x1)
Col5 = λx let #A5{x0 x1} = x; (Join x0 x1)
Col6 = λx let #A6{x0 x1} = x; (Join x0 x1)
Col7 = λx let #A7{x0 x1} = x; (Join x0 x1)
Col8 = λx let #A8{x0 x1} = x; (Join x0 x1)
Col9 = λx let #A9{x0 x1} = x; (Join x0 x1)
ColA = λx let #AA{x0 x1} = x; (Join x0 x1)
ColB = λx let #AB{x0 x1} = x; (Join x0 x1)
ColC = λx let #AC{x0 x1} = x; (Join x0 x1)
ColD = λx let #AD{x0 x1} = x; (Join x0 x1)
ColE = λx let #AE{x0 x1} = x; (Join x0 x1)
ColF = λx let #AF{x0 x1} = x; (Join x0 x1)

// Finds a solution by applying Foo to superposed inputs
Main =
  (Col0 (Col1 (Col2 (Col3
  (Col4 (Col5 (Col6 (Col7
  (Col8 (Col9 (ColA (ColB
  (ColC (ColD (ColE (ColF
  (Foo
    #A0{T F} #A1{T F} #A2{T F} #A3{T F}
    #A4{T F} #A5{T F} #A6{T F} #A7{T F}
    #A8{T F} #A9{T F} #AA{T F} #AB{T F}
    #AC{T F} #AD{T F} #AE{T F} #AF{T F}
  ))))))))))))))))
```

Discussion: https://twitter.com/VictorTaelin/status/1744788091833917696 (https://twitter.com/VictorTaelin/status/1744788091833917696)

H.50 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-search1.md

Fast Discrete Program Search with HVM Superpositions (SUP nodes) - finding ADD-CARRY

URL: https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34 (https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34)

FILE: FAST_DPS_ADD_CARRY.MD

HOC'S FAST DISCRETE PROGRAM SEARCH (DPS)

HOC will soon (EOY?) launch an API for our DPS solution. The interface will be simple:

- You give us a set of examples (input/output pairs)
- We'll give you a (Python?) function that models it

And that's it. It will be an universal function finder.

We'll not release the full source, but, meanwhile, I can talk about the theory. As I've written before, the main idea is to exploit "superpositions" - a feature that HVM inherited from Interaction Combinators - to accelerate DPS (or Symbolic Regression). SUPs are the opposite of DUPs: instead of "cloning one value into two locations", they "merge two values into one location". For example, if you write:

```
Main = (+ {10 20} 1)
```

HVM will add the superposition of {10 20} to 1, resulting in {11 21}. While this looks like something simple you could just add to existing languages - it isn't. To get an asymptotical speedup, SUPs must be first-class, deeply ingrained into the runtime - and that's incompatible with the classical model that all languages use (the full λ-Calculus).

BUT HOW SUPS ACCELERATE DPS?

To explain this, suppose you're trying to find the binary ADD-CARRY algorithm. Assume you don't know the implementation, but you know it can be done with a single recursive pass through both strings. As such, you write the following template:

```
ADC :: Int -> [Int] -> [Int] -> Bool -> [Int]
ADC 0 a     b     c = []
ADC n (0:a) (0:b) 0 = ? : ADC (n-1) a b ?
ADC n (0:a) (1:b) 0 = ? : ADC (n-1) a b ?
ADC n (1:a) (0:b) 0 = ? : ADC (n-1) a b ?
ADC n (1:a) (1:b) 0 = ? : ADC (n-1) a b ?
ADC n (0:a) (0:b) 1 = ? : ADC (n-1) a b ?
ADC n (0:a) (1:b) 1 = ? : ADC (n-1) a b ?
ADC n (1:a) (0:b) 1 = ? : ADC (n-1) a b ?
ADC n (1:a) (1:b) 1 = ? : ADC (n-1) a b ?
```

Now, all you need to do is find the correct replacement for each of the 16 missing bits (?). In a conventional language (like Haskell), you'd need to use a loop to *enumerate* and *try* all the 65,536 options separately. Now, here's the insight: all these "add candidates" share a lot in common. Couldn't we save a lot of time if we could optimally "cache" identical computations across different guesses? Indeed: that's where hash-consing - lol, jk - SUP nodes come to save the day.

On HVM, instead of trying each candidate separately, we can create a superposition of all possible functions. Then, we can apply them all to some test cases. Finally, we can collapse the result into a single value, to recover a function that works. More or less like this:

```
let adc = {superposition of candidates}

let test_0 = add 48436 10318 == 58754
let test_1 = add 38148 25462 == 63610

collapse $ if test_0 && test_1 then None else Some BITS
```

Running this on HVM will output the missing bits:

```
[0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0]
```

And voilà: we just discovered the ADD-CARRY function!

So, how fast this is? In this implementation, a single call to 'add' takes about ~4k interactions. As such, calling it 65k times would take about 262m interactions. Yet, the program above returns in just 36k interactions! That's a ridiculous 7277x speedup. More impressively, we achieved sub-1 interactions per guess! And that speedup only gets larger as the search space grows.

Below I publish the HVM1 source to replicate this experiment:

```
// type Bool = F | T
F = λf λt f
T = λf λt t

// type Bin = (O Bin) | (I Bin)
(O xs) = λo λi (o xs)
(I xs) = λo λi (i xs)

// type Nat = (S n) | Z
(S n)  = λs λz (s n)
Z      = λs λz z

// If-Then-Else
(If 0 t f) = f
(If x t f) = t

// Converts U60 to Bin
(Bin 0 n) = E
(Bin s n) = (If (== (% n 2) 0) (O (Bin (- s 1) (/ n 2))) (I (Bin (- s 1) (/ n 2))))

// Converts Bin to U60
(U60 0 bs) = 0
(U60 n bs) = (bs λp (+ (* 2 (U60 (- n 1) p)) 0) λp (+ (* 2 (U60 (- n 1) p)) 1))

// Converts U60 to Nat
(Nat 0) = Z
(Nat n) = (S (Nat (- n 1)))

// Boolean Fns
And = λa (a λb(F) λb(b))
Or  = λa (a λb(T) λb(b))

// Maybe Or
(Join None     b) = b
(Join (Some x) b) = (Some x)

// Collapse an N-label Superposition
(Collapse 0 x) = x
(Collapse n x) = (HVM.DUP (- n 1) x λx0λx1(Collapse (- n 1) (Join x0 x1)))

// Checks if two Bins are equal
Eq = λn (n
  λn λa (a
    λa λb (b λb ((Eq) n a b) λb F)
    λa λb (b λb F λb ((Eq) n a b)))
  λa λb T)

// Target Add Function (we're searching this)
Add = λn (n
  λn λa (a
    λa λb (b
      λb λk (k
        (O ((Add) n a b F))
        (I ((Add) n a b F)))
      λb λk (k
        (I ((Add) n a b F))
        (O ((Add) n a b T))))
    λa λb (b
      λb λk (k
        (I ((Add) n a b F))
        (O ((Add) n a b T)))
      λb λk (k
        (O ((Add) n a b T))
        (I ((Add) n a b T)))))
  λa λb λk E)

// Superposed Function Template (with 16 missing bits)
Fun = λn (n
  λn λa (a
    λa λb (b
      λb λk (k
        (#0{λx(O x) λx(I x)} ((Fun) n a b #8{T F}))
        (#1{λx(O x) λx(I x)} ((Fun) n a b #9{T F})))
      λb λk (k
        (#2{λx(O x) λx(I x)} ((Fun) n a b #10{T F}))
        (#3{λx(O x) λx(I x)} ((Fun) n a b #11{T F}))))
    λa λb (b
      λb λk (k
        (#4{λx(O x) λx(I x)} ((Fun) n a b #12{T F}))
        (#5{λx(O x) λx(I x)} ((Fun) n a b #13{T F})))
      λb λk (k
```

```
        (#6{λx(O x) λx(I x)} ((Fun) n a b #14{T F}))
        (#7{λx(O x) λx(I x)} ((Fun) n a b #15{T F})))))
  λa λb λe E)

// Superposition Labels used in the Template
Map = [
  #0{0 1} #1{0 1}  #2{0 1}  #3{0 1}  #4{0 1}  #5{0 1}  #6{0 1}  #7{0 1}
  #8{0 1} #9{0 1} #10{0 1} #11{0 1} #12{0 1} #13{0 1} #14{0 1} #15{0 1}
]

// Tests if `Fun a b == c`
(Test s a b c) =
  let x = ((Fun) (Nat s) (Bin s a) (Bin s b) F)
  let y = (Bin s c)
  ((Eq) (Nat s) x y)

// Main
// ----

// Finds the Binary Addition Function
Main =
  let e0 = (Test 16 48436 10318 58754) // test: 48436 + 10318 = 58754
  let e1 = (Test 16 38148 25462 63610) // test: 38148 + 25462 = 63610
  let ok = ((And) e0 e1) // looks for a Fun that passes both tests
  (Collapse 16 (ok None (Some Map))) // output: [0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0]
```

This is the heart of the algorithm we'll be running, and I'm posting it for anyone to explore. In special, note how we fully replaced *loops* and *enumeration* by *superposition* and *collapsing*. Of course, this version is greatly simplified, as I provided a template for the ADD-CARRY shape, simplifying the task. Our full solution generates the templates on the fly, and employs a bunch of new tricks to let us to recover functions from I/O pairs alone. We also use a custom HVM version, and (depending on our Series A outcome 😅) a MASSIVE LOT of compute.

Even with all these tricks, the general problem is still exponential, but, by greatly optimizing the cost of a guess (all the way to <1 interaction), and with massive compute, this "dumb" approach can discover programs and proofs that would otherwise be intractable.

Discussion: https://x.com/VictorTaelin/status/1819774880130158663 (https://x.com/VictorTaelin/status/1819774880130158663)

H.51 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-search3.md

A Superposed λ-Calculus Enumerator for Program Search

URL: https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e (https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e)

FILE: SUPERPOSED_LAMBDA_CALCULUS_EVALUATOR.C

// This file is a mirror of: // https://github.com/HigherOrderCO/HVM3/blob/main/book/lambda_enumerator_optimal.hvml (https://github.com/HigherOrderCO/HVM3/blob/main/book/lambda_enumerator_optimal.hvml)

// An Optimal λ-Calculus Enumerator for Program Search // ---------- // This file shows a template on how to enumerate superposed terms in a // higher-order language (here, the affine λ-Calculus) for proof search. // Instead of generating the source syntax (like superposing all binary strings // and parsing it as a binary λ-calculus), we create a superposition of all // λ-terms *directly*, in a way that head constructors are emitted as soon as // possible. This allows HVM to prune branches and backtrack efficiently as it // computes the application of all λ-terms to some expression. The result is a // reduction in the interactions needed to solve the equation: // > (?X λt(t 1 2)) == λt(t 2 1) // From ~32 million to just 76k, a 420x speedup*, which increases the harder the // problem is. This generator works by synthesizing a λ-term in layers. On each // layer, we either generate a lambda and extend a context, or select one of the // variables in the context to return. When we select a variable, we will apply // it to either 0, 1 or 2 other variables in the context (we don't generate // terms with >2 arity apps here). This is not ideal; in a typed version, we // would be able to tell the arity of the context variable, and generate the // right amount of applications, without making a guess. // * NOTE: we're actually able to bring the naive approach down to 1.7 million // interactions. So, the numbers are: // - Enum binary λC with loops (Haskell): 0.992s // - Enum binary λC with sups (HVM): 0.026s (38x speedup) // - Enum λC directly with sups (HVM): 0.0011s (862x speedup) // The main contribution of this file is on the shape of the superposer. There // are a few things that one must get right to achieve the desired effect. // First, how do we split a linear context? It depends: when generating an // application like `(f ?A ?B)`, we need to pass the context to `?A`, get the // leftover, and pass it to `?B`, in such a way that `?A` and `?B` won't use the // same variable twice. This happens within the same "universe". Yet, when // making a choice, like "do we return a lambda or a variable here", we need to // clone the linear context with the same label forked the universe itself, // allowing a variable to be used more than once, as long as its occurrences are // in

different universes. Handling this correctly is very subtle, which is why // this file can be useful for study. // Second, how do we handle labels? As discussed recently on Discord: // https://discord.com/channels/912426566838013994/915345481675186197/1311434500911403109 (https://discord.com/channels/912426566838013994/915345481675186197/1311434500911403109) // We only need one label to fully enumerate all natural numbers. Yet, that // doesn't work for binary trees. My conclusion is that we need to "fork" the // label whenever we enumerate a constructor that branches; i.e., that has more // than one field. Nats and Bits are safe because their constructors only have // one field, but a binary Tree needs forking. To fork a label, we just mul by 2 // and add 0 or 1, and the seed has to be 1, so that forked branches never use // the same label. We apply this here to the arity-2 app case. // Third, how do we emit constructors as soon as possible, while still passing a // context down? It is easy to accidentally mess this up by making the enum // monadic. This will cause it to sequentialize its execution, meaning no ctor // is emitted until the entire enumerator returns. That's a big problem, since // we need head ctors to be available as soon as possible. That's how HVM is // able to invalidate universes and backtrack. While this is a silly issue, it // can spoil the whole thing, so I've elaborated it here: // https://gist.github.com/VictorTaelin/fb798a5bd182f8c57dd302380f69777a (https://gist.github.com/VictorTaelin/fb798a5bd182f8c57dd302380f69777a) // The enumerator in this file is the simplest "template" enumerator that has // everything a higher order language needs and is structured in a way that can // be studied and extended with more sophisticate approaches, like types. // EDIT: the dependently typed version has been pushed. It reduces the rewrite // count to 3k, and greatly improves the enumerator shape.

```
data List { #Nil #Cons{head tail} }

data Bits { #O{pred} #I{pred} #E }

data Term { #Lam{bod} #App{fun arg} #Var{idx} #Sub{val} }

data pair { #Pair{fst snd} }

data Result { #Result{src val} }

data Maybe { #None #Some{value} }

// Prelude // -------

@if(c t f) = ~ c { 0: f p: t }

@when(c t) = ~ c { 0: * p: t }

@tail(xs) = ~ xs { #Nil: * #Cons{h t}: t }

@and(a b) = ~ a !b { 0: 0 p: b }

@unwrap(mb) = ~ mb { #None: * #Some{x}: x }

@tm0(x) = !&0{a b}=x a @tm1(x) = !&0{a b}=x b

// Stringification // ----------------

@show_nat(nat) = ~nat { 0: λk #Cons{'Z' k} p: λk #Cons{'S' (@show_nat(p) k)} }

@show_dec(n r) = ! &{n n0} = n ! &{n n1} = n ! chr = (+ (% n 10) '0') ~ (< n0 10) !chr !r { 0: @show_dec((/ n1 10) #Cons{chr r}) t: #Cons{chr r} }

@do_show_dec(n) = @show_dec(n #Nil)

@show_bits(bits) = ~bits { #O{pred}: λk #Cons{'#' #Cons{'0' #Cons{'{' (@show_bits(pred) #Cons{'}' k})}}} #I{pred}: λk #Cons{'#' #Cons{'1' #Cons{'{' (@show_bits(pred) #Cons{'}' k})}}} #E: λk #Cons{'#' #Cons{'E' k}} }

@do_show_bits(bits) = (@show_bits(bits) #Nil)

@show_term(term dep) = ~term !dep { #Var{idx}: λk @show_dec((- (- dep idx) 1) k) #Lam{bod}: λk !&{d0 d1}=dep #Cons{'λ' (@show_term((bod #Var{d0}) (+ d1 1)) k)} #App{fun arg}: λk !&{d0 d1}=dep #Cons{'(' (@show_term(fun d0) #Cons{' ' (@show_term(arg d1) #Cons{')' k})})} #Sub{val}: * }

@do_show_term(term) = (@show_term(term 0) #Nil)

// Equality // --------

@eq(a b dep) = ~ a !b !dep { #Lam{a_bod}: ~ b !dep { #Lam{b_bod}: !&{dep d0}=dep !&{dep d1}=dep !&{dep d2}=dep @eq((a_bod #Var{d0}) (b_bod #Var{d1}) (+ 1 d2)) #App{b_fun b_arg}: 0 #Var{b_idx}: 0 #Sub{b_val}: * } #App{a_fun a_arg}: ~ b !dep { #Lam{b_bod}: 0 #App{b_fun b_arg}: !&{dep d0}=dep !&{dep d1}=dep @and(@eq(a_fun b_fun d0) @eq(a_arg b_arg d1)) #Var{b_idx}: 0 #Sub{b_val}: * } #Var{a_idx}: ~ b !dep { #Lam{b_bod}: 0 #App{b_fun b_arg}: 0 #Var{b_idx}: (== a_idx b_idx) #Sub{b_val}: * } #Sub{a_val}: * }
```

// Evaluation // ----------

@wnf(term) = ~ term { #Lam{bod}: #Lam{bod} #App{fun arg}: @wnf_app(@wnf(fun) arg) #Var{idx}: #Var{idx} #Sub{val}: #Sub{val} }

@wnf_app(f x) = ~ f !x { #Lam{bod}: @wnf((bod @wnf(x))) #App{fun arg}: #App{#App{fun arg} x} #Var{idx}: #App{#Var{idx} x} #Sub{val}: #App{#Sub{val} x} }

// Normalization // -------------

@nf(term) = ~ @wnf(term) { #Lam{bod}: #Lam{λx @nf((bod #Sub{x}))} #App{fun arg}: #App{@nf(fun) @nf(arg)} #Var{idx}: #Var{idx} #Sub{val}: val }

// Enumeration // -----------

// Enumerates affine λ-terms. // - lim: max context length (i.e., nested lambdas) // - lab: superposition label. should be 1 initially. // - ctx: the current scope. should be [] initially. // If the binder limit has been reached, destroy this universe. // Otherwise, make a choice. // - A. We generate a fresh lambda. // - B. We select a variable from context. // Note that, every time we make a choice, we ”fork” the current context by // using DUP nodes with the same label that we used in the choice SUP node. @all(&L &lim ctx) = ~&lim { 0: * &lim: !&L{ctxL ctxR} = ctx &L{ @lam(&L &lim ctxL) @ret(&L (+ &lim 1) ctxR λk(k)) } }

// Generate a fresh lambda and extend the context with its variable. @lam(&L &lim ctx) = !&0{ctx bod} = @all(&L &lim #Cons{#Some{$x} ctx}) &0{@tail(ctx) #Lam{λ$x(bod)}}

// Return a variable from the context. // If the context is empty, destroy this universe. // Otherwise, make a choice. // - A. We emit the head of the context, and apply it to things. // - B. We keep the head of the context, and go to the next element. @ret(&L &lim ctx rem) = ~ctx { #Nil: * #Cons{val ctx}: !&L{remL remR} = rem !&L{valL valR} = val !&L{ctxL ctxR} = ctx &L{ @app(&L &lim (remL #Cons{#None ctxL}) valL) @ret(&L &lim ctxR λk(remR #Cons{valR k})) } }

// To apply a value to things, we will make a triple choice. // - A. Just return it directly. // - B. Apply it to 1 argument. // - C. Apply it to 2 arguments. // When we apply it to 2 arguments, as in `(App ?A ?B)`, we need to fork the // label, so that DUPs/SUPs in `?A` and `?B` never use the same label. @app(&L &lim ctx val) = ~ val { #None: * #Some{val}: !&L{val val0} = val !&L{val val1} = val !&L{val val2} = val !&L{ctx ctx0} = ctx !&L{ctx ctx1} = ctx !&L{ctx ctx2} = ctx ! arity_0 = &0{ctx0 val0} ! arity_1 = !&0{ctx1 argX} = @all(&L &lim ctx1) &0{ctx1 #App{val1 argX}} ! arity_2 = !&0{ctx2 argX} = @all((+(*&L 2) 0) &lim ctx2) !&0{ctx2 argY} = @all((+(*&L 2) 1) &lim ctx2) &0{ctx2 #App{#App{val2 argX} argY}} &L{arity_0 &L{arity_1 arity_2}} }

// Tests // -----

//A= λt (t ^1 ^2) //A= λ((Z SZ) SSZ) @A = #Lam{λt #App{#App{t #Var{1}} #Var{2}}}

//B= λt (t ^2 ^1) //B= λ((Z SSZ) SZ) @B = #Lam{λt #App{#App{t #Var{2}} #Var{1}}}

//R= λx (x λa λb λt (t b a)) //R= λ(Z λλλ((SSSZ SSZ) SZ)) @R = #Lam{λx #App{x #Lam{λa #Lam{λb #Lam{λt #App{#App{t b} a}}}}}}

//X= (all terms) @X = @tm1(@all(1 5 #Nil))

// Solves for `?X` in `(?X λt(t A B)) == λt(t B A)`. // It finds `?X = λλ(1 λλ((2 0) 1))` in 76k interactions. @main = ! solved = @eq(@nf(#App{@X @A}) @B 0) @when(solved @do_show_term(@X))

H.52 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-sieve.md

Can a simple functional sieve be fast? Optimizing Tromp's algorithm on HVM.

URL: https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8 (https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8)

FILE: SIMPLE_FAST_FUNCTIONAL_SIEVE.MD

CAN A SIMPLE FUNCTIONAL SIEVE BE FAST? OPTIMIZING TROMP'S ALGORITHM ON HVM.

Today, John Tromp - creator of the Binary λ-Calculus (https://tromp.github.io/cl/Binary_lambda_calculus.html), and one of the smartest functional wizards alive - ported his famous prime number generator (first published 12 years ago!) to HVM (https://github.com/higherorderco/hvm):

```
B0     = λx λy x
B1     = λx λy y
cons   = λx λy λz (z x y)
cons0  = λy λz (z B0 y)
Y      = λf ((λx (x x)) (λx (f (x x))))
S0     = λcont  λx λxs (cons0  (xs cont))
Ssucc  = λSn λc λx λxs (cons x (xs (Sn c)))
ap     = λx λy λz (x z (y z))
sieve  = λSn (cons B1 (ap sieve Y (Ssucc Sn)))
primes = (cons0 (cons0 (sieve S0)))
C2     = λf λx (f (f x))
C5     = λf λx (f (f (f (f x)))))
C10    = λf (C2 (C5 f))
C1K    = (C10 C2)
C3     = λf λx (f (f (f x)))
C4     = λf λx (f (f (f (f x))))
C8     = (C3 C2)
C256   = (C8 C2)
C12    = λf (C3 (C4 f))
C4K    = (C12 C2)
Take   = λcont λx λxs (SCons (x '0' '1') (xs cont))
Main   = (primes (C1K Take (B0 (B0 SNil))))
```

His algorithm is remarkable for being extremely elegant and small - possibly, the smallest of all! Sadly, it isn't so fast. To generate the `N` th bit, a "siever" has to be applied `N` times to it, which makes its complexity quadratic. So, the question is: **can we take advantage of HVM's optimality, to gain an asymptotical speedup on this elegant algorithm?** My first intuition was to isolate the simplest siever (i.e., `siever(1)`), and measure its iterated complexity. As expected, it was linear; applying 1 million sievers had a cost of *30,000,040 interactions*.

```
App   = λn λf λx match n { 0: x; +: (App (- n 1) f (f x)) } // applies a function N times
Y     = λf (f (Y f)) // y-combinator
B0    = λb0 λb1 b0 // bit 0
B1    = λb0 λb1 b1 // bit 1
cons  = λx λxs λcons (cons x xs) // cons on infinite lists
head  = λxs (xs λx λxs x) // gets the first bit
ones  = λc(c B1 ones) // infinite bitstring with 1's
S0    = λcont λx λxs (cons B0 (xs cont)) // the first siever (i.e., siever(1))
apply = λs λx (x (Y s)) // applies a siever
(F n) = (head (App n (apply S0) ones)) // applies siever(1) N times to [1..], and gets the first bit
Main  = (F 1000000) // COST: 30,000,040 interactions
```

Note that, on HVM, each *interaction* (also called "rewrite") is a constant-time operation that takes just a few ASM instructions. As such, it is a precise measure of complexity. That is, instead of measuring time and using the imprecise `big-O` notation (`siever1^N = O(N)`), we can measure and write it in terms of interactions: `siever1^N = 30*N + 40 interactions`. Below are some specific measures:

```
// F(100) =     3040 interactions
// F(200) =     6040 interactions
// F(300) =     9040 interactions
// F(400) =    12040 interactions
// F(500) =    15040 interactions
// F(600) =    18040 interactions
// F(700) =    21040 interactions
// F(800) =    24040 interactions
// F(900) =    27040 interactions
// F(1k)  =    30040 interactions
// F(1kk) = 30000040 interactions
```

Thankfully, HVM as a clever way to perform repeated application in logarithmic time: fusion! To obtain a fusible siever, I applied 3 transformations:

1. Perform repeated application by squared self-composition.
2. Flip the siever, so that it is a applied to the list (not the other way around).
3. Lift the cons lambda inside the siever, allowing it to be shared.

I then ran the same computation and - wow - it completed in just *1012 interactions*! Now, that's an optimization. By merely fusing the siever, we got a massive 29644x speedup. Asymptotics!

```
App   = λn λf λx match n { 0: x; +: (App (/ n 2) λx(f (f x)) (match m = (% n 2) { 0: x; +: (f x) })) } // applies a function N times
Y     = λf (f (Y f)) // y-combinator
B0    = λb0 λb1 b0 // bit 0
B1    = λb0 λb1 b1 // bit 1
cons  = λx λxs λcons (cons x xs) // cons on infinite lists
head  = λxs (xs λx λxs x) // gets the first bit
ones  = λc(c B1 ones) // infinite bitstring with 1's
S0    = λcont λbs λcons (bs λxλxs(cons B0 (cont xs))) // the first siever (i.e., siever(1))
apply = λs λx ((Y s) x) // applies a siever
(F n) = (head (App n (Y S0) ones)) // applies siever(1) N times to [1..], and gets the first bit
Main  = (F 1000000) // COST: 1012
```

Below are some updated measures:

```
// F(1)   =  381 interactions
// F(2)   =  428 interactions
// F(3)   =  480 interactions
// F(4)   =  475 interactions
// F(5)   =  490 interactions
// F(6)   =  527 interactions
// F(7)   =  537 interactions
// F(8)   =  522 interactions
// F(9)   =  527 interactions
// F(1k)  =  537 interactions
// F(1kk) = 1012 interactions
```

So, can this trick be used to create a simple *and* fast pure functional prime sieve? Probably! But one would need to carefully apply this technique to fuse arbitrary sievers during the sieve generation. Tricky, yes; but doable. So, a better question is: who will be the first one to develop it? Could be you!

Note that, as of today (29 Jan 2024), HVM's production-ready version isn't officially available yet, but you can already test the code above by installing the `lazy_mode` branch of HVM-Lang (https://github.com/higherOrderCO/HVM-lang), with the `-L` flag.

This discussion is taking place right now on HOC's Discord:

https://discord.HigherOrderCO.com/ (https://discord.HigherOrderCO.com/) (#hvm channel)

H.53 · bend-docs/openspec/changes/bend-docs-site/research/victor/hvm-unordered-sup.md

Truly Optimal Evaluation with Unordered Superpositions

URL: https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a (https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a)

FILE: TRULY_OPTIMAL_EVALUATION_WITH_UNORDERED_SUPERPOSITIONS.MD

Truly Optimal Evaluation with Unordered Superpositions

In this post, I'll address two subjects:

1. How to solve HVM's quadratic slowdown compared to GHC in some cases
2. Why that is relevant to logic programming, unification, and program search

OPTIMAL EVALUATORS AREN'T OPTIMAL

HVM is an optimal λ-calculus evaluator in Lévy's sense: given an admissible λ-term, it computes its normal form with the minimal number of β-reductions. This, in some cases, leads to exponential speedups compared to non β-optimal evaluators like Haskell's GHC, making HVM a promising tool for designing purely functional algorithms. No wonder I was surprised when @oxalica opened a GitHub issue (https://github.com/HigherOrderCO/HVM/issues/60) reporting a function which HVM computes quadratically slower than GHC.

After some research, I learned this problem isn't a bug - it is known and documented in the literature. The culprit is the accumulation of "redundant" control nodes or duplication nodes in some cases. To my understanding, only by eliminating these can an "optimal" evaluator be truly optimal, not just in Lévy's sense (minimizing β-reductions), but also

in general (minimizing all steps in a Turing machine). Asperti, BOHM's author, "patched" this by using "safe nodes", which they describe in chapter 9 of "The Optimal Implementation of Functional Programming Languages". Unfortunately, while this solution "works well in practice", it is more of a workaround that doesn't work in all cases.

WHAT IT TAKES TO BE OPTIMAL

After reasoning about the problem, I developed a good intuition about the issue, which I'll explain in simple terms. To perform optimal evaluation, in the sense of minimizing β-reductions, we need two things: 1. laziness, and 2. incremental duplications. These must be features of an optimal evaluator since, without them, work can be wasted either by evaluating unnecessary expressions or by evaluating the same expression more than once. That's the key insight when it comes to optimal functional evaluation.

Interestingly, Haskell is accidentally "almost optimal" because 1. it is lazy, and 2. it also implements a form of lazy cloning via STG's thunks. Haskell's thunks allow immutable datatypes to be shared and expressions to be "memoized". Unfortunately, this doesn't work with lambdas, since we can't share free variables, as they can become different in different contexts. This forces Haskell to undo sharing to evaluate lambdas, duplicating work and leading to the famous cases where Haskell is exponential while HVM is linear. Church Number exponentiation is notorious precisely because it is a λ-term with tons of inherent sharing, which fails to be captured by Haskell's evaluator.

Fan nodes, discovered in Optimal Evaluation literature, solve that issue precisely by providing a solution to the problem of incrementally cloning a lambda. Sadly, they introduce a different issue: that of accumulating "redundant" fan nodes, leading to a quadratic slowdown in some cases. A question thus remains: how do we avoid this quadratic slowdown in an otherwise optimal evaluator, without an "ugly hack" like safe nodes? I now believe I have a satisfactory approach that is quite elegant, which I'm going to share soon. But before that, let's talk about superpositions and program search.

EMULATING LOGIC PROGRAMMING WITH SUPERPOSITIONS

An interesting potential application of optimal evaluation is search. In particular, if we turn fan nodes into first-class constructs, they can be used directly in a program to express "superposed expressions", allowing us to greatly optimize some computations. For example, consider:

```
@main = (@sum [1, 2, {3,4}, 5])
```

Here, `{3,4}` represents a superposition of `3` and `4`, placed in the same index of the list. HVM will compute this expression and return `{11,12}`, which is a superposition of the sum of both lists:

```
@main = {(@sum [1,2,3,5]) (@sum [1,2,4,5])}
```

Except that it won't actually clone the list like that. Instead, it will traverse it only once, sharing computations along the way. In this Gist (https://gist.github.com/VictorTaelin/7fe49a99ebca42e5721aa1a3bb32e278), I elaborate on how that can be used to accelerate Discrete Program Search. It can also be used to emulate logic-based programming by representing pseudo metavars as superpositions of all terms of a type, and then collapsing them to a single value. See here (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615) for a SAT solving example.

For the purpose of this post, let's focus on a simpler example. Suppose we want to solve the following equation:

```
X + 1 == 0b10010111
```

That is, we want to invert the `+ 1` operation and find the predecessor of 0b10010111. With superpositions, we could do it like this:

```
// Metavar X: a superposition of all bitstrings
X = {E {(O X) (I X)}}

// BitString representing 0b10010111
Y = (I (I (I (O (I (O (O (I E)))))))

// Solves X + 1 == Y
Main =
  let goal = (Equal (Add X (I E)) Y)
  (Collapse (if goal { X } else { * }))
```

This will essentially "fork" HVM's runtime into infinite universes, one for each value of X, and return the first one that satisfies the equation `X + 1 == Y`. Then, in that universe, we print the value of `X`. In any other runtime, doing so would be exponential on the bitstring length `S` - after all, there are 2^S values to consider. Yet, this actually works on HVM and is merely quadratic. The reason is that there is an immense amount of shared work across these "universes", and optimal lazy evaluation quickly "aborts" any universe where the equation isn't satisfied.

VISUALIZING QUADRATIC SLOWDOWN ON PSEUDO UNIFICATION

Does the above mean HVM (and Bend) can be used as great logic-based programming tools out of the box? Not quite yet. Unfortunately, this same quadratic slowdown also manifests here, somewhat preventing the idea. The issue is that, while we indeed avoid the exponential blowup in cases like searching for a binary string that satisfies a proposition - which is cool - we embarrassingly need quadratic time to in cases that should absolutely be linear. For example, consider the goal:

```
X = 100
```

Where `X` is a Peano Nat. In any half-decent logic-based programming language, this equation would immediately unify `X` as `100` in `O(1)` time - obviously so. Now, in the "pseudo metavar" solution, we should expect that to be, at best, linear time; after all, we need to spawn 100 "parallel universes" only to destroy all universes where `X != 100`. Even if we completely ignored the cost of spawning and destroying universes, just comparing `100 == 100` is already linear. So, that's the best we can do. Yet, that's not too bad either. If it was indeed linear, and efficient, it would actually be quite practical.

Yet, unfortunately, it isn't linear - it is quadratic. And the interaction count blows up pretty badly for large enough numbers. Not exponentially bad, but quadratically bad. Which is still bad. So, why does that happen?

To visualize this issue, I've drawn, by hand, the full evaluation of the process of solving `X == 4` in HVM, where `4` is a Peano Nat, and `X` is a superposition of all Peano Nats up to 4. Here it is:

```
 {Z (S {Z (S {Z (S {Z (S {Z (S Z)})})})})} == (s (s (s (s z))))
------------------------------------------------------
{x0 x1} = (s (s (s (s z))))
{(Z == x0) ((S {Z (S {Z (S {Z (S Z)})})}) == x1)}
------------------------------------------------
{x0 x1} = (s (s (s z)))
{(Z == (s x0)) ((S {Z (S {Z (S {Z (S Z)})})}) == (s x1))}
------------------------------------------------------
{*  x1} = (s (s (s z)))
{⊥ ({Z (S {Z (S {Z (S Z)})})} == x1)}
-----------------------------------
{*  x1} = (s (s (s z)))
{x2 x3} = x1
{⊥ {(Z == x2) ((S {Z (S {Z (S Z)})}) == x3)}}
---------------------------------------------- WASTED 1
{*  x1} = (s (s z))
{x2 x3} = (s x1)
{⊥ {(Z == x2) ((S {Z (S {Z (S Z)})}) == x3)}}
---------------------------------------------- WASTED 2
{*  x1} = (s (s z))
{x2 x3} = x1
{⊥ {(Z == (s x2)) ((S {Z (S {Z (S Z)})}) == (s x3))}}
-----------------------------------------------------
{*  x1} = (s (s z))
{*  x3} = x1
{⊥ {⊥ ({Z (S {Z (S Z)})} == x3)}}
---------------------------------
{*  x1} = (s (s z))
{*  x3} = x1
{x4 x5} = x3
{⊥ {⊥ {(Z == x4) ((S {Z (S Z)}) == x5)}}}
----------------------------------------- WASTED 3
{*  x1} = (s z)
{*  x3} = (s x1)
{x4 x5} = x3
{⊥ {⊥ {(Z == x4) ((S {Z (S Z)}) == x5)}}}
----------------------------------------- WASTED 4
{*  x1} = (s z)
{*  x3} = x1
{x4 x5} = (s x3)
{⊥ {⊥ {(Z == x4) ((S {Z (S Z)}) == x5)}}}
----------------------------------------- WASTED 5
{*  x1} = (s z)
{*  x3} = x1
{x4 x5} = x3
{⊥ {⊥ {(Z == (s x4)) ((S {Z (S Z)}) == (s x5))}}}
-------------------------------------------------
{*  x1} = (s z)
{*  x3} = x1
{*  x5} = x3
{⊥ {⊥ {⊥ ({Z (S Z)} == x5)}}}
-----------------------------
{*  x1} = (s z)
{*  x3} = x1
{*  x5} = x3
{x6 x7} = x5
{⊥ {⊥ {⊥ {(Z == x6) ((S Z) == x7)}}}}
------------------------------------ WASTED 6
{*  x1} = z
{*  x3} = (s x1)
{*  x5} = x3
{x6 x7} = x5
{⊥ {⊥ {⊥ {(Z == x6) ((S Z) == x7)}}}}
------------------------------------ WASTED 7
{*  x1} = z
{*  x3} = x1
{*  x5} = (s x3)
{x6 x7} = x5
{⊥ {⊥ {⊥ {(Z == x6) ((S Z) == x7)}}}}
------------------------------------ WASTED 8
{*  x1} = z
{*  x3} = x1
{*  x5} = x3
{x6 x7} = (s x5)
{⊥ {⊥ {⊥ {(Z == x6) ((S Z) == x7)}}}}
------------------------------------ WASTED 9
{*  x1} = z
{*  x3} = x1
{*  x5} = x3
{x6 x7} = x5
{⊥ {⊥ {⊥ {(Z == (s x6)) ((S Z) == (s x7))}}}}
```

```
-----------------------------------------------
{*  x1} = z
{*  x3} = x1
{*  x5} = x3
{*  x7} = x5
{⊥ {⊥ {⊥ {⊥ (Z == x7)}}}}
------------------------
{*  x1} = z
{*  x3} = x1
{*  x5} = x3
{*  x7} = x5
{⊥ {⊥ {⊥ {⊥ (Z == x7)}}}}
------------------------ WASTED 10
{*  x3} = z
{*  x5} = x3
{*  x7} = x5
{⊥ {⊥ {⊥ {⊥ (Z == x7)}}}}
------------------------ WASTED 11
{*  x5} = z
{*  x7} = x5
{⊥ {⊥ {⊥ {⊥ (Z == x7)}}}}
------------------------ WASTED 12
{*  x7} = z
{⊥ {⊥ {⊥ {⊥ (Z == x7)}}}}
------------------------ WASTED 13
{⊥ {⊥ {⊥ {⊥ (Z == z)}}}}
------------------------ WASTED 14
{⊥ {⊥ {⊥ {⊥ T}}}}
```

Now, take a careful look at the computation above. Can you spot the issue? Of course you can, since I labeled the bad interactions with "WASTED". The issue is, precisely, that of accumulating redundant duplication nodes. For example, in the part from "WASTED 6" to "WASTED 9", it takes 4 interactions to copy that single "S" from the right-hand side value to the equations, because the constructor has to jump between a chain of "redundant" duplications that make a copy only to immeidatelly discard it. As the program evolves, each subsequent copy of a head "S" constructor takes +1 interaction, resulting in a quadratic blowup. To fully copy the Peano Nat `1000`, we would need about 1 million interactions!

BOHM'S OLD SOLUTION: SAFE NODES AND INTERACTIONS

So, how could we solve this? As I mentioned, Asperti solved it "in practice" on BOHM by implementing "safe nodes". The idea is to remove redundant duplications; i.e., whenever we have:

```
{* x} = T
```

We just substitute `x <- T` and remove that fan node. The problem is that this rule isn't always safe: if there is a SUP node with the same label as that DUP node in some other part of the program, merely removing it could result in an incorrect interaction later on. Asperti's idea is to flag nodes that are safe in the sense that applying this rule is harmless. That basically happens when a DUP node never annihilated with a LAM node. In HVM terms, that's because the DUP-LAM rule creates a DUP and a SUP, rather than two DUPs:

```
! &L{r s} = λx(f)
----------------- DUP-LAM
! &L{f0 f1} = f
r <- λx0(f0)
s <- λx1(f1)
x <- &L{x0 x1}
```

As long as there are only DUPs (or only SUPs) of a given fan node label, this rule can be safely applied for these nodes. Asperti keeps a boolean flag on the nodes, which works well enough. For HVM, this approach isn't so promising, though, because this technique relies on erasure interactions, which HVM doesn't perform. So, what's the plan?

VISUALIZING A HASKELLISH SOLUTION

Let's for a moment consider the following thought experiment: imagine we had a new kind of superposition node that behaved like a hybrid of Haskell's thunks and HVM's duplicators. That is, whenever a SUP would generate a DUP; for example, via the APP-SUP rule:

```
(&L{a b} c)
---------------- APP-SUP
! &L{x0 x1} = c
&L{(a x0) (b x1)}
```

We, instead, create a "Haskell-like thunk". How would that work? Let's try.

```
{Z (S {Z (S {Z (S {Z (S Z)})})})} == (s (s (s (s))))
------------------------------------------------------
{x} = (s (s (s (s))))
{(Z == x) ((S {Z (S {Z (S {Z (S Z)})})}) == x)}
------------------------------------------------
{x} = (s (s (s)))
{(Z == (s x)) ((S {Z (S {Z (S {Z (S Z)})})}) == (s x))}
------------------------------------------------------
{x} = (s (s (s)))
{⊥ ({Z (S {Z (S {Z (S Z)})})} == x)}
---------------------------------- *
{x} = (s (s (s)))
{⊥ {(Z == x) ((S {Z (S {Z (S Z)})}) == x)}}
--------------------------------------------
{x} = (s (s z))
{⊥ {(Z == (s x)) ((S {Z (S {Z (S Z)})}) == (s x))}}
---------------------------------------------------
{x} = (s (s z))
{⊥ {⊥ ({Z (S {Z (S Z)})} == x)}}
------------------------------- JOIN x
{x} = (s (s z))
{⊥ {⊥ {(Z == x) ((S {Z (S Z)}) == x)}}}
--------------------------------------
{x} = (s z)
{⊥ {⊥ {(Z == (s x)) ((S {Z (S Z)}) == (s x))}}}
-------------------------------------------------
{x} = (s z)
{⊥ {⊥ {⊥ ({Z (S Z)} == x)}}}
---------------------------
{x} = (s z)
{⊥ {⊥ {⊥ {(Z == x) ((S Z) == x)}}}}
-----------------------------------
{x} = z
{⊥ {⊥ {⊥ {(Z == (s x)) ((S Z) == (s x))}}}}
-------------------------------------------
{x} = z
{⊥ {⊥ {⊥ {⊥ (Z == x)}}}}
------------------------
{⊥ {⊥ {⊥ {⊥ (Z == z)}}}}
------------------------
{⊥ {⊥ {⊥ {⊥ ⊤}}}}
```

Well, that's much shorter! But what does that even mean in the context of Interaction Combinators? Can these concepts mix? I think they can.

THUNKS ARE UNORDERED DUP NODES

My interpretation of this is that, from the point of view of Interaction Combinators, what the program above did was exploit the concept of an *unordered superposition* - that is, a SUP node `{x y}` such that `∀x y. {x y} == {y x}` - as well as its DUP node counterpart. In other words, where the runtime is free to swap the SUP's ports, and where the user isn't able to split a SUP node in two; they can only "pop an element of the bag". But how does that relate to the evaluation above?

Let's think about this: suppose we had a tree of DUP nodes cloning the nat `3`, and using it inside some list later:

```
{x0 xA} = (S (S (S Z)))
{x1 xB} = xA
{x2 xC} = xB
{x3 xD} = xC
...
[... x0 x1 x2 x3 ... ]
```

How many steps would HVM need to take the `wnf` of `x0`, `x1`, `x2` and `x3`? Intuitively, it should be `1` for all - but that's not what happens. Since the DUP nodes form a tree, `x0` will immediately evaluate to `(S ...)`, in a single DUP-CTR interaction, while `x3` will require 4 interactions, as `S` moves down the tree, all the way to the list:

```
{x0 xA} = (S (S (S Z)))
{x1 xB} = xA
{x2 xC} = xB
{x3 xD} = xC
...
[... x0 x1 x2 x3 ... ]
----------------------
{x0 xA} = (S (S Z))
{x1 xB} = (S xA)
{x2 xC} = xB
{x3 xD} = xC
...
[... (S x0) x1 x2 x3 ... ]
--------------------------
{x0 xA} = (S (S Z))
{x1 xB} = xA
{x2 xC} = (S xB)
{x3 xD} = xC
...
[... (S x0) (S x1) x2 x3 ... ]
------------------------------
{x0 xA} = (S (S Z))
{x1 xB} = xA
{x2 xC} = xB
{x3 xD} = (S xC)
...
[... (S x0) (S x1) (S x2) x3 ... ]
----------------------------------
{x0 xA} = (S (S Z))
{x1 xB} = xA
{x2 xC} = xB
{x3 xD} = xC
...
[... (S x0) (S x1) (S x2) (S x3) ... ]
--------------------------------------
```

Which is suboptimal. Now, suppose that SUP/DUP nodes were unordered. This would allow the runtime to freely rearrange the tree of DUPs. So, if we knew, for a fact, that `x3` was more important, for whatever reason - we could, in theory, rearrange the nodes so that `x3` is the closest to the number instead. That alone would be enough for us to solve the quadratic slowdown in that case. But why stop there? If we're able to rearrange DUPs at will, why do we need to store two variables on DUP nodes at all? After all, if DUPs are unordered, then the `x` and `y` in `{x y} = V` are observationally equivalent. So, we can, instead, merge both ports and store a single variable instead:

```
{x} = (S (S (S Z)))
{y} = x
{z} = y
{w} = z
...
[... x y z w ... ]
```

But wait - these *chains of DUPs* are now clearly redundant. So, we can just shorten it all as:

```
{x} = (S (S (S Z)))
[... x x x x ... ]
```

And, now, any of the occurrences of `x` can wnf in `O(1)`! Cool, right?

UNORDERED DUP/SUP INTERACTIONS: WHERE THEY HELP, WHAT THEY BREAK

But wait, hold on - have we actually just reinvented Haskell's thunk? Are we discovering the groundbreaking technique of... *ref counting*? Well, yes and no. Let's take a moment to reason about how this would actually work in the context of interactions. For example, if the conventional DUP-CTR interaction is:

```
{x0 x1} = (S pred)
------------------ DUP-CTR
x0 <- (S y0)
x1 <- (S y1)
{y0 y1} = pred
```

Then, the UDUP-CTR interaction is:

```
{x} = (S pred)
-------------- UDUP-CTR
x <- (S y)
{y} = pred
```

That basically means that the first thread to come in performs the interaction, peeling an `S` layer off, and storing `(S y)` as `x` in the substitution map, where `y = pred` is a fresh new unordered duplication node. Moreover, since many parts of the program can now access the same variable `x`, we better make the subst map ref-counted, allowing us to clear an entry when a substitution has been consumed by all of its occurrences. Fair enough, right? And indeed, so far, this looks almost exactly like how Haskell behaves. So, where do things fall apart?

Let's have a look at the DUP-LAM interaction:

```
{f0 f1} = λx(body)
------------------- DUP-LAM
{body0 body1} = body
f0 <- λx0(body0)
f1 <- λx1(body1)
x  <- {x0 x1}
```

If we collapsed SUPs and DUPs to be unordered, then the corresponding UDUP-LAM would become something like...

```
{f0 f1} = λx(body)
------------------- UDUP-LAM
{body'} = body
f0 <- λx0(body')
f1 <- λx1(body')
x  <- {x0 x1}
```

Which looks rather problematic. This is why Haskell undoes sharing when a lambda is involved. Otherwise, we'd have the bizarre behavior that cloned lambdas become "entangled" in a rather defective manner: they now share the same body, and whenever we apply one of them to an argument, the substitution may end up happening on its entangled lambda's body - randomly so. After all, the `{x0 x1}` superposition is part of both bodies, but that SUP is unordered! That looks like a very useless feature to have and definitely something too strange to call a lambda - even by my standards. For example, consider:

```
{f0 f1} = λx(x)
[f0 f1]
---------------
{f0 f1} = {x0 x1}
[λx1(f0) λx0(f1)]
-----------------
[λx1(x0) λx0(x1)]
```

The program above attempted to copy the identity function by using an unordered DUP. It resulted in two lambdas with their bodies hilariously mixed up - which is not invalid in HVM, but is devoid of any meaning or sense. This also means that, sadly, unordered dups can't really be used to emulate logic-based programming with λ-encodings. But they provide a way to copy low-order datatypes that is closer to how Haskell does it (solving the quadratic issue posted by @oxalica), while also allowing us to emulate logic-based programming on HVM with, I believe, possibly reasonable asymptotics: to emulate a metavariable, we just create a superposition of all elements of a datatype.

CONCLUSION

In essence, this is kinda like ref counting - or, rather, how ref counting would work in the context of an interaction net system, and where it would break. In special, it justifies and explains how we should proceed to incorporate something akin to Haskell's thunk on HVM's model. This gives us Haskell-like asymptotics in cases where HVM would be quadratic (as pointed in that old GitHub issue), while still allowing us to remain linear in cases where GHC is exponential (sharing closures). And a cool side-effect is that we gain access to the dual of thunks - i.e., unordered superpositions. And that could have powerful applications to search and logic based programming, since USUP nodes allow us to create superposed values in a way that optimally reuses computation across universes.

I've not implemented any of this yet, and this is all theoretical for now and I could be wrong. But it makes sense to me, and I plan to add it soon to the HVM3 WIP repository `(https://github.com/HigherOrderCO/HVM3)`.

EDIT:

This has now been implemented experimentally on HVM3 and it, indeed, **reduces the interaction count (and run time) of the program above, from quadratic to linear!** That said, I have more to say about the UDUP-LAM interaction.

Initially, my plan was to just not implement it. That is, consider it undefined behavior and throw a runtime error when it occurs. Sadly, for the examples above to work, we actually need to provide some UDUP-LAM rule. That's because UDUPs will eventually interact with LAM nodes during its reduction. This was not visible on my drawings because I conveniently reduced `A == B` in a single step, but `==` is meant to represent a user-defined equality function for Peano Nats. Something like:

```
eq a b = match a {
  #Z: match b {
    #Z: 1
    #S: λb_pred 0
  }
  #S: λa_pred match b {
    #Z: 0
    #S: λb_pred (eq ap bp)
  }
}
```

So, when we have an application like `(eq {A0 A1} B)`, i.e., with an unordered superposition, it will reduce to:

```
match {A0 A1} {
  #Z: zero-case
  #S: λa_pred succ-case
}
```

Which will then reduce to:

```
{Z} = zero-case
{S} = λa_pred succ-case
{
  match A0 { #Z:zero-case #S:succ-case }
  match A1 { #Z:zero-case #S:succ-case }
}
```

And here we have `{S} = λa_pred succ-case`, which means we absolutely need to provide *some* justification for the UDUP-LAM rule. What now?

I could go ahead and just implement the rule I suggested. It would actually work, if we ignore the fact it isn't too useful to clone lambdas (due to the swapped bodies issue). The problem is, due to the optimization where we store both UDUP binders in a single port - which is what allow us to merge UDUP nodes and get the quadratic speedup - we can't actually implement the UDUP-LAM rule as written. That rule relies on both ports existing.

Now, observe that, for this specific case, the `{Z} = zero-case` and the `{S} = succ-case` do not actually need to clone anything, because our initial superposition covered different variants of the Nat type: Z and S. As such, either we'll select zero/succ, or succ/zero, but never zero/zero or succ/succ. The body is never cloned. This "happy coincidence" (inets have many of these, don't they?) suggests a clever solution: rather than making the UDUP-LAM rule undefined, we can define it, under the assumption that it will only be used once. The definition, thus, becomes:

```
! &L{F} = λx(f)
----------------- UDUP-LAM
! &L{G} = f
F <- λx0(G)
F <- *
x <- &L{x0 *}
```

That is, when we perform an "unordered duplication" of a lambda, we forcefully erase one of the copies, based on the assumption that this UDUP node is only triggered once. That's always the case when it was generated by a superposition of different variants of an ADT, which is the main use-case I'm looking for. That said, this means HVM3 is not a faithful implementation of unordered superpositions. I'll think whether we can have the quadratic speedup and a proper UDUP-LAM interaction later on, as the "body swapping" behavior might perhaps be desirable in some contexts (which?). Perhaps we should just keep the 2 ports, and just swap UDUPs when convenient?

EDIT: SIMPLER IMPLEMENTATION BASED ON OPORTUNISTIC SWAPS

(Note: ignore this section - move to the last edit)

~~I've now changed HVM3's implementation to keep unordered DUP/SUP nodes identical to the ordered versions; that is, I'm not merging UDUP binders into a single port anymore. That greatly simplifies the code, as we don't need to implement a new node type: just store an "unordered" bit on normal DUP/SUP nodes, and reuse the same code. That bit essentially tells the runtime we don't care about the order of elements, allowing it to permute nodes at will. Then, to solve the quadratic issue, it suffices to perform oportunistic swaps on unordered DUP nodes. Specifically, whenever we enter an UDUP node from its aux port, and provided its parent node is also an UDUP node, we will swap these nodes - and that's it.~~

~~The reason this work is that, whenever we access a DUP node, we essentially push it closer to the values it wants to clone, giving it "higher priority". The chain of "redundant" UDUP nodes will still be formed, but it will now be harmless, since redundant nodes will be "pushed away from the action", and be collected later. This solves the quadratic issue, as now we're able to use unordered dups to share immutable datatypes with the same asymptotics of Haskell's thunks.~~

~~In short: the simplest way to solve the quadratic slowdown in an optimal evaluator is to merely store an "unordered" bit on SUP/DUP nodes, and swap UDUP with their parents whenever we access them during the lazy evaluation traversal.~~

~~To be more specific, I use the following commutation rules in HVM3:~~

```
// ,----------------,---------------,
// :    v      v    :    v      v   :
// :    |      |    :    |      |   :
// :   /#\    /_\   :   /#\    /_\  :
// :   | |    | |   :   | |    | |  :
// :  /_\y    a/#\  :  /_\y   /#\b  :
// :  | |     ↑| |  :  | |    | |↑  :
// :  a b      x y  :  a b    x y   :
// :  ↑        b    :    ↑    a     :
// :----------------|---------------|
// :   v       v    :   v       v   :
// :   |       |    :   |       |   :
// :  /#\     /_\   :  /#\     /_\  :
// :  | |     | |   :  | |     | |  :
// : x /_\    a/#\  : x /_\   /#\b  :
// :   | |    ↑| |  :   | |   | |↑  :
// :   a b     x y  :   a b   x y   :
// :   ↑         b  :     ↑     a   :
// '----------------'---------------'
```

~~Or, in textual form:~~

```
// %L{y0 y1} = v
// %L{x0 x1} = y0
// --------------- DUH_DUH
// %L{x0 x1} = v
// %L{y0 y1} = y0
// x1 <- y0
```

~~(And variations.)~~

~~This approach is cleaner than what I wrote earlier not just because it is much simpler to implement, but because it is a proper implementation of unordered sups/dups, allowing us to define all interactions, including UDUP-LAM, without the weird issues and limitations I mentioned earlier. An "unordered duplication" of a lambda might still cause lambdas to swap their bodies, but the operation is well-defined now.~~

~~It is worth noting, though, that unordered sups/dups might break strong confluence, as it is an interaction that happens between a node and its aux port. Specifically, the use of unordered node causes HVM3 to report different interaction counts when certain optimizations are on (i.e., when the graph is traversed in a different order). This is something that I plan to investigate futurely.~~

EDIT:

Oops - after testing and inspecting the memory, I realize the "cleaner" approach above doesn't actually solve the quadratic issue, as the complexity is just moved to swap rules. That said, the initial approach works, and I also realize there is a more sensible definition for the UDUP-LAM rule:

```
{F} = λx(f)
---------------- UDP-LAM
{F} = λx0(G)
{G} = f
x <- {x0 x1}
λx1(G)
```

This says that, to unordered-dup a lambda, we create two fresh lambdas: `λx0`, for our immediate consumption, and `λx1`, for future cloners. Then, we substitute `x` by an unordered superposition, `{x0 x1}`, including both lambda's variables. Finally, we create an unordered duplication of the λ's body, and attach it to both fresh lambdas. Essentially, this creates a mechanism to clone a λ infinitely many times. Each time a clone is made, its bound variable if superposed further and further.

As far as my tests go, this works really well. I also realized that we don't need to store two ports on UDUP nodes; just one. That's because an UDUP node is never "consumed", becoming a substitution. It is always kept alive, serving as an "infinite" (or ref-counted) producer of the value it duplicates. In a way, this also means UDUPs are actually an implementation of non-linear variables on interaction nets, sort of?

Below is an example on how to use that to solve `X + 12345 = 1000000` on HVM3:

```
// Unary Peano Nats
data Nat { #Z #S{pred} }

// Converts an U32 to a Nat
@nat(n) = ~n{
  0: #Z
  _: λnp #S{@nat(np)}
}

// Converts a Nat to an U32
@u32(n) = ~n{
  #Z: 0
  #S: λnp (+ 1 @u32(np))
}

// Adds two Nats
@add(a b) = ~a{
  #Z: b
  #S: λap #S{@add(ap b)}
}

// Compares two Nats for equality
@eq(a b) = ~a{
  #Z: ~b{
    #Z: 1
    #S: λbp 0
  }
  #S: λap ~b{
    #Z: 0
    #S: λbp @eq(ap bp)
  }
}

// A Pseudo Metavar "X"
// (Implemented as a superposition of all Nats)
@X = %0{#Z #S{@X}}

// Collapses a boolean view on 'X', returning it
@colX(X) = !%0{v}=X @colXGo(v)

@colXGo(v) = ~v{
  0: (+ 1 @colXGo(v))
  _: λp 0
}

// Solves X + 1234 = 10000
@main = @colX(@eq(@add(@nat(12345) @X) @nat(1000000)))
```

The program above correctly outputs `987655` in just `24m` interactions, showing it is clearly linear on the length of the Nat (which is `1m`). We've successfully used HVM3 as a logic programming language, even though it isn't!

For completion, below is an snapshot of the entire HVM3 lazy-mode design:

HVM3 SYNTAX

```
// Top-Level Function
@name(x y z) = body

// Expressions
@name(x y z)       // Function Call
x                  // Variable
λx λy λz expr      // Lambda
(f x y z)          // Application
&L{x y}            // Superposition
!%L{x y}=v expr    // Duplication
!x=a expr          // Let binding
#Ctr{x y z}        // Constructor
~x{ #Z:A #S:λp B } // Pattern matching
123                // Number literal
(+ x y)            // Numeric operation
%L{x y}            // Unordered Superposition
!%L{x}=a expr      // Unordered Duplication
```

HVM3 INTERACTIONS

```
@foo
--------- REF
book[foo]

! x = val
bod
--------- LET
x <- val
bod

(* a)
----- APP-ERA
*


(λx(body) a)
-------------- APP-LAM
x <- a
body

(&L{a b} c)
----------------- APP-SUP
! &L{x0 x1} = c
&L{(a x0) (b x1)}

(%L{a b} c)
----------------- APP-USUP
! %L{x} = c
%L{(a x) (b x)}

(#{x y z ...} a)
---------------- APP-CTR
⊥

(123 a)
------- APP-W32
⊥

! &L{x y} = *
------------- DUP-ERA
x <- *
y <- *

! &L{x y} = λz(f)
----------------- DUP-LAM
! &L{f0 f1} = f
x <- λz0(f0)
y <- λz1(f1)
z <- &L{z0 z1}

! &L{x y} = &R{a b}
------------------- DUP-SUP
if L == R:
  x <- a
  y <- b
else:
  x <- &R{a0 b0}
  y <- &R{a1 b1}
  ! &L{a0 a1} = a
  ! &L{b0 b1} = b

! &L{x y} = %R{a b}
------------------- DUP-USUP
x <- %R{a0 b0}
y <- %R{a1 b1}
! &L{a0 a1} = a
! &L{b0 b1} = b

! &L{x y} = #{a b c ...}
----------------------- DUP-CTR
! &L{a0 a1} = a
! &L{b0 b1} = b
! &L{c0 c1} = c
...
x <- #{a0 b0 c0 ...}
y <- #{a1 b1 c1 ...}

! &L{x y} = 123
-------------- DUP-W32
```

```
x <- 123
y <- 123

~ * {K0 K1 K2 ...}
------------------ MAT-ERA
*

~ λx(x) {K0 K1 K2 ...}
---------------------- MAT-LAM
⊥

~ &L{x y} {K0 K1 K2 ...}
------------------------ MAT-SUP
! &L{k0a k0b} = K0
! &L{k1a k1b} = K1
! &L{k2a k2b} = K2
...
&L{ ~ x {K0a K1a K2a ...}
    ~ y {K0b K1b K2b ...} }

~ %L{x y} {K0 K1 K2 ...}
------------------------ MAT-USUP
! %L{k0} = K0
! %L{k1} = K1
! %L{k2} = K2
...
%L{ ~ x {K0 K1 K2 ...}
    ~ y {K0 K1 K2 ...} }

~ #N{x y z ...} {K0 K1 K2 ...}
------------------------------ MAT-CTR
(((KN x) y) z ...)

~ num {K0 K1 K2 ... KN}
----------------------- MAT-W32
if n < N: Kn
else    : KN(num-N)

<op(* b)
--------- OPX-ERA
*

<op(λx(B) y)
------------- OPX-LAM
⊥

<op(&L{x0 x1} y)
------------------------ OPX-SUP
! &L{y0 y1} = y
&L{<op(x0 y0) <op(x1 y1)}

<op(%L{x0 x1} y)
------------------------- OPX-USUP
! %L{y0} = y
%L{<op(x0 y0) <op(x1 y0)}

<op(#{x0 x1 x2...} y)
---------------------- OPX-CTR
⊥

<op(x0 x1)
----------- OPX-W32
<op(x0 x1)

>op(a *)
--------- OPY-ERA
*

>op(a λx(B))
------------- OPY-LAM
⊥

>op(a &L{x y})
------------------------- OPY-SUP
&L{>op(a x) >op(a y)}

>op(a %L{x y})
--------------------- OPY-USUP
%L{>op(a x) >op(a y)}

>op(#{x y z ...} b)
---------------------- OPY-CTR
```

```
⊥

>op(x y)
--------- OPY-W32
x op y

! %L{x} = *
---------- UDUP-ERA
x <- *

! %L{x} = λz(f)
--------------- UDUP-LAM
! %L{G} = f
x <- λz0(G)
z <- %L{z0 z1}
λz1(G)

! %L{x} = &R{a b}
---------------- UDUP-SUP
x <- &R{a0 b0}
! %L{a0} = a
! %L{b0} = b

! %L{x} = %R{a b}
---------------- UDUP-USUP
if L == R:
  ! %L{x} = b
  a
else:
  x <- %R{a0 b0}
  ! %L{a0} = a
  ! %L{b0} = b

! %L{x} = #{a b c ...}
--------------------- UDUP-CTR
! %L{ax} = a
! %L{bx} = b
! %L{cx} = c
...
x <- #{ax bx cx ...}

! %L{x} = 123
------------ UDUP-W32
x <- 123
```

H.54 · bend-docs/openspec/changes/bend-docs-site/research/victor/kind-conv-checker.md

Kind2's Conversion Checker

URL: https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5 (https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5)

FILE: KIND2_EQUALITY.MD

KIND2'S CONVERSION CHECKER

Since Kind2 is based on Self Types, that allows us to dispense a native datatype system in favor of existing λ-Encodings. We can represent ANY inductive family by taking the dependent eliminator of that family and replacing the witness by a self type. For example, Nats are just:

```
Nat
: *
= $self
  ∀(P: ∀(n: Nat) *)
  ∀(succ: ∀(n: Nat) (P (Nat.succ n)))
  ∀(zero: (P Nat.zero))
  (P self)

Nat.succ : ∀(n: Nat) Nat
= λn ~λP λsucc λzero (succ n)

Nat.zero : Nat
= ~λP λsucc λzero zero
```

Notice, though, there are many degrees of recursion in this representation:

1. The Nat.succ constructor stores a Nat.
2. The Nat motive, P, refers to Nat.
3. The Nat constructors Nat.succ and Nat.zero refer to Nat, and vice-versa.

This kind of self-reference is inevitable, because the dependent eliminator of an inductive datatype refers to the type being eliminated. This truth is universal: even in langauges like Coq, this effect is present, it is merely hidden within the type-checker of dependent eliminators.

Since we can't rely on a hard-coded datatype system, that means conversion checking must be capable of dealing with the degree of mutual recursion seen in these encodings, otherwise, it would fail to halt. Most dependent type checkers are NOT capable of handling even simple recursive types; for example, it is impossible to represent Scott Nats *directly* (without wrappers) in Coq, Lean, Agda, and even Haskell. Given that the type above is an *inductive* view of Scott Nats, there would be no hope to type-check it in existing proof assistants.

In Kind1, the solution we adopted for this was to keep a map of "seen" equations. If, during the recursion process, we arrive to an equation that we previously visited, we just return true, short-cutting the recursion. This approach works, but is also complex and slow. Thankfully, Kind2's approach is much simpler, faster, but it is also subtler, in the sense we rely on reductions being performed in a precise order, otherwise it would fall into infinite recursive limbos. The way it works is straightforward:

```
// Equal
A == B ::=
  If A and B are identical:
    return true
  Otherwise:
    reduce A and B to weak normal
    check if A and B are similar

// Similar
A ~~ B ::=
  check if the fields of A and B are equal

// Identical
A <> B ::=
  check if A and B are "textually" the same
```

So, for example, to check if `(pair 1 4) == (pair 1 (+ 2 2))`, we would first check of both sides are identical. They're not, as evidenced by the fact they have different writings. We then reduce A and B to weak normal, and check if they're similar. This will recursivelly check if `1 == 1` and `4 == (+ 2 2)`. The first is already identical. The second becomes identical after a single reduction. And that's it!

As you can see, this is obvious enough. So, what's the catch? Suppose we wanted to check a self-encoded type like `(List #U60) == (List Char)`, where `Char` is just an alias to `#U60`. In this case, we'd have something like:

```
(List #U60) == (List Char)
------------------------------------------------------------------------ // not identical; reduce
$self ∀(P: ∀(x:(List #U60)) *) ... ~~ $self ∀(P: ∀(x:(List Char)) *) ...
------------------------------------------------------------------------ // recurse on components
∀(P: ∀(x:(List #U60)) *) ... == ∀(P: ∀(x:(List Char)) *) ...
------------------------------------------------------------------------ // not identical; reduce
∀(P: ∀(x:(List #U60)) *) ... ~~ ∀(P: ∀(x:(List Char)) *) ...
------------------------------------------------------------------------ // recurse on components
(List #U60) == (List Char)
------------------------------------------------------------------------ // not identical; reduce
... infinite loop ...
```

Now, as inoffensive as this problem may look, solving it often lead us to undesirable directions. It may be a good exercise to pause here and think about how you'd solve it. For example, an obvious idea could be to have a special flag for "constructors" (like `List` and `Char`), and never reduce these during the equality. This would cause the algorithm to check if both sides are component-wise equal BEFORE reducing, which would work on this case. Yet, in other cases where we need to expand a constructor, such as `(not true) == false`, that would fail. Kind1's approach was a solution, albeit over-engineered. There are many "wrong" ways one might approach this problem, that inevitably lead to complexity bloat and alterations on the "natural" equality algorithm that inevitably come to bite us later.

So, how do we solve it on Kind2? First, we must note that this algorithm is, obviously, undecidable in general. So, our first goal must be to cover, at least, the same set of instances that a classical checker like Coq would. The algorithm above accomplishes exactly that. After all, if two terms have the same normal form, this algorithm *will* identify them as equal, as can be easily seen. Coq only deals with normalizable terms, thus, this already covers the same set of programs and proofs!

The only problem is how we deal with terms that are NOT expressive in Coq. In most cases, we don't care! After all, if Kind isn't capable of handling a term that no other checker would, then it doesn't matter. The problem is, the example above IS an instance the algorithm above can't handle, and we NEED it to, because, otherwise, we'd not have a way to encode inductive datatypes! So, how do we modify our equality checker just enough to make it not drown in that specific case, without making it complex, slow or, worse, wrong?

First, notice that this problem is non-existant in existing proof checkers, because they can NOT reduce `(List #U60)` further; `List` is an axiom. Sadly, we can **not** prevent it from unrolling in Kind, since this mechanism is necessary to check constructors properly (otherwise, `λP λt λf t :: Bool` would be a `non-function-application`). So, it is unavoidable that self-types will, depending on the situation, unroll during conversion checking.

As such, what we need is not a mechanism to block reduction, but a mechanism to undo a reduction.

In general, this looks like a terrible idea and a massive complication. But, for self-types specifically, there is a quite straight-forward way to do it: **just annotate the `self` binder**! For example, `List` would be written as:

```
List = λT
  $(self: (List T))
  ∀(P: ∀(xs: (List T)) *)
  ∀(cons: ∀(head: T) ∀(tail: (List T)) (P (List.cons T head tail)))
  ∀(nil: (P (List.nil T)))
  (P self)
```

This looks like a small redundancy, but it allows us to modify the `similar` algorithm to do exactly that: "undo" the reduction on the `self` case. That is, instead of:

```
($x A) ~~ ($y B) ::= (A x) == (B y)
```

We have:

```
($(x:X) A) ~~ ($(y:Y) B) ::= X ~~ Y
```

Note that we do NOT recurse to `==` (equality), but to `~~` (similarity), which will NOT reduce `X` and `Y` and will, instead, check each component separately. This simple change effectively gives a mechanism for the conversion checker to move from the complex expansion of a self-type to its compact applicative form, and proceed to compare it component-wise.

This is correct, because type constructors are injective, so, no false positive can occur by doing this; it merely provides the checker a simpler route. Yet, this WILL result in a false negative: two "identical" self-types (such as `Bit` and `Bool`) will no longer pass the type-checker, because it will **not** try expanding these references. Which may even be seen as a feature, as it gives us a `newtype` functionality for free. (Note that a checker must not give false positives; false negatives are fine.)

This article turned out to be way more verbose, but this is a big problem that deserves to be documented in details. Not in the sense that it is hard, but in the sense that it is really easy to get wrong; so much that Kind1 operated for years with an inefficient solution that was one of its one of its main drawbacks. So, I'm glad to have made some progress in that front, and the new approach looks promising. Now, this all is just for conversion checking; now, on to actual unification. 🤔

This Gist is a snapshot of https://github.com/HigherOrderCO/kind2/blob/c8fd9e54d500a783dc9e428b178647cbcb27235f/docs/equality.md `(https://github.com/HigherOrderCO/kind2/blob/c8fd9e54d500a783dc9e428b178647cbcb27235f/docs/equality.md)`

H.55 · bend-docs/openspec/changes/bend-docs-site/research/victor/term-checker.md

Mini Termination Checker Via Fold Generation

URL: https://gist.github.com/VictorTaelin/676241eb4e290a006e5618f97b8a1c25 (https://gist.github.com/VictorTaelin/676241eb4e290a006e5618f97b8a1c25)

FILE: MINI_TERMINATION_CHECKER_VIA_FOLD_GENERATION.HS

-- == -- MINI TERMINATION CHECKER VIA FOLD GENERATION -- finding the legal recursive calls of a clause, without comparing sizes -- == -- -- Everything here is written in the term language and nothing else: -- -- x a variable -- λx. e a lambda -- (f x) application (n-ary: (f x y)) -- () the unit value -- #0{e} left injection (the first constructor of a sum) -- #1{e} right injection (the second constructor of a sum) -- (e, e') a pair -- λ{#0: o; #1: i} a match -- -- Types: 1 (unit) · A|B (sum) · A&B (pair) · A→B · μX.B · X~[...] (see below). -- The two example types are encodings, nothing built in: -- -- N = μX. 1 | X zero = #0{()} succ p = #1{p} -- N[] = μX. 1 | (N & X) nil = #0{()} cons h t = #1{(h, t)} -- -- -- WHAT WE WANT -- ============ -- A recursive function may only call itself on smaller arguments, or it can -- loop forever. For each clause we want the exact list of calls that are sure -- to be smaller: its FOLDS. Below, the folds sit in place of each body; a fold -- with a leftover λ may be applied to anything in that spot. -- -- F : N → N → N -- F #0{()} b = (no folds) -- F #1{a} #0{()} = λy.(F a y) -- F #1{a} #1{#0{()}} = λy.(F a y) -- F #1{a} #1{#1{b}} = λy.(F a y), (F #1{a} b), (F #1{a} #1{b}) -- -- G : N[] → N -- G #0{()} = (no folds) -- G #1{(#0{()}, t)} = (G t) -- G #1{(#1{p}, t)} = λs.(G #1{(p, s)}), (G t) -- -- These folds ARE the permitted recursive calls. A clause is safe exactly when -- every self-call in its body is one of its folds with the leftover λ's filled -- by anything: λy.(F a y) permits (F a ANYTHING); (F #1{a} b) permits only that -- exact call; λs.(G #1{(p,s)}) permits (G #1{(p, ANYTHING)}). This file only -- GENERATES the folds; checking a body against them is that one extra step. -- -- How could we KNOW these are right? Why is λy.(F a y) fine for any second -- argument, but (F #1{a} b) only for that exact b? Why does a #1 head give -- λs.(G #1{(p,s)}) but a #0 head gives no such call? Why does the last clause, -- which peels the second argument twice and binds b, get BOTH (F #1{a} b) and -- (F #1{a} #1{b})? The rest of the file answers this, never measuring a size. -- -- -- THE OVERVIEW -- ============ -- Walk down the function's type, following the clause's pattern, and carry the -- recursive call with its undecided arguments left as λ-holes. Call that term -- the CARRY. Three things happen on the way down: -- -- SPEC the pattern matches a constructor → wrap it onto the carry. -- SNAP we open a recursive type → save the carry at the spot. -- EMIT the pattern binds a variable → apply each snapshot saved -- there to it; each is one fold. -- -- The folds are just snapshots applied to the variables the pattern binds. -- -- -- THE DETAILS -- =========== -- A recursive type μX.B refers to itself as X. In N = μX. 1 | X the X is the -- RECURSIVE SPOT: where a smaller value of the same type sits (the predecessor -- of #1, the tail of #1{(h,t)}). Snapshots are saved on spots. -- -- Why every fold is smaller. -- A snapshot is saved BEFORE the constructor at that spot is peeled. The -- variable bound there later is what is LEFT after peeling one constructor. -- Feeding it back into the snapshot rebuilds the call with one argument one -- constructor shorter. So it is smaller, always, for free. -- -- Why snapshots are a list, not one. -- A pattern can peel the same spot twice (#1{#1{b}}). Each peel saves another -- snapshot, one layer deeper. Binding the last variable applies them all, one -- fold per depth: the deep snapshot gives (F #1{a} b) (two layers off), the -- shallow one gives (F #1{a} #1{b}) (one layer off). This only happens when a -- clause matches a BOUND variable again; a language that cannot do that keeps a -- single snapshot per spot, so the list is always a singleton there. -- -- Why some λ-holes survive into a fold. -- Applying a snapshot fills one hole; the carry's other holes stay as λ's, and -- can be anything, because once one argument shrinks, everything to its right -- is free. Two cases: a later argument not yet reached → a partial fold, -- λy.(F a y); a sibling field not yet bound → a flexible field, λs.(G #1{(p,s)}). -- -- -- THE CARRY -- ========= -- The carry is the clause's left side, rebuilt as a call as far as we have -- matched, the unfinished spots left as λ-holes. It grows. For f : N → N on the -- clause f #1{a}: -- -- start λx. (f x) one hole -- match #1 λp. (f #1{p}) the hole slid under a #1 -- bind a (f #1{a}) no holes: the clause's own lhs -- -- A pair field ADDS a hole. For g : N[] → N on g #1{(h,t)}: -- -- start λx. (g x) one hole -- match #1 λp. (g #1{p}) one hole -- split pair λh.λt. (g #1{(h,t)}) two holes now (head, tail) -- bind h, t (g #1{(h,t)}) no holes: the lhs again -- -- The final carry is always the clause's own lhs, the same size, never a legal -- call. Folds come from EARLIER carries (the snapshots) applied to the bound -- variables: the snapshot λx.(f x) applied to a is (f a), one constructor smaller. -- -- Strip the bookkeeping and a fold is just the recursive call with ONE argument -- replaced by a structurally smaller variable, the rest left free. The carry as a -- term is one way to compute that; an implementation may instead carry the plain -- list of arguments-so-far and skip the rebuilding entirely. -- -- -- THE ALGORITHM -- ============= -- fold takes the carry c, the type still to walk, the clause skeleton (itself a -- term: a match, a λ that binds, or () for the body), and the folds so far. -- X~~ms is a spot already opened, holding the snapshot list ms. -- -- fold c k () ms fs = ms fs -- fold c (1 → k) s fs = fold (c ()) k s fs -- fold c (A&B → k) s fs = fold (λh.λt. (c (h,t))) (A→B→k) s fs -- fold c (μX.B → k) λ{#0:o;#1:i} fs = fold c (B[X := X][c]] → k) λ{#0:o;#1:i} fs -- fold c (X~~ms → k) λ{#0:o;#1:i} fs = fold c (B[X := X](ms++[c])] → k) λ{#0:o;#1:i} fs -- fold c (A|B → k) λ{#0:o;#1:i} fs = λ{ #0: fold (λw. (c #0{w})) (A→k) o fs -- ; #1: fold (λw. (c #1{w})) (B→k) i fs } -- fold c (X~~ms → k) λv. s fs = λv. fold (c v) k s (fs ++ [(m v) | m ← ms]) -- fold c (A → k) λv. s fs = λv. fold (c v) k s fs -- -- line 1 (body) return the folds collected. -- line 2 (1) a unit field: plug (), no bind, no fold. -- line 3 (A&B) a pair field: SPEC, the carry grows two holes. -- line 4 (μX.B) open a fresh μ to match it: SNAP, save c (ms = [c]). -- line 5 (X~~ms) match an opened spot again: SNAP, add c (ms ++ [c]). -- line 6 (A|B) the match: each branch SPECs its tag and walks its payload. -- line 7 (X~~ms) bind a variable at a spot: EMIT one fold per snapshot. -- line 8

~~(A) bind a variable elsewhere: no snapshot, no fold. -- -- Every line rebuilds the carry the same way — app fills a hole, the constructor -- cases wrap one on — and walks on. Binding is uniform: lines 7 and 8 both do -- (c v); only line 7, sitting at a spot, also emits. There is no per-argument -- special-casing and no notion of "this λ binds a whole argument" versus "a -- field": a bind is a bind, an emit only happens on a spot. -- - - -- WORKED DERIVATIONS -- ========== -- Each derivation reduces one whole function. The dashed line names the rule -- applied next. A leaf shows < carry : remaining-type ; [folds] > while it -- has work, or just its [folds] at a body. A snapshot list is written T[…]. To -- save steps, a #0~~ branch fills its () and a #1 branch splits its pair in the -- same match step, so neither shows as a binder. The leaves are the clauses. -- -- -- F : N → N → N -- -- < F : N → N → N ; [] > -- ---------- unroll-N -- < F : (1 | N~[F]) → N → N ; [] > -- ---------- mat-N -- λ{ -- #0: < (F #0{()}) : N → N ; [] >; -- #1: < λp.(F #1{p}) : N~[F] → N → N ; [] > -- } -- ---------- abs-N -- λ{ -- #0: λb. []; -- #1: λa. < (F #1{a}) : N → N ; [λy.(F a y)] > -- } -- ---------- unroll-N -- λ{ -- #0: λb. []; -- #1: λa. < (F #1{a}) : (1 | N~[(F #1{a})]) → N ; [λy.(F a y)] > -- } -- ---------- mat-N -- λ{ -- #0: λb. []; -- #1: λa. λ{ -- #0: [λy.(F a y)]; -- #1: < λp.(F #1{a} #1{p}) : N~[(F #1{a})] → N ; [λy.(F a y)] > -- } -- } -- ---------- unroll-N -- λ{ -- #0: λb. []; -- #1: λa. < (F #1{a}) : (1 | N~[(F #1{a})]) → N ; [λy.(F a y)] > -- } -- ---------- mat-N -- λ{ -- #0: λb. []; -- #1: λa. λ{ -- #0: [λy.(F a y)]; -- #1: < λp.(F #1{a} #1{p}) : N~[(F #1{a})] → N ; [λy.(F a y)] > -- } -- } -- ---------- unroll-N -- λ{ -- #0: λb. []; -- #1: λa. λ{ -- #0: [λy.(F a y)]; -- #1: < λp.(F #1{a} #1{p}) -- : (1 | N~[(F #1{a}), λp.(F #1{a} #1{p})]) → N -- ; [λy.(F a y)] > -- } -- } -- ---------- mat-N -- λ{ -- #0: λb. []; -- #1: λa. λ{ -- #0: [λy.(F a y)]; -- #1: λ{ -- #0: [λy.(F a y)]; -- #1: < λp.(F #1{a} #1{#1{p}}) -- : N~[(F #1{a}), λp.(F #1{a} #1{p})] → N -- ; [λy.(F a y)] > -- } -- } -- } -- ---------- abs-N -- λ{ -- #0: λb. []; -- #1: λa. λ{ -- #0: [λy.(F a y)]; -- #1: λ{ -- #0: [λy.(F a y)]; -- #1: λb. [λy.(F a y), (F #1{a} b), (F #1{a} #1{b})] -- } -- } -- } -- -- -- G : N[] → N -- -- < G : N[] → N ; [] > -- ---------- unroll-N[] -- < G : (1 | (N & N[]~~[G]~~)) → N ; [] > -- ---------- mat-N[] -- λ{ -- #0: []; -- #1: < λh.λt.(G #1{(h,t)}) : N → N~~[]~~[G] → N ; [] > -- } -- ---------- unroll-N -- λ{ -- #0: []; -- #1: < λh.λt.(G #1{(h,t)}) -- : (1 | N~[λh.λt.(G #1{(h,t)})]) → N[]~~[G] → N -- ; [] > -- } -- ---------- mat-N -- λ{ -- #0: []; -- #1: λ{ -- #0: < λt.(G #1{(#0{()},t)}) : N~~[]~~[G] → N ; [] >; -- #1: < λp.λt.(G #1{(#1{p},t)}) -- : N~[λh.λt.(G #1{(h,t)})] → N[]~~[G] → N -- ; [] > -- } -- } -- ---------- abs-N[] -- λ{ -- #0: []; -- #1: λ{ -- #0: λt. [(G t)]; -- #1: < λp.λt.(G #1{(#1{p},t)}) -- : N[λh.λt.(G #1{(h,t)})] → N[]~~[G] → N -- ; [] > -- } -- } -- ---------- abs-N -- λ{ -- #0: []; -- #1: λ{ -- #0: λt. [(G t)]; -- #1: λp. < λt.(G #1{(#1{p},t)}) : N~~[]~~[G] → N ; [λs.(G #1{(p,s)})] > -- } -- } -- ---------- abs-N[] -- λ{ -- #0: []; -- #1: λ{ -- #0: λt. [(G t)]; -- #1: λp. λt. [λs.(G #1{(p,s)}), (G t)] -- } -- } -- -- ==========

```
module Main where

import Data.List (intercalate) import Control.Monad (forM_)

-- Types

<hr>

-- Term ::= x | λx.f | (f x) | () | #0{x} | #1{x} | λ{#0:o;#1:i} | (x,y) | [folds] -- Type ::= 1 | A|B | A&B | A→B | μX.T | T~[carry,...] -- Mat, Lam and One double as the clause skeleton; Fds holds a body's folds.

type Name = String

data Term = Var Name | Lam (Term -> Term) | App Term Term | One | Inl Term | Inr Term | Mat Term Term | Con Term Term | Fds [Term]

-- Rec holds its own unrolling and the snapshots saved while reaching it. data Type = Uni | Eit Type Type | Par Type Type | Fun Type Type | Fix (Type -> Type) | Rec (Type -> Type) [(Term, Type)]

type Test = (Name, Term, Type, Term)

-- Stringification

<hr>

name :: Int -> Name name n = ['a' .. 'z'] !! n]

-- flatten an application into head and arguments: (App (App F a) b) -> [F,a,b] spine :: Term -> [Term] spine (App f x) = spine f ++ [x] spine t = [t]

show_term :: Int -> Term -> String show_term d (Fds fs) = "[" ++ intercalate ", " (map (show_term d) fs) ++ "]" show_term d (Mat o i) = "λ{#0: " ++ show_term d o ++ "; #1: " ++ show_term d i ++ "}" show_term d (Lam f) = "λ" ++ name d ++ ". " ++ show_term (d + 1) (f (Var (name d))) show_term d (App f x) = "(" ++ unwords (map (show_term d) (spine (App f x))) ++ ")"
```

show_term _ (Var x) = x show_term _ One = "()" show_term d (Inl x) = "#0{" ++ show_term d x ++ "}" show_term d (Inr x) = "#1{" ++ show_term d x ++ "}" show_term d (Con a b) = "(" ++ show_term d a ++ ", " ++ show_term d b ++ ")"

-- Carry application

<hr>

-- Fill the carry's outermost hole, and nothing else: (λx.f)(v) reduces, anything -- else stays applied. This is NOT the language's evaluator. The carry's only -- redexes are the holes we put there with Lam; its head (F, G) is a plain Var and -- is never the function of an App, so the recursive call is never run. Keep this -- restraint when porting: there F is a real reference, and a true evaluator would -- unfold and EXECUTE the very call we are trying to inspect (and likely diverge). app :: Term -> Term -> Term app (Lam f) x = f x app f x = App f x

-- Termination

<hr>

-- folds tm fn ty fs walks the type ty along the skeleton tm, carrying the carry -- fn and the folds fs so far, returning the skeleton with each body replaced by -- its fold list. The eight clauses are the eight lines of THE ALGORITHM. folds :: Term -> Term -> Type -> [Term] -> Term folds One _ _ fs = Fds fs folds tm fn (Fun Uni b) fs = folds tm (app fn One) b fs folds tm fn (Fun (Par ax ay) b) fs = folds tm (Lam (\h -> Lam (\t -> app fn (Con h t)))) (Fun ax (Fun ay b)) fs folds (Mat o i) fn (Fun (Fix nf) b) fs = folds (Mat o i) fn (Fun (nf (Rec nf [(fn, Fun (Fix nf) b)])) b) fs folds (Mat o i) fn (Fun (Rec nf ms) b) fs = folds (Mat o i) fn (Fun (nf (Rec nf (ms ++ [(fn, Fun (Rec nf ms) b)]))) b) fs folds (Mat o i) fn (Fun (Eit l r) b) fs = Mat (folds o (Lam (\w -> app fn (Inl w))) (Fun l b) fs) (folds i (Lam (\w -> app fn (Inr w))) (Fun r b) fs) folds (Lam g) fn (Fun (Rec _ ms) b) fs = Lam (\v -> folds (g v) (app fn v) b (fs ++ map (\m -> app (fst m) v) ms)) folds (Lam g) fn (Fun _ b) fs = Lam (\v -> folds (g v) (app fn v) b fs) folds _ _ _ _ = error "folds: ill-formed skeleton"

-- Tests

<hr>

nat :: Type nat = Fix (\x -> Eit Uni x)

lst :: Type lst = Fix (\x -> Eit Uni (Par nat x))

-- each carry is the recursive call eta-expanded to the function's arity, so a -- partial fold keeps a visible λ per argument not yet supplied selfF :: Term selfF = Lam (\x -> Lam (\y -> App (App (Var "F") x) y))

selfG :: Term selfG = Lam (\x -> App (Var "G") x)

-- a skeleton is a case tree: Mat is a match, Lam binds a variable, One is a body. -- Each test runs one whole function; its leaves are that function's clauses. tests :: [Test] tests = [("F : N -> N -> N", selfF, Fun nat (Fun nat nat), Mat (Lam (_ -> One)) (Lam (_ -> Mat One (Mat One (Lam (_ -> One)))))), ("G : N[] -> N", selfG, Fun lst nat, Mat One (Mat (Lam (_ -> One)) (Lam (_ -> Lam (_ -> One)))))]

main :: IO () main = do forM_ tests $ (lbl, fn, ty, sk) -> do putStrLn ("-- " ++ lbl) putStrLn (show_term 0 (folds sk fn ty [])) putStrLn ""

PART I · HVM4 runtime documentation (HigherOrderCO/HVM4 clone, verbatim)

The HVM4 repository's own docs (C runtime 'before launch'): README + agent onboarding, docs/primer.md, docs/theory/interaction_calculus.md, and docs/hvm/ core (term AST + grammar), memory (term layout, linked/quoted terms), syntax (parser/desugaring), collapser (CNF readback) and ffi, followed by all 61 per-rule sheets in docs/hvm/interactions (APP-LAM, DUP-SUP, … one file per WNF interaction, each mirroring the matching C module's sequent-calculus comment). Ends with the repository's whole test suite (216 tiny .hvm programs with expected outputs in comments) as one aggregated section — raw material for exercises.

Source root: `forgecode_clone/hvm` · **71 files · 138.4 KB**

▼ In this part (file list)

- **I.1** · `forgecode_clone/hvm/README.md`
- **I.2** · `forgecode_clone/hvm/CLAUDE.md`
- **I.3** · `forgecode_clone/hvm/docs/primer.md`
- **I.4** · `forgecode_clone/hvm/docs/theory/interaction_calculus.md`
- **I.5** · `forgecode_clone/hvm/docs/hvm/core.md`
- **I.6** · `forgecode_clone/hvm/docs/hvm/memory.md`
- **I.7** · `forgecode_clone/hvm/docs/hvm/syntax.md`
- **I.8** · `forgecode_clone/hvm/docs/hvm/collapser.md`
- **I.9** · `forgecode_clone/hvm/docs/hvm/ffi.md`
- **I.10** · `forgecode_clone/hvm/docs/hvm/interactions/alo_cop.md`
- **I.11** · `forgecode_clone/hvm/docs/hvm/interactions/alo_dry.md`
- **I.12** · `forgecode_clone/hvm/docs/hvm/interactions/alo_dup.md`
- **I.13** · `forgecode_clone/hvm/docs/hvm/interactions/alo_lam.md`
- **I.14** · `forgecode_clone/hvm/docs/hvm/interactions/alo_nam.md`
- **I.15** · `forgecode_clone/hvm/docs/hvm/interactions/alo_nod.md`
- **I.16** · `forgecode_clone/hvm/docs/hvm/interactions/alo_var.md`
- **I.17** · `forgecode_clone/hvm/docs/hvm/interactions/and_era.md`
- **I.18** · `forgecode_clone/hvm/docs/hvm/interactions/and_inc.md`
- **I.19** · `forgecode_clone/hvm/docs/hvm/interactions/and_num.md`
- **I.20** · `forgecode_clone/hvm/docs/hvm/interactions/and_sup.md`
- **I.21** · `forgecode_clone/hvm/docs/hvm/interactions/app_ctr.md`
- **I.22** · `forgecode_clone/hvm/docs/hvm/interactions/app_dry.md`
- **I.23** · `forgecode_clone/hvm/docs/hvm/interactions/app_era.md`
- **I.24** · `forgecode_clone/hvm/docs/hvm/interactions/app_inc.md`
- **I.25** · `forgecode_clone/hvm/docs/hvm/interactions/app_lam.md`
- **I.26** · `forgecode_clone/hvm/docs/hvm/interactions/app_mat_ctr.md`
- **I.27** · `forgecode_clone/hvm/docs/hvm/interactions/app_mat_num.md`
- **I.28** · `forgecode_clone/hvm/docs/hvm/interactions/app_mat_sup.md`
- **I.29** · `forgecode_clone/hvm/docs/hvm/interactions/app_nam.md`
- **I.30** · `forgecode_clone/hvm/docs/hvm/interactions/app_sup.md`
- **I.31** · `forgecode_clone/hvm/docs/hvm/interactions/ddu_era.md`
- **I.32** · `forgecode_clone/hvm/docs/hvm/interactions/ddu_inc.md`
- **I.33** · `forgecode_clone/hvm/docs/hvm/interactions/ddu_num.md`
- **I.34** · `forgecode_clone/hvm/docs/hvm/interactions/ddu_sup.md`
- **I.35** · `forgecode_clone/hvm/docs/hvm/interactions/dsu_era.md`
- **I.36** · `forgecode_clone/hvm/docs/hvm/interactions/dsu_inc.md`
- **I.37** · `forgecode_clone/hvm/docs/hvm/interactions/dsu_num.md`
- **I.38** · `forgecode_clone/hvm/docs/hvm/interactions/dsu_sup.md`
- **I.39** · `forgecode_clone/hvm/docs/hvm/interactions/dup_dry.md`

- I.40 · forgecode_clone/hvm/docs/hvm/interactions/dup_lam.md
- I.41 · forgecode_clone/hvm/docs/hvm/interactions/dup_nam.md
- I.42 · forgecode_clone/hvm/docs/hvm/interactions/dup_nod.md
- I.43 · forgecode_clone/hvm/docs/hvm/interactions/dup_sup.md
- I.44 · forgecode_clone/hvm/docs/hvm/interactions/eql_any.md
- I.45 · forgecode_clone/hvm/docs/hvm/interactions/eql_ctr.md
- I.46 · forgecode_clone/hvm/docs/hvm/interactions/eql_dry.md
- I.47 · forgecode_clone/hvm/docs/hvm/interactions/eql_era.md
- I.48 · forgecode_clone/hvm/docs/hvm/interactions/eql_inc.md
- I.49 · forgecode_clone/hvm/docs/hvm/interactions/eql_lam.md
- I.50 · forgecode_clone/hvm/docs/hvm/interactions/eql_mat.md
- I.51 · forgecode_clone/hvm/docs/hvm/interactions/eql_nam.md
- I.52 · forgecode_clone/hvm/docs/hvm/interactions/eql_num.md
- I.53 · forgecode_clone/hvm/docs/hvm/interactions/eql_sup.md
- I.54 · forgecode_clone/hvm/docs/hvm/interactions/eql_use.md
- I.55 · forgecode_clone/hvm/docs/hvm/interactions/mat_inc.md
- I.56 · forgecode_clone/hvm/docs/hvm/interactions/op2_era.md
- I.57 · forgecode_clone/hvm/docs/hvm/interactions/op2_inc.md
- I.58 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_era.md
- I.59 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_num.md
- I.60 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_sup.md
- I.61 · forgecode_clone/hvm/docs/hvm/interactions/op2_sup.md
- I.62 · forgecode_clone/hvm/docs/hvm/interactions/or_era.md
- I.63 · forgecode_clone/hvm/docs/hvm/interactions/or_inc.md
- I.64 · forgecode_clone/hvm/docs/hvm/interactions/or_num.md
- I.65 · forgecode_clone/hvm/docs/hvm/interactions/or_sup.md
- I.66 · forgecode_clone/hvm/docs/hvm/interactions/uns.md
- I.67 · forgecode_clone/hvm/docs/hvm/interactions/use_era.md
- I.68 · forgecode_clone/hvm/docs/hvm/interactions/use_inc.md
- I.69 · forgecode_clone/hvm/docs/hvm/interactions/use_sup.md
- I.70 · forgecode_clone/hvm/docs/hvm/interactions/use_val.md
- I.71 · forgecode_clone/hvm/test/*.hvm (aggregated)

I.1 · forgecode_clone/hvm/README.md

HVM

HVM is a high-performance runtime for the Interaction Calculus.

NOTE: you're here before launch. Use at your own risk.

BUILDING AND RUNNING

```
# Build
cd clang && clang -O2 -o main main.c
# On Linux, add: -ldl

# Run a file (use collapse mode by default)
./clang/main test/file.hvm -s -C10

# Run all tests
./test/_all_.sh

# Run benchmarks (from sibling bench repo)
cd ../bench && ./bench.ts --hvm-interpreted
```

Flags:

- `-s` shows performance stats

- `-D` prints each intermediate reduction step with interaction labels
- `-C10` collapses and flattens superpositions (limit to 10 lines)
- `--to-c` emits a standalone AOT C program to stdout
- `--as-c` emits + compiles + runs a standalone executable once
- `-o <path>`, `--output <path>` emits + compiles a standalone executable to a file path
- `--ffi <path>` loads one FFI shared library before parsing
- `--ffi-dir <path>` loads all FFI shared libraries in a directory before parsing

EXAMPLES

```
@main = ((@add 1) 2)
//3
```

```
@main = (&{1, 2} + 10)
//11
//12
```

```
@main = (! x &A= 3; (x₀ + x₁))
//6
```

DOCUMENTATION

- Theory: docs/theory/interaction_calculus.md
- Core language: docs/hvm/core.md
- Memory layout: docs/hvm/memory.md
- Interaction rules: docs/hvm/interactions/

I.2 · forgecode_clone/hvm/CLAUDE.md

AGENTS.md

QUICK ONBOARDING

HVM is a runtime for the Interaction Calculus (IC), a lambda-calculus extension with explicit duplication (DUP) and superposition (SUP). These primitives enable optimal sharing for lazy evaluation, even inside lambdas. This repo is the C runtime: parse source -> build static book terms -> lazily allocate dynamic heap terms -> reduce with WNF/SNF interactions -> print results.

Key terms:

- static/book term: immutable definition stored in the book (de Bruijn levels).
- dynamic term: mutable heap term used during evaluation (linked by pointers).
- WNF: weak normal form (head reduction with interactions).
- SNF: strong normal form (full reduction).
- CNF: collapsed normal form (full lambda-calculus readback).
- interaction: a rewrite rule (APP-LAM, DUP-SUP, etc.).

BUILD AND TEST

```
# Build
cd clang && clang -O2 -o main main.c

# Run a file
./clang/main test/file.hvm -s -C10

# Run tests (interpreted, then AOT compiled)
./test/_all_.sh

# Run benchmarks (from the unified bench repo)
cd ../bench && ./bench.ts --hvm-interpreted
cd ../bench && ./bench.ts --hvm-compiled
```

DOCS MAP

- `README.md` : entry point, build/run examples, links.
- `STYLEGUIDE.md` : authoritative C style rules for `clang/` (mirrored in `clang/STYLE.md`).
- `docs/primer.md` : quick intro to the language and runtime usage.
- `docs/theory/interaction_calculus.md` : IC theory + examples.
- `docs/hvm/core.md` : core term AST and grammar.
- `docs/hvm/syntax.md` : parser syntax, precedence, and desugaring rules.
- `docs/hvm/memory.md` : term layout, heap representation, linked/quoted terms.
- `docs/hvm/collapser.md` : CNF readback and collapse algorithm.
- `docs/hvm/interactions/*.md` : one file per WNF interaction; mirrors the sequent calculus comment in the matching `clang/wnf/<name>.c`.

CODE MAP (C RUNTIME)

TOP-LEVEL ENTRY

- `clang/hvm.c` : single translation unit; defines tags/bit layout/globals and includes every module in build order. Start here to understand the whole runtime.
- `clang/main.c` : CLI entry point and runtime setup.

TERM REPRESENTATION

- `clang/term/new.c` : pack a term word from fields.
- `clang/term/tag.c` : extract tag.
- `clang/term/ext.c` : extract ext.
- `clang/term/val.c` : extract val.
- `clang/term/arity.c` : arity per tag.
- `clang/term/clone.c` : duplication helper.
- `clang/term/sub/get.c` : read SUB bit.
- `clang/term/sub/set.c` : set/clear SUB bit.
- `clang/term/new/*.c` : constructors for each tag; `clang/term/new/_.c` allocates heap nodes.

PARSER

- `clang/parse/*.c` : lexer utilities, binding stack, and definition parsing.
- `clang/parse/term/*.c` : term parsers; `clang/parse/term/_.c` dispatches.

EVALUATION

- `clang/wnf/_.c` : stack-based WNF evaluator and interaction dispatch.
- `clang/wnf/*.c` : one interaction per file; see matching doc in `docs/hvm/interactions/`.
- `clang/eval/normalize.c` : SNF normalization (WNFs every reachable node).

COLLAPSE (CNF READBACK)

- `clang/cnf/_.c` : lift one SUP to the top, without recursing into branches.
- `clang/eval/collapse.c` : BFS enumeration of SUP branches; prints quoted SNF.

- `clang/data/pq.c` : priority queue for collapse ordering (INC affects priority).
- `clang/data/wspq.c` : work-stealing priority queue for parallel collapse.

PRINTING AND NAMES

- `clang/print/term.c` : term pretty-printer (dynamic/static modes).
- `clang/print/name.c` : alpha name generation.
- `clang/print/utf8.c` : UTF-8 printing helpers.
- `clang/nick/*.c` : base64-ish name encoding/decoding utilities.
- `clang/table/*.c` : global name table (id <-> string) with lookup helpers.

HEAP AND SYSTEM

- `clang/heap/alloc.c` : heap allocation helpers.
- `clang/heap/subst_var.c` : install lam substitutions.
- `clang/heap/subst_cop.c` : install dup substitutions.
- `clang/prim/init.c` : register built-in primitives at startup.
- `clang/prim/register.c` : primitive registration helpers.
- `clang/prim/fn/*.c` : primitive implementations.
- `clang/thread/get_count.c` : read worker thread count.
- `clang/thread/set_count.c` : set worker thread count.
- `clang/sys/error.c` : error formatting.
- `clang/sys/file_read.c` : file read utility.
- `clang/sys/path_join.c` : path joining.
- `clang/prelude/_.c` : prelude stub (empty).

NAMING RULE (CRITICAL)

The file path is the function name: replace `/` with `_`, drop `.c`. Example: `wnf/app_lam.c` defines `wnf_app_lam()`. `_.c` represents the directory root.

I.3 · `forgecode_clone/hvm/docs/primer.md`

HVM Primer

HVM is a runtime for the Interaction Calculus, a model of computation that extends lambda calculus to achieve **optimal reduction**: no work is ever duplicated, even when copying functions. It's lazy like Haskell, but sharing extends inside lambdas, not just at the data level.

RUNNING HVM

```
hvm file.hvm -s -C10 # run with collapse
```

Flags:

- `-s` : show performance stats
- `-D` : print intermediate reduction steps
- `-C10` : collapse superpositions (limit output to 10 lines)

Modes:

- *Normalize*: reduce to SNF. Keeps SUPs and DUPs in output.
- *Collapse*: eliminate SUPs and DUPs, enumerate as pure lambda terms.

GRAMMAR

```
File     ::= (Include | Def)*
Include  ::= "#include" '"' Path '"'
Def      ::= "@" Name "=" Term

Term ::=
  | Name                                  -- variable
  | Name "₀" | Name "₁"                   -- dup variable (first/second)
  | Integer                               -- number literal
  | "@" Name                              -- reference to definition
  | "*"                                   -- wildcard (erased)

  -- Lambda & Application
  | "λ" Params "." Term                   -- lambda: λx.body
  | "λ" "&" Name "." Term                 -- cloned lambda: λ&x.body
  | "λ" Name "&" Label "." Term           -- lambda with inline dup
  | Term "(" Args ")"                     -- application: f(x,y)

  -- Pattern Matching
  | "λ" "{" Cases "}"                     -- match lambda
  | "λ" "{" Term "}"                      -- use (strict unbox)

  -- Duplication & Superposition
  | "!" Name "&" Label "=" Term ";" Term  -- dup: !x&L=val;body
  | "!" Name "&" "(" Term ")" "=" Term ";" Term -- dynamic dup
  | "&" Label "{" Term "," Term "}"       -- superposition: &L{a,b}
  | "&" "(" Term ")" "{" Term "," Term "}" -- dynamic superposition
  | "&" "{" "}"                           -- erasure

  -- Let Bindings (sugar)
  | "!" Name "=" Term ";" Term            -- let: !x=val;body
  | "!" "&" Name "=" Term ";" Term        -- cloned let
  | "!!" Name "=" Term ";" Term           -- strict let

  -- Constructors
  | "#" Name "{" Args "}"                 -- constructor: #Pair{a,b}
  | "#" Name "{" "}"                      -- nullary: #ZER{} or #True{}

  -- Operators
  | Term Op Term                          -- binary: (a + b), (x == y)
  | Term "===" Term                       -- structural equality
  | Term ".&." Term                       -- short-circuit AND
  | Term ".|." Term                       -- short-circuit OR

  -- Special
  | "↑" Term                              -- priority (collapse order)
  | "&" Label "λ" Names "{" Term ";" Term "}"  -- fork
  | "[" Args "]"                          -- list: [a,b,c]
  | '"' Chars '"'                         -- string
  | "'" Char "'"                          -- char: 'x' -> #CHR{120}
  | Integer "n"                           -- nat: 2n -> #SUC{#SUC{#ZER}}

Cases    ::= (Pattern ":" Term ";")* [Default]
Pattern  ::= "#" Name | Integer | "[]" | "<>" | Integer "n" | Integer "n+"

Operators (low to high): || && == != < <= > >= << >> + - * / % ^
```

KEY CONCEPTS

AFFINE VARIABLES

Variables are **affine**: used at most once.

```
// WRONG: x used twice
@bad = λx. (x + x)

// RIGHT: use cloned binder
@square = λ&x. (x * x)
@main = @square(5)
//25
```

The `&` prefix tells the parser to auto-insert duplication nodes.

Common affinity errors:

1. Using a variable in multiple match branches:

```
// WRONG: x appears in both branches
@bad = λx. λ{
  #T: x
  #F: x
}

// RIGHT: match first, then bind x in each branch
@good = λ{
  #T: λx. x
  #F: λx. x
}
@main = @good(#T{}, 42)
//42
```

2. Using unnecessary clones:

```
// BAD: inline λ{...}(x) is poor style

@filter = λ&f. λ{
  []: []
  <>: λ&h. λ&t.
    λ{
      #T: h <> @filter(f, t)
      #F: @filter(f, t)
    }(f(h))
}

// GOOD: split into two functions

@keep = λ{
  #T: λh. λt. h <> t
  #F: λh. λt. t
}

@filter = λ&f. λ{
  []: []
  <>: λ&h. λt. @keep(f(h), h, @filter(f, t))
}
```

It is very important, on HVM, to avoid dups as much as possible. As such, it is considered a good style to format all top-level definitions as case-trees, rearranging arguments to avoid unecessary clones, and avoiding inline λ-match applications (such as `λ{...}(x)`).

SUPERPOSITION (SUP)

A SUP `&L{a,b}` represents two values in one location:

```
@main = (&X{10, 20} + 5)
//15
//25
```

When collapsed (`-C` flag), each branch prints separately.

LABEL MECHANICS

Labels control how DUPs interact with SUPs:

- **Same label**: DUP and SUP annihilate (pair extraction)
- **Different labels**: they commute (cross product)

To observe this, use explicit DUP syntax `!x&L = val; body` which binds x_0 and x_1:

```
// Same label: pairwise extraction
@main =
  !x&A = &A{1, 2};
  [x₀, x₁]
//[1,2]
```

```
// Different labels: the DUP stays inside each SUP branch
@main =
  !x&A = &B{1, 2};
  [x₀, x₁]
//[1,1]
//[2,2]
```

```
// Cross product from two SUPs with different labels (4 results)
@main = [&A{1, 2}, &B{10, 20}]
//[1,10]
//[1,20]
//[2,10]
//[2,20]
```

Prefer `λ&x` for general functions. Use explicit DUPs only when you need label control (pairwise extraction or explicit SUP/DUP interactions).

PATTERN MATCHING

Match on constructors with `λ{...}`:

```
@not = λ{
  #T: #F{}
  #F: #T{}
}
@main = @not(#T{})
//#F{}
```

```
@pred = λ{
  0n:  0n
  1n+: λp. p
}
@main = @pred(3n)
//2n
```

Match cases bind constructor fields as lambda arguments.

PEANO NATURALS

`Nn` is sugar for `#SUC{...#SUC{#ZER}...}`. Patterns: `0n:` matches `#ZER`, `1n+:` matches `#SUC` and binds predecessor:

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}
@main = @add(2n, 3n)
//5n
```

CHARACTERS

A char literal `'x'` desugars to `#CHR{120}` (the constructor `#CHR` wrapping the Unicode codepoint as a number). Escape sequences: `'\n'`, `'\t'`, `'\r'`, `'\0'`, `'\\'`, `'\''`, `'\"'`.

```
@main = 'A'
//'A'
```

To match on a character, match the `#CHR` constructor to extract the codepoint, then switch on the numeric value:

```
@is_a = λ{
  #CHR: λ{'A': #T; λn. #F}
}

@main = @is_a('A')
//#T{}
```

LISTS

Lists use the built-in constructors `#CON` and `#NIL` (all caps).

Sugar:

- `[]` desugars to `#NIL{}`
- `[a, b, c]` desugars to `#CON{a, #CON{b, #CON{c, #NIL{}}}}`
- `a <> b` desugars to `#CON{a, b}` (cons)

Pattern sugar:

- `[]:` matches `#NIL`
- `<>:` matches `#CON`

```
@len = λ{
  []: 0
  <>: λh. λt. (1 + @len(t))
}
@main = @len([1, 2, 3, 4, 5])
//5
```

STRINGS

Strings are lists of `#CHR`. `"hi"` desugars to:

#CON{#CHR{104}, #CON{#CHR{105}, #NIL{}}}

Backtick can be used instead of quotes.

```
@main = "hi"
//"hi"
```

Since strings are just lists, list functions work on them directly:

```
...
@main = @len("hello")
//5
```

NUMERIC PATTERNS

Switch on machine ints with `0:`, `1:`, etc., and `λn.` for default:

```
@fib = λ{
  0: 0
  1: 1
  λ&n. (@fib((n - 1)) + @fib((n - 2)))
}
@main = @fib(10)
//55
```

FORK NOTATION

`&Lλx,y{A;B}` is sugar for parallel branching. A uses x_0, y_0; B uses x_1, y_1:

```
@main = (&Lλx,y{(x + y); (x * y)})(3, 4)
//7
//12
```

This desugars to:

```
@main =
  !x&L = 3;
  !y&L = 4;
  &L{(x₀ + y₀), (x₁ * y₁)}
//7
//12
```

UNSCOPED LAMBDAS

Unscoped lambdas allow a variable to be used outside its binder's lexical scope. You can write them directly as:

```
λ$x. body
```

This is sugar for:

```
! f = λ x ; body
```

This declares in `body`:

- `f = λy. λ$x. y` — a function that wraps its argument in an unscoped lambda
- `x = $x` — an unscoped variable usable anywhere

So `λ$x.body` is equivalent to constructing it as `f(body)`:

```
@main = (λ$x. (x + 1))(10)
//11
```

Applying the unscoped lambda to `10` binds `x` to `10`, yielding `11`. (Equivalent sugar: `! f = λ x ; f(x + 1)(10)`.)

Another example using the unscoped variable in a pair:

```
@main = ! f = λ v ; #P{f(1, 2), v}
//#P{1,2}
```

Here `f(1, 2)` is `f(1)(2)` which produces `λ$v.1` then applies it to `2`, binding `v` to `2`. The pair contains `1` and the captured value `2`.

This is useful for continuation-passing or "exporting" a binding out of a nested context.

EXAMPLES

FACTORIAL WITH PEANO NATURALS

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}

@mul = λ{
  0n:  λb. 0n
  1n+: λa. λ&b. @add(b, @mul(a, b))
}

@fac = λ{
  0n:  1n
  1n+: λ&p. @mul(1n+p, @fac(p))
}

@main = @fac(3n)
//6n
```

LIST MAP

```
@map = λ&f. λ{
  []: []
  <>: λh. λt. (f(h) <> @map(f, t))
}
@main = @map(λx. (x * 2), [1, 2, 3])
//[2,4,6]
```

ENUMERATING BINARY STRINGS

Superpositions can enumerate multiple results; each branch prints separately in collapse mode:

```
@bits = λ{
  0n:  #E{}
  1n+: λ&p.
    &B{#O{@bits(p)}, #I{@bits(p)}}
}
@main = @bits(2n)
//#O{#O{#E{}}}
//#O{#I{#E{}}}
//#I{#O{#E{}}}
//#I{#I{#E{}}}
```

CHURCH NUMERALS

Standard lambda encoding with a cloned binder for `s`:

```
@c2   = λ&s. λz. s(s(z))
@cmul = λa. λb. λs. λz. a(b(s), z)
@main = @cmul(@c2, @c2)
//λa.λb.a(a(a(a(b))))
```

SOLVING EQUATIONS VIA ENUMERATION

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}

@eq = λ{
  0n:  λ{
    0n:  #T{}
    1n+: λp. #F{}
  }
  1n+: λa. λ{
    0n:  #F{}
    1n+: λb. @eq(a, b)
  }
}

@if = λ{
  #T: λt. λf. t
  #F: λt. λf. f
}

// All naturals: 0, 1, 2, ...
@X = &N{0n, 1n+@X}

// Solve X + 2 = 4
@main = @if(@eq(@add(@X, 2n), 4n), @X, &{})
//2n
```

FILTER

```
@even = λ{
  0: #T{}
  1: #F{}
  λp. @even((p - 2))
}

@filter = λ&f. λ{
  []: []
  <>: λ&h. λt. @keep(f(h), h, @filter(f, t))
}

@keep = λ{
  #T: λh. λr. (h <> r)
  #F: λh. λr. r
}

@main = @filter(@even, [0, 1, 2, 3, 4, 5, 6])
//[0,2,4,6]
```

COMMON PATTERNS

- `λ&x.` : Cloned binder (auto-dup)
- `!x&L = v; body` : Explicit dup with label L
- `x₀ , x₁` : Dup variable branches
- `&L{a,b}` : Superposition
- `&{}` : Erasure (discard value)
- `a <> b` : List cons
- `@name` : Reference to definition

Constructor match:

```
λ{
  #C: ...
}
```

Peano natural match:

```
λ{
  0n:  ...
  1n+: λp. ...
}
```

Numeric switch (machine ints):

```
λ{
  0:  ...
  1:  ...
  λn. ...
}
```

GOTCHAS

1. **No whitespace application**: use `f(x)` not `f x`
2. **Variables are affine**: use `&` for multiple uses
3. **Subscripts required**: dup-bound vars need `₀` or `₁`
4. **Labels matter**: same labels annihilate, different multiply branches
5. **Constructor syntax**: always `#Name{...}`, even nullary `#ZER{}`
6. **Pattern arms are lambdas**: `1n+: λpred. body` "binds" the field
7. **No inline match application**: avoid writting `λ{...}(x)`

REDUCTION MODES

- **WNF**: weak normal form (head only)
- **SNF**: strong normal form (full, keeps SUPs/DUPs)
- **CNF**: collapsed normal form (enumerate as pure lambda terms)

Use `-C` flag for collapse mode, omit for normalize mode.

I.4 · forgecode_clone/hvm/docs/theory/interaction_calculus.md

Interaction Calculus

The Interaction Calculus is a model of computation that extends the Lambda Calculus with **duplications** and **superpositions**, two primitives that enable optimal lazy evaluation.

CONTENTS

- Optimal Lazy Evaluation
- Duplications
- Superpositions
- Duplicating Lambdas
- Dup/Sup Labels
- The Four Core Interactions
- Relation to Interaction Combinators
- Complete Examples

OPTIMAL LAZY EVALUATION

Programs can be evaluated **strictly** (compute everything immediately) or **lazily** (compute only when needed). Lazy evaluation avoids unnecessary work, but has a flaw: when a value is used twice, its computation might happen twice. Haskell solves this with *memoization* (thunks), caching results so repeated access doesn't recompute. This works for data, but breaks down inside lambdas.

The Interaction Calculus achieves **optimal sharing**: work is never duplicated, even inside lambdas. Two dual primitives make this possible:

- **Duplications**: allow a single value to exist in multiple locations
- **Superpositions**: allow multiple values to exist in a single location

These correspond to Lafont's "fan nodes" in Interaction Combinators (1997), with opposite polarities. Together, they provide the mechanics for lazy cloning that extends into lambda bodies.

DUPLICATIONS

Duplications allow a **single value** to exist in **two locations**. The `! x &= v; t` construct duplicates `v`, making it available as `x₀` and `x₁` in `t`:

```
! x &= 2;
(x₀ + x₁)
--------- DUP-NUM
(2 + 2)
--------- OP2-NUM
4
```

Here, the number `2` was duplicated, then added to itself.

Duplication is incremental - it happens layer by layer, on demand:

```
! x &= [1, 2, 3];
(x₀, x₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [2, 3];
(1 <> t₀, 1 <> t₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [3];
(1 <> 2 <> t₀, 1 <> 2 <> t₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [];
(1 <> 2 <> 3 <> t₀, 1 <> 2 <> 3 <> t₁)
------------------------------------ DUP-CTR
([1, 2, 3], [1, 2, 3])
```

The list was cloned element by element. Each step peels off one layer - duplicating the head and creating a duplication for the tail. The tail duplication only triggers when both copies are accessed.

SUPERPOSITIONS

Superpositions allow **two values** to exist in **one location**. The `&{a, b}` construct creates a superposition of `a` and `b`:

```
(&{1, 2} + 10)
--------------------- OP2-SUP
! x &= 10;
&{(1 + x₀), (2 + x₁)}
--------------------- DUP-NUM
&{(1 + 10), (2 + 10)}
--------------------- OP2-NUM, OP2-NUM
&{11, 12}
```

Here, we added `10` to a superposition of `1` and `2`. The addition applied to both values, producing a superposition of results. Notice that `10` had to be duplicated; SUPs generate DUP nodes (DP0/DP1) as byproducts, and vice-versa.

Superpositions and duplications are duals. When a DUP meets a SUP, they annihilate, extracting the two values:

```
! x &= &{1, 2};
(x₀ + x₁)
--------------- DUP-SUP
(1 + 2)
--------------- OP2-NUM
3
```

This is like a pair projection: `x₀` gets the first element, `x₁` gets the second. SUPs and DUP nodes create and eliminate each other, just like LAMs and APPs.

DUPLICATING LAMBDAS

Lambdas are also duplicated incrementally. When that happens, the bound variable becomes superposed, and temporarily escapes its scope:

```
! f &= λx.(x + 1);
(f₀(10), f₁(20))
---------------------------- DUP-LAM
! b &= (&{$x0, $x1} + 1);
((λ$x0.b₀)(10), (λ$x1.b₁)(20))
---------------------------- APP-LAM, APP-LAM
! b &= (&{10, 20} + 1);
(b₀, b₁)
---------------------------- OP2-SUP, DUP-NUM
! b &= &{(10 + 1), (20 + 1)};
(b₀, b₁)
---------------------------- OP2-NUM, OP2-NUM
! b &= &{11, 21};
(b₀, b₁)
---------------------------- DUP-SUP
(11, 21)
```

Notice how, on the first step, the variables `$x0` and `$x1` are bound *outside* the lambda's body. This is why the Interaction Calculus needs globally scoped variables. It's also what enables **optimal sharing inside lambdas**: the body is now shared by `b`, so any computation there only happens once.

DUP/SUP LABELS

Consider this example:

```
(&A{1, 2} + &A{10, 20})
---------------------- OP2-SUP
! x &A= &A{10, 20};
&A{(1 + x₀), (2 + x₁)}
---------------------- DUP-SUP (same label: annihilate)
&A{(1 + 10), (2 + 20)}
---------------------- OP2-NUM, OP2-NUM
&A{11, 22}
```

Here, the superpositions *annihilated*: the first element paired with the first, and the second with the second. But what if we wanted all combinations instead?

Labels control this behavior. When a DUP meets a SUP with the *same* label, they annihilate. With *different* labels, they commute:

```
(&A{1, 2} + &B{10, 20})
------------------------------------------------ OP2-SUP
! x &A= &B{10, 20};
&A{(1 + x₀), (2 + x₁)}
------------------------------------------------ DUP-SUP (different labels: commute)
! a &A= 10;
! b &A= 20;
&A{(1 + &B{a₀, b₀}), (2 + &B{a₁, b₁})}
------------------------------------------------ DUP-NUM, DUP-NUM
&A{(1 + &B{10, 20}), (2 + &B{10, 20})}
------------------------------------------------ OP2-SUP, OP2-SUP
&A{&B{(1 + 10), (1 + 20)}, &B{(2 + 10), (2 + 20)}}
------------------------------------------------ OP2-NUM (x4)
&A{&B{11, 21}, &B{12, 22}}
```

Now we get a nested superposition containing all four results. Labels let us control whether superpositions collapse together or stay independent.

THE FOUR CORE INTERACTIONS

The minimal Interaction Calculus has just four rules. Two create computation (APP-LAM, DUP-SUP) and two propagate it (APP-SUP, DUP-LAM):

APP-LAM - Application eliminates lambda:

```
(λx.body)(arg)
-------------- APP-LAM
x ← arg
body
```

DUP-SUP - Duplication eliminates superposition (same label):

```
! x &L= &L{a, b}; t
------------------- DUP-SUP
x₀ ← a
x₁ ← b
t
```

APP-SUP - Application propagates through superposition:

```
(&L{a, b})(c)
------------------- APP-SUP
! x &L= c;
&L{a(x₀), b(x₁)}
```

DUP-LAM - Duplication propagates through lambda:

```
! f &L= λx.body; t
------------------ DUP-LAM
f₀ ← λ$X0.B₀
f₁ ← λ$X1.B₁
x  ← &L{$X0, $X1}
! B &L= body;
t
```

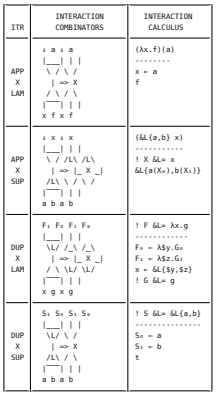
When labels differ, DUP-SUP commutes instead of annihilating:

```
! x &L= &R{a, b}; t
-------------------- DUP-SUP (L ≠ R)
! A &L= a;
! B &L= b;
x₀ ← &R{A₀, B₀}
x₁ ← &R{A₁, B₁}
t
```

These four rules form a complete system. Every other interaction in HVM is an extension for practical constructs: numbers, constructors, pattern-matching.

RELATION TO INTERACTION COMBINATORS

The Interaction Calculus is similar to Interaction Combinators, a parallel model of computation described by Lafont (1997). This similarity can be visualized as:



It can also be seen as a completion of the λ-Calculus, giving a computational meaning to previously undefined expressions: applying a pair and projecting a lambda. The Interaction Calculus provides sensible reduction rules for these, inspired by its Interaction Combinator equivalence: applying a pair (superposition) distributes over both elements (APP-SUP), and duplicating a lambda creates two lambdas with a superposed bound variable (DUP-LAM). This makes every possible interaction well-defined.

COMPLETE EXAMPLES

Here's an example demonstrating optimal sharing. We duplicate a lambda containing `(2 + 2)` and apply the copies to different arguments:

```
! F &= (λx. λy. ! z &= x; ((z₀ + z₁), y) 2);
(F₀(10), F₁(20))
---------------------------------------------- APP-LAM
! z &= 2;
! F &= λy.((z₀ + z₁), y);
(F₀(10), F₁(20))
---------------------------------------------- DUP-NUM
! F &= λy.((2 + 2), y);
(F₀(10), F₁(20))
---------------------------------------------- DUP-LAM
! B &= ((2 + 2), &{y0, y1});
((λy0.B₀ 10), (λy1.B₁ 20))
---------------------------------------------- APP-LAM (x2)
! B &= ((2 + 2), &{10, 20});
(B₀, B₁)
---------------------------------------------- DUP-CTR
! H &= (2 + 2);
! T &= &{10, 20};
((H₀, T₀), (H₁, T₁))
---------------------------------------------- OP2-NUM (shared!)
! H &= 4;
! T &= &{10, 20};
((H₀, T₀), (H₁, T₁))
---------------------------------------------- DUP-NUM, DUP-SUP
((4, 10), (4, 20))
```

Notice that `(2 + 2)` was computed only **once**, even though the lambda was duplicated and each copy was applied to a different argument. The result `4` flowed to both copies through the DUP-NUM interaction.

CHURCH 2^2

Here's a more complex example - computing 2^2 using Church numerals:

```
(λf. ! F &L= f; λx.(F₀ (F₁ x)) λg. ! G &K= g; λy.(G₀ (G₁ y)))
---------------------------------------------------------- APP-LAM
! F &L= λg. ! G &K= g; λy.(G₀ (G₁ y));
λx.(F₀ (F₁ x))
---------------------------------------------------------- DUP-LAM
! G &K= &L{g0, g1};
! F &L= λy.(G₀ (G₁ y));
λx.((λg0.F₀) (λg1.F₁ x))
---------------------------------------------------------- APP-LAM
! G &K= &L{(λg1.F₁ x), g1};
! F &L= λy.(G₀ (G₁ y));
λx.F₀
---------------------------------------------------------- DUP-LAM
! G &K= &L{(λg1.λy1.F₁ x), g1};
! F &L= (G₀ (G₁ &L{y0, y1}));
λx.λy0.F₀
---------------------------------------------------------- DUP-SUP (L ≠ K)
! A &K= (λg1.λy1.F₁ x);
! B &K= g1;
! F &L= (&L{A₀, B₀} (&L{A₁, B₁} &L{y0, y1}));
λx.λy0.F₀
---------------------------------------------------------- APP-SUP
! A &K= (λg1.λy1.F₁ x);
! B &K= g1;
! U &L= (&L{A₁, B₁} &L{y0, y1});
! F &L= &L{(A₀ U₀), (B₀ U₁)};
λx.λy0.F₀
---------------------------------------------------------- DUP-SUP (L = L)
! A &K= (λg1.λy1.(B₀ U₁) x);
! B &K= g1;
! U &L= (&L{A₁, B₁} &L{y0, y1});
λx.λy0.(A₀ U₀)
---------------------------------------------------------- APP-LAM
! A &K= λy1.(B₀ U₁);
! B &K= x;
! U &L= (&L{A₁, B₁} &L{y0, y1});
λx.λy0.(A₀ U₀)
---------------------------------------------------------- DUP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= (&L{λy11.A₁, B₁} &L{y0, &K{y10, y11}});
λx.λy0.((λy10.A₀) U₀)
---------------------------------------------------------- APP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= (&L{λy11.A₁, B₁} &L{y0, &K{U₀, y11}});
λx.λy0.A₀
---------------------------------------------------------- APP-SUP
! A &K= (B₀ U₁);
! B &K= x;
! V &L= &L{y0, &K{U₀, y11}};
! U &L= &L{((λy11.A₁) V₀), (B₁ V₁)};
λx.λy0.A₀
---------------------------------------------------------- DUP-SUP (L = L), APP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= &L{(A₁ y0), (B₁ &K{U₀, y11})};
λx.λy0.A₀
---------------------------------------------------------- DUP-SUP (L = L)
! A &K= (B₀ (B₁ &K{(A₁ y0), y11}));
! B &K= x;
λx.λy0.A₀
```

At this point, the computation is done. The result is a lambda `λx.λy0.A₀` connected to a small network of nodes. This compressed form represents the answer but is hard to read directly. To see the familiar Church numeral 4, we can **collapse** the Interaction Calculus term back into a proper λ-Term, by applying 2 extra interactions, DUP-VAR and DUP-APP:

```
...
-------------------------------- DUP-VAR
! A &K= (x (x &K{(A₁ y0), y11}));
λx.λy0.A₀
-------------------------------- DUP-APP, DUP-VAR
! X &K= (x &K{((x X₁) y0), y11});
λx.λy0.(x X₀)
-------------------------------- DUP-APP, DUP-VAR
! Y &K= &K{((x (x Y₁)) y0), y11};
λx.λy0.(x (x Y₀))
-------------------------------- DUP-SUP (K = K)
λx.λy0.(x (x ((x (x y11)) y0)))
-------------------------------- APP-LAM (x2)
λx.λy0.(x (x (x (x y0))))
```

The Church numeral 2 (`λf.λx.(f (f x))`) was applied to itself, yielding 4 (`λf.λx.(f (f (f (f x))))`). Despite temporarily escaping variables, the system correctly computes the result. This is the kind of symbolic computation where sharing matters most, and is often used as a benchmark in the literature. HVM completes it in 14 interactions, which is optimal.

I.5 · `forgecode_clone/hvm/docs/hvm/core.md`

HVM Core Terms

This document defines the core surface syntax for HVM terms. These terms are parsed into static (book) terms and later instantiated as dynamic terms during execution. See `docs/hvm/memory.md` for the dynamic/static memory layout.

GRAMMAR

```
Term ::=
  | Var  Name                                    -- variable
  | Dp0  Name "₀"                                -- first dup variable
  | Dp1  Name "₁"                                -- second dup variable
  | Ref  "@" Name                                -- reference
  | Pri  "%" Name                                -- primitive (native) function
  | Nam  "^" Name                                -- name (stuck head)
  | Dry  "^" "(" Term " " Term ")"               -- dry (stuck application)
  | Era  "&{}"                                   -- erasure
  | Sup  "&" Label "{" Term "," Term "}"         -- superposition
  | Dup  "!" Name "&" Label "=" Term ";" Term    -- duplication term
  | Ctr  "#" Name "{" Term,* "}"                 -- constructor
  | Mat  "λ" "{" "#" Name ":" Term ";" Term "}"  -- pattern match
  | Swi  "λ" "{" Num ":" Term ";" Term "}"       -- number switch
  | Use  "λ" "{" Term "}"                        -- use (unbox)
  | Lam  "λ" Name "." Term                       -- lambda
  | App  "(" Term " " Term ")"                   -- application
  | Num  integer                                 -- number literal
  | Op2  "(" Term Oper Term ")"                  -- binary operation
  | Eql  "(" Term "==" Term ")"                  -- equality test
  | And  "(" Term ".&." Term ")"                 -- short-circuit AND
  | Or   "(" Term ".|." Term ")"                 -- short-circuit OR
  | DSu  "&" "(" Term ")" "{" Term "," Term "}"  -- dynamic superposition
  | DDu  "!" Name "&" "(" Term ")" "=" Term ";" Term -- dynamic duplication term
  | Inc  "↑" Term                                -- priority wrapper
  | Alo  "@" "{" Name,* "}" Term                 -- allocation
  | Uns  "!" "$" "{" Name "," Name "}" ";" Term  -- unscoped binding

Name  ::= [_A-Za-z0-9]+
Label ::= Name
Oper  ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||"
        | "^" | "~" | "<<" | ">>" | "==" | "!="
        | "<" | "<=" | ">" | ">="
```

NOTES

- Variables are affine: each variable is used at most once.
- Variables are global: a variable can occur outside its binder's lexical scope.
- Labels determine how duplications and superpositions interact; equal labels annihilate, different labels commute.
- Primitives (`%name`) are native functions and must be fully applied with the correct arity; `%log` prints a string and yields `#Nil` .

- Surface sugar accepts `λ$x. body` as an unscoped lambda, equivalent to `! f = λ x ; f(body)` with fresh `f` (see `docs/hvm/syntax.md`).

I.6 · forgecode_clone/hvm/docs/hvm/memory.md

HVM Memory Layout

HVM represents every term as a 64-bit word. The same layout is used for dynamic terms (mutable heap entries) and static terms (immutable book entries).

TERM POINTER LAYOUT (64-BIT)

```
SUB (1 bit) | TAG (7 bits) | EXT (16 bits) | VAL (40 bits)
```

- `SUB` : marks a substitution cell; ignored for immediates.
- `TAG` : constructor variant (APP, LAM, SUP, etc.).
- `EXT` : metadata (dup label, ctor/ref name, level, op code, flags).
- `VAL` : payload (heap loc, immediate, or 0).

TERM MEMORY TABLE

TERM	TAG	EXT	VAL	NOTES
application	APP	0	node: [func, arg]	dynamic or static
linked variable	VAR	0	lam body or subst	dynamic only; follows SUB cells
linked duplicate 0	DP0	label	dup expr or subst	dynamic only; twin of DP1
linked duplicate 1	DP1	label	dup expr or subst	dynamic only; twin of DP0
linked lambda	LAM	level+flags	node: [body]	dynamic only; binder for VAR
quoted lam	LAM	level+flags	node: [body]	quoted; level stored in EXT
superposition	SUP	label	node: [tm0, tm1]	dynamic or static
linked duplication term	DUP	label	node: [expr, body]	dynamic or static; binder for DP0/DP1
quoted duplication term	DUP	label	node: [expr, body]	expr typically `&L{BJ0,BJ1}` in quoted mode
number literal	NUM	0	unboxed u32	
constructor arity 0	C00	ctor name	0	tag encodes arity
constructor arity N	C01-16	ctor name	node: [field0..fieldN-1]	N = tag - C00
pattern match	MAT	ctor name	node: [hit, miss]	match on constructor
number switch	SWI	number literal	node: [zero, succ]	parser/printer distinction from MAT
use (unbox)	USE	0	node: [fun]	forces argument, then applies
binary op	OP2	op code	node: [lhs, rhs]	EXT stores OP_ADD, OP_MUL, etc.
equality	EQL	0	node: [lhs, rhs]	structural equality
logical AND	AND	0	node: [lhs, rhs]	short-circuit AND

TERM	TAG	EXT	VAL	NOTES
logical OR	OR	0	node: [lhs, rhs]	short-circuit OR
priority wrapper	INC	0	node: [term]	collapse priority wrapper
name literal	NAM	name id	0	literal ^name
stuck application	DRY	0	node: [fun, arg]	literal ^(f x)
reference	REF	name id	0	book reference @name
primitive	PRI	name id	node: [arg0..argN-1]	native function call; arity from prim table
allocation	ALO	bind list length	direct or packed pair	len=0: VAL=book term; len>0: VAL->(low24=book term, high40=bind list head)
unscoped binding	UNS	0	node: [body]	helper to construct unscoped lams
wildcard	ANY	0	0	duplicates itself, equals anything
quoted lam var	BJV	0	de Bruijn level	quoted lam-bound var
quoted dup var 0	BJ0	label	de Bruijn level	quoted dup-bound var
quoted dup var 1	BJ1	label	de Bruijn level	quoted dup-bound var
dynamic superposition	DSU	0	node: [lab, tm0, tm1]	label computed dynamically
dynamic duplication	DDU	0	node: [lab, val, body]	label computed dynamically

SUBSTITUTION CELLS (SUB BIT)

When a linked binder interacts (APP-LAM, DUP-*), its body or expr slot is replaced by a substitution term with the SUB bit set. Any VAR/DP0/DP1 that points to that heap location must read the substitution instead of the original binder. This is how linked variables resolve their binders without extra maps.

DUP NODES VS DUP TERMS

A DUP term is the syntactic binder `[expr, body]` with tag `DUP`. A DUP node is the shared expr slot that DP0/DP1 point to; it has no body and no parent, and it is substituted when a duplication interaction occurs.

LINKED VS QUOTED BINDERS

Linked binders are dynamic lams/dups whose vars point to heap locations. Linked LAM uses `EXT = level | flags` and its VARs point to the body slot. Linked DUP uses DP0/DP1 vars that point to the shared expr slot.

Quoted binders are terms encoded with de Bruijn levels in `EXT` (and for BJ0/BJ1, the dup label). Their variables are BJV/BJ0/BJ1 indices, so there are no heap links and no interaction with APP-LAM or DUP-SUP.

During collapse, linked LAM/DUP binders are converted into quoted LAM/DUP binders, and free VAR/DP0/DP1 become BJV/BJ0/BJ1 at the current level. This is used to turn interaction nets into full lambda terms.

DYNAMIC VS STATIC TERMS

Dynamic terms live on the heap and are mutable: binders can be replaced by substitutions, and DUP nodes evolve as duplication is forced. Static terms live in the book and are immutable; they store de Bruijn levels and are never updated in place.

Allocation (ALO) terms bridge the two: they reference a static book term and a bind list, lazily expanding one layer into a dynamic term when forced. This keeps static definitions compact while still allowing dynamic sharing during execution.

ALO RUNTIME ENCODINGS

- `ALO` with `len == 0` (empty substitution list) is stored directly:
  - `ALO.val = book_term_loc`
  - no extra heap allocation for an ALO pair node.
- `ALO` with `len > 0` stores one packed ALO pair word at `ALO.val`:
  - `high 40 bits = bind list head location`
  - `low 24 bits = book term location` (truncated static location)

This keeps `len > 0` ALO pairs at one heap word while preserving full 40-bit dynamic locations for bind-list heads.

STATIC BOOK LOCATION BOUND

Because packed ALO pairs store book term locations in 24 bits, static/book allocation must fit in that range. After parsing, the runtime checks this bound and reports an error if static locations exceed `2^24` words.

ALO BIND-LIST NODES

For `len > 0`, the bind list is a linked list of 2-slot nodes:

- `node[0]`: bound term cell (the substitution target term lives here).
- `node[1]`: `NUM(next_node_loc)`; `0` means end of list.

This means the pointer used by variables/copiers is the bind-node location itself (slot `0`), not a separate allocation.

- In `ALO-LAM`, `node[0]` is the lambda body slot.
- In `ALO-DUP`, `node[0]` is the shared DUP expr slot.

LAM EXT FLAGS

- `LAM_ERA_MASK` (0x8000): binder is unused in lambda body (erasing lambda).

I.7 · forgecode_clone/hvm/docs/hvm/syntax.md

HVM Parser Syntax

Desugarings use the core surface syntax from `docs/hvm/core.md`. Core application is written there as `(f x)`, so this doc uses that notation for desugaring only; the parser itself does not accept whitespace application.

PARSING MODEL (HOW TERMS GROUP)

- Base unit: an atom.
- Postfix call: `f(...)` binds tightest.
- `<>` binds tighter than infix operators.
- `===`, `.&.`, `.|.` bind tighter than infix operators (same precedence class as `==`, `&&`, `||`, but separate nodes).
- Infix operators are left-associative, by the precedence table below.
- No whitespace application: `f x` is not an application; use `f(x)`.

LEXICAL RULES

- Whitespace: space, tab, newline, `\r`.
- Line comments: `// ...`.
- Names: `_a-zA-Z0-9$`.
  - A term starting with a digit is parsed as a number literal, not a name.

ATOMS

- Variable: `x`
- Dup variables: `x₀` or `x₁` (see “Dup variables”)
- Grouping: `(term)`
- Constructor: `#Name` or `#Name{a,b,c}` (commas optional, trailing comma ok)
- Reference: `@Name`

- Primitive: `%Name`
- Name head: `^Name`
- Dry app: `^(f x)`
- Priority wrapper: `↑atom`
- Wildcard: `*`
- Lambda / dup / sup / match / literals (see below)

APPLICATION

- `f(a, b, c)` desugars to `((f a) b) c`.
  - Commas are optional: `f(a b c)` is the same.
  - `f()` is allowed and desugars to `f`.

LITERALS

NUMBERS

- `123` is a numeric literal.

NATURALS (PEANO)

Using built-in constructors `#Zer` and `#Suc`:

- `Nn` desugars to `#Suc{...#Suc{#Zer}...}` (N times)
- `Nn+term` desugars to `#Suc{...#Suc{term}...}` (N times)
  - `n+` must be contiguous; `3n + x` is not a natural literal.

Examples:

```
3n    => #Suc{#Suc{#Suc{#Zer}}}
2n+xs => #Suc{#Suc{xs}}
```

CHARACTERS

- `'c'` desugars to `#Chr{#<codepoint>}`.

Escapes:

- Standard escapes: `\n`, `\t`, `\r`, `\0`, `\\`, `\'`, `\"`.
- Other backslashes escape the next byte verbatim.
- Otherwise UTF-8 sequences are decoded into a numeric codepoint.

STRINGS

Strings are lists of `#Chr`:

- `"str"` desugars to `#Con{#Chr{#s}, #Con{#Chr{#t}, #Con{#Chr{#r}, #Nil}}}`.

Same escape rules as characters.

LISTS

Lists use built-in `#Nil` and `#Con`:

- `[]` desugars to `#Nil`.
- `[a, b, c]` desugars to `#Con{a, #Con{b, #Con{c, #Nil}}}`.
  - Commas are optional; trailing comma allowed.
- `a <> b` desugars to `#Con{a, b}` (cons sugar).

LAMBDAS AND BINDERS

LAMBDA

- `λx. body` is core `Lam`.

- `λx,y,z. body` desugars to `λx. λy. λz. body`.
- `λ$x. body` is an unscoped lambda binder (see "Unscoped lambda (UNS)").

CLONED BINDERS (AUTO-DUP)

Prefix `&` allows multiple uses; the parser inserts dup nodes.

- `λ&x. body` desugars to `λx. body` plus auto-duplication of `x`.
- Auto-duplication expands a cloned variable with N uses into N-1 nested dups using fresh labels. Example for three uses: `body[x,x,x]` becomes `!d0&=x; !d1&=d0₁; body[d0₀,d1₀,d1₁]`.

INLINE DUPLICATION ON LAMBDAS

Short form for inserting a dup on the bound variable:

- `λx&L. body` desugars to `λx. !x&L = x; body`.
- `λx&. body` uses a fresh label.
- `λx&(lab). body` desugars to `λx. !x&(lab) = x; body`.

All of these combine with cloning: `λ&x&L. ...`.

DUPLICATION AND LET (`!`)

LET SUGAR

- `!x = v; body` desugars to `((λx. body) v)`.
- `!!x = v; body` desugars to `((λ{λx. body}) v)` (strict let).
- Cloned forms `!&x = ...` and `!!&x = ...` allow multiple uses of `x` and insert auto-dup as needed.

DUPLICATION (STATIC LABEL)

- `!x&L = v; body` is core `Dup`.
- `!x& = v; body` uses a fresh label.
- Optional `;` after `v` is allowed.
- Cloned dup binder: `!&x&L = v; body` (or `!&x& = ...`).

DUPLICATION (DYNAMIC LABEL)

- `!x&(lab) = v; body` is core `DDu`.
- `!&x&(lab) = v; body` is the cloned form.

UNSCOPED LAMBDA (UNS)

- `! f = λ x ; body` desugars to `!${f,x}; body`.
- `λ$x. body` desugars to `! f = λ x ; f(body)` with fresh `f`.
  - Commas are allowed: `λ$x,y. body` (x unscoped, y scoped).
  - The unscoped variable is referenced as plain `x` inside the body.

SUPERPOSITION (`&`) AND FORK

SUPERPOSITION AND ERASURE

- `&L{a, b}` is core `Sup`.
- `&(lab){a, b}` is core `DSu`.
- `&{}` is core `Era`.
  - Commas are optional; trailing comma allowed.

FORK SUGAR

```
&Lλx,y{A;B}
```

desugars to:

```
λx&L. λy&L. &L{A', B'}
```

where `A'` uses x_0, y_0 and `B'` uses x_1, y_1.

Inside each branch you can omit $_0$ / $_1$ and the parser chooses the side automatically. You can also prefix branches with $\&_0$: or $\&_1$: to force the side for all un-subscripted dup variables in that branch. Semicolons between branches are optional; trailing `;` allowed.

Dynamic labels are supported: `&(lab)λx,y{A;B}`.

MATCH / SWITCH / USE (`λ{...}`)

ERASURE

- `λ{}` desugars to `&{}`.

USE / UNBOX

- `λ{term}` is core `Use`.
- `λ{_: term}` also desugars to `λ{term}` if it is the only entry.

PATTERN MATCH AND NUMERIC SWITCH

Syntax:

```
λ{
  pat1: t1;
  pat2: t2;
  ...
  _ : default
}
```

Patterns:

- `#Name:` match constructor `#Name`
- `[]:` match `#Nil`
- `<>:` match `#Con`
- `0n:` match `#Zer`
- `1n+:` match `#Suc` (digits before `n+` are ignored)
- `N:` numeric switch on literal `N`
- `_:` default branch

Desugaring is right-nested:

```
λ{#A: t1; #B: t2; d}
=> λ{#A: t1; λ{#B: t2; d}}

λ{0: t0; 1: t1; d}
=> λ{0: t0; λ{1: t1; d}}
```

Semicolons between cases are optional; trailing `;` allowed. The final default can be written as `_ : d` or as a bare `d`.

NAMES AND REFERENCES

- `@Name` is a book reference.
- `%Name` is a primitive (native) function; it must be fully applied as `%Name(arg1, arg2, ...)` with the correct arity (bare `%Name` is a parse error).
- `^Name` is a stuck name head.
- `^(f x)` is a dry application.

PRIMITIVES

- `%log` prints a string (list of `#Chr`) to stdout and returns `#Nil`.

PRIORITY WRAPPER AND WILDCARD

- `↑atom` is core `Inc` (binds to a single atom).
- `*` is core `Any`.

INFIX OPERATORS

All infix operators are left-associative.

Precedence (higher binds tighter):

- 8: `^`
- 7: `* / %`
- 6: `+ -`
- 5: `<< >>`
- 4: `< <= > >=`
- 3: `== !=`
- 2: `&&`
- 1: `||`

Notes:

- `~` parses as `OP_NOT` but is infix. Evaluation ignores the left operand and uses only the right one (`~b`). For unary not, write `0 ~ x`.

STRUCTURAL EQUALITY AND SHORT-CIRCUIT OPS

These are separate nodes (not `Op2`):

- `a === b` structural equality
- `a .&. b` short-circuit AND
- `a .|. b` short-circuit OR

DUP VARIABLES AND SUBSCRIPTS

- Variables bound by `Dup` (or inline `λx&L`) must be referenced as x_0 or x_1.
- Inside fork branches, the parser can auto-select the side if you omit the subscript.
- Dynamic dup variables (`!x&(lab)=...`) always require a subscript.

NOT PARSED DIRECTLY

Some internal terms have print forms but are not parsed directly (for example `@{...}` for `ALO`). The list above reflects what the parser accepts today.

I.8 · forgecode_clone/hvm/docs/hvm/collapser.md

Collapser (CNF Readback)

Interaction Calculus is affine and uses explicit duplication (DUP) and superposition (SUP). That is great for optimal sharing but hard for humans to read. The collapser is the readback procedure: it turns one IC term into a stream of ordinary lambda terms, i.e., collapsed normal form (CNF).

Two insights make this possible:

- Quoting removes DUPs. When a branch is ready to print, we run `cnf(term)`. Linked vars become quoted vars (BJV/BJ0/BJ1). DUP interactions are defined on quoted dup vars, so DUP nodes are cloned away and disappear from the printed term.
- Lifting removes SUPs. We lift the first SUP to the top and enumerate its branches. Same-label SUPs annihilate pairwise; different-label SUPs commute and create a cross product of branches.

ALGORITHM (AS IMPLEMENTED)

- `cnf` (clang/cnf/_.c): reduce to WNF, then lift the first SUP to the top and return immediately. ERA propagates upward; INC is left in place for the flattener. When collapse threads are idle, cnf can spawn subterm tasks to use the same worker pool.

- `eval_collapse` (clang/eval/collapse.c): breadth-first traversal with a work-stealing key queue. Lower numeric keys are popped first; SUP increases key, INC decreases key. Single-threaded runs pop FIFO within each key bucket for deterministic ordering. When a branch has no SUP, it prints `cnf(term)`.

LABEL BEHAVIOR (PAIRWISE VS CROSS PRODUCT)

Different labels commute (cross product):

```
[&A{1,2}, &B{3,4}]
```

```
[1,3]
[1,4]
[2,3]
[2,4]
```

Same labels annihilate pairwise:

```
[&A{1,2}, &A{3,4}]
```

```
[1,2]
[3,4]
```

WHERE TO LOOK

- `clang/cnf/_.c` : SUP lifting rules.
- `clang/eval/collapse.c` : branch enumeration + SNF quoting for output.
- `clang/data/wspq.c` : work-stealing key queue used by collapse (FIFO when T=1).

I.9 · forgecode_clone/hvm/docs/hvm/ffi.md

C FFI

This document describes the C FFI for HVM: how it is implemented in the runtime and how to use it from external C code.

Overview

- FFI libraries are loaded before parsing, so primitives can be registered in time for arity checks.
- Each library exposes `hvm_ffi_init` and receives an API table with all safe entry points.
- Registered primitives are callable with the `%name(args...)` syntax.
- Re-registering an existing primitive name overrides it and prints a warning.

FFI Header

- Include the public header at `clang/hvm_ffi.h`.
- This header defines `HvmApi`, `Term`, tags, and `ABI_VERSION`.

Loading Model

- Load a shared library with `--ffi <path>`.
- Load all shared libraries in a directory with `--ffi-dir <path>`.
- Libraries must export:

```
void hvm_ffi_init(const HvmApi *api);
```

Implementation Notes

- Dynamic loading is done with `dlopen` / `dlsym` in `clang/ffi/load.c` and `clang/ffi/load_dir.c`.
- The API table is built in `clang/ffi/api.c` and passed to `hvm_ffi_init`.

- Primitive registration uses `prim_register` in `clang/prim/register.c`, which stores a function pointer and arity by name ID.
- The parser uses `prim_arity` to validate `%name(...)` applications before evaluation.
- The evaluator calls primitives via `wnf/pri.c`.

Minimal Example

```
#include "clang/hvm_ffi.h"

static const HvmApi *api;

static Term prim_c_add(Term *args) {
  Term a = api->wnf(args[0]);
  Term b = api->wnf(args[1]);
  if (api->term_tag(a) != NUM || api->term_tag(b) != NUM) {
    api->runtime_error("%c_add expected numbers");
  }
  return api->term_new_num(api->term_val(a) + api->term_val(b));
}

void hvm_ffi_init(const HvmApi *api_arg) {
  api = api_arg;
  if (api->abi_version != ABI_VERSION) {
    api->runtime_error("FFI ABI mismatch");
  }
  api->register_prim("c_add", 5, 2, prim_c_add);
}
```

Build the FFI Library

- macOS:

```
clang -dynamiclib -fPIC -I /path/to/hvm \
  -o /path/to/libffi_example.dylib \
  /path/to/ffi_example.c
```

- Linux:

```
clang -shared -fPIC -I /path/to/hvm \
  -o /path/to/libffi_example.so \
  /path/to/ffi_example.c
```

Use It

```
./clang/main program.hvm --ffi /path/to/libffi_example.dylib
./clang/main program.hvm --ffi-dir /path/to/ffi_dir
```

Strings and Constructors

- HVM strings are lists using `#CON`, `#CHR`, and `#NIL` (uppercase).
- `term_new_ctr` takes a symbol-table constructor ID.
- Use `name_from_str("CON", 3)` and friends to intern/get the correct constructor IDs.

Threading and Safety

- Primitives may run concurrently in `-T<N>` mode.
- Calling `wnf`, `heap_alloc`, or `heap_read/set` inside a primitive is safe because primitives run on runtime worker threads with proper thread IDs.
- Do not call HVM API functions from threads you create yourself.
- `table_find` mutates global state, so only call it during `hvm_ffi_init`.
- Any external side effects (I/O, logging, global state) must be synchronized by the FFI if you need deterministic ordering.

Limitations

- GPU execution does not support FFI (runtime aborts).

I.10 · forgecode_clone/hvm/docs/hvm/interactions/alo_cop.md

ALO-COP

```
@{s} n₀
------- ALO-DP0
s[n]₀ or n₀ when substitution missing (n is a de Bruijn level)

@{s} n₁
------- ALO-DP1
s[n]₁ or n₁ when substitution missing (n is a de Bruijn level)
```

I.11 · forgecode_clone/hvm/docs/hvm/interactions/alo_dry.md

ALO-DRY

```
@{s} ^(f x)
-------------- ALO-DRY
^(@{s}f @{s}x)
```

I.12 · forgecode_clone/hvm/docs/hvm/interactions/alo_dup.md

ALO-DUP

```
@{s} ! x &L = v; t
------------------ ALO-DUP
x' ← fresh
! x' &L = @{s} v
@{x',s} t
```

I.13 · forgecode_clone/hvm/docs/hvm/interactions/alo_lam.md

ALO-LAM

```
@{s} λx.f
------------ ALO-LAM
x' ← fresh
λx'.@{x',s}f
```

I.14 · forgecode_clone/hvm/docs/hvm/interactions/alo_nam.md

ALO-NAM

```
@{s} ^n
------- ALO-NAM
^n
```

I.15 · forgecode_clone/hvm/docs/hvm/interactions/alo_nod.md

ALO-NOD

```
@{s} T{a,b,...}
---------------- ALO-NOD
T{@{s}a, @{s}b, ...}
```

I.16 · forgecode_clone/hvm/docs/hvm/interactions/alo_var.md

ALO-VAR

```
@{s} n
------ ALO-VAR
s[n] or n when substitution missing (n is a de Bruijn level)
```

I.17 · forgecode_clone/hvm/docs/hvm/interactions/and_era.md

AND-ERA

```
(&{} .&. b)
----------- AND-ERA
&{}
```

I.18 · forgecode_clone/hvm/docs/hvm/interactions/and_inc.md

AND-INC

```
(↑a & b)
--------- AND-INC
↑(a & b)
```

I.19 · forgecode_clone/hvm/docs/hvm/interactions/and_num.md

AND-NUM

```
(#0 .&. b)
---------- AND-ZER
#0

(#n .&. b)   [n ≠ 0]
------------------- AND-ONE
b
```

I.20 · forgecode_clone/hvm/docs/hvm/interactions/and_sup.md

AND-SUP

```
(&L{a0,a1} .&. b)
------------------------- AND-SUP
! B &L = b
&L{(a0 .&. B₀), (a1 .&. B₁)}
```

I.21 · forgecode_clone/hvm/docs/hvm/interactions/app_ctr.md

APP-CTR

```
(ctr a)
---------- APP-CTR (stuck)
^(ctr a)
```

I.22 · forgecode_clone/hvm/docs/hvm/interactions/app_dry.md

APP-DRY

```
(^(f x) a)
---------- APP-DRY
^(^(f x) a)
```

I.23 · forgecode_clone/hvm/docs/hvm/interactions/app_era.md

APP-ERA

```
(&{} a)
------- APP-ERA
&{}
```

I.24 · forgecode_clone/hvm/docs/hvm/interactions/app_inc.md

APP-INC

```
(↑f x)
-------- APP-INC
↑(f x)
```

I.25 · forgecode_clone/hvm/docs/hvm/interactions/app_lam.md

APP-LAM

```
(λx.f a)
-------- APP-LAM
x ← a
f
```

I.26 · forgecode_clone/hvm/docs/hvm/interactions/app_mat_ctr.md

APP-MAT-CTR

```
(λ{#K:h; m} #K{a,b})
-------------------- APP-MAT-CTR-MAT
(h a b)

(λ{#K:h; m} #L{a,b})
-------------------- APP-MAT-CTR-MIS
(m #L{a,b})
```

I.27 · forgecode_clone/hvm/docs/hvm/interactions/app_mat_num.md

APP-MAT-NUM

```
(λ{#a:h; m} #a)
--------------- APP-MAT-NUM-MAT
h

(λ{#a:h; m} #b)
--------------- APP-MAT-NUM-MIS
(m #b)
```

I.28 · forgecode_clone/hvm/docs/hvm/interactions/app_mat_sup.md

APP-MAT-SUP

```
(λ{#K:h; m} &L{a,b})
-------------------- APP-MAT-SUP
! H &L = h
! M &L = m
&L{(λ{#K:H₀; M₀} a)
  ,(λ{#K:H₁; M₁} b)}
```

I.29 · forgecode_clone/hvm/docs/hvm/interactions/app_nam.md

APP-NAM

```
(name a)
--------- APP-NAM
^(name a)
```

I.30 · forgecode_clone/hvm/docs/hvm/interactions/app_sup.md

APP-SUP

```
(&L{f,g} a)
----------------- APP-SUP
! A &L = a
&L{(f A₀),(g A₁)}
```

I.31 · forgecode_clone/hvm/docs/hvm/interactions/ddu_era.md

DDU-ERA

```
! X &(&{}) = v; b
----------------- DDU-ERA
&{}
```

I.32 · forgecode_clone/hvm/docs/hvm/interactions/ddu_inc.md

DDU-INC

```
! X &(↑x) = v; b
---------------- DDU-INC
↑(! X &(x) = v; b)
```

I.33 · forgecode_clone/hvm/docs/hvm/interactions/ddu_num.md

DDU-NUM

```
! X &(#n) = v; b
---------------- DDU-NUM
! X &n = v
b(X₀, X₁)
```

I.34 · forgecode_clone/hvm/docs/hvm/interactions/ddu_sup.md

DDU-SUP

```
! X &(&L{x,y}) = v; b
------------------------------ DDU-SUP
! V &L = v
! B &L = b
&L{! X &(x) = V₀; B₀, ! X &(y) = V₁; B₁}
```

I.35 · forgecode_clone/hvm/docs/hvm/interactions/dsu_era.md

DSU-ERA

```
&(&{}){a, b}
------------ DSU-ERA
&{}
```

I.36 · forgecode_clone/hvm/docs/hvm/interactions/dsu_inc.md

DSU-INC

```
&(↑x){a, b}
------------ DSU-INC
↑(&(x){a, b})
```

I.37 · forgecode_clone/hvm/docs/hvm/interactions/dsu_num.md

DSU-NUM

```
&(#n){a, b}
----------- DSU-NUM
&n{a, b}
```

I.38 · forgecode_clone/hvm/docs/hvm/interactions/dsu_sup.md

DSU-SUP

```
&(&L{x,y}){a, b}
------------------------- DSU-SUP
! A &L = a
! B &L = b
&L{&(x){A₀,B₀}, &(y){A₁,B₁}}
```

I.39 · forgecode_clone/hvm/docs/hvm/interactions/dup_dry.md

DUP-DRY

```
! X &L = ^(f x)
--------------- DUP-DRY
! F &L = f
! A &L = x
X₀ ← ^(F₀ A₀)
X₁ ← ^(F₁ A₁)
```

I.40 · forgecode_clone/hvm/docs/hvm/interactions/dup_lam.md

DUP-LAM

```
! F &L = λx.f
---------------- DUP-LAM
F₀ ← λ$x0.G₀
F₁ ← λ$x1.G₁
x  ← &L{$x0,$x1}
! G &L = f
```

I.41 · forgecode_clone/hvm/docs/hvm/interactions/dup_nam.md

DUP-NAM

```
! X &L = name
------------ DUP-NAM
X₀ ← name
X₁ ← name
```

I.42 · forgecode_clone/hvm/docs/hvm/interactions/dup_nod.md

DUP-NOD

```
! X &L = T{a,b,...}
------------------- DUP-NOD
! A &L = a
! B &L = b
...
X₀ ← T{A₀,B₀,...}
X₁ ← T{A₁,B₁,...}
```

I.43 · forgecode_clone/hvm/docs/hvm/interactions/dup_sup.md

DUP-SUP

```
! X &L = &R{a,b}
---------------- DUP-SUP
if L == R:
  X₀ ← a
  X₁ ← b
else:
  ! A &L = a
  ! B &L = b
  X₀ ← &R{A₀,B₀}
  X₁ ← &R{A₁,B₁}
```

I.44 · forgecode_clone/hvm/docs/hvm/interactions/eql_any.md

EQL-ANY

```
(* === b)
----------- EQL-ANY-L
1

(a === *)
----------- EQL-ANY-R
1
```

I.45 · forgecode_clone/hvm/docs/hvm/interactions/eql_ctr.md

EQL-CTR

```
(#K{a0,a1...} === #K{b0,b1...})  (same tag)
--------------------------------------- EQL-CTR-MAT
For SUC (1n+): ↑(pred === pred)
For CON (<>): ↑((head === head) & ↑(tail === tail))
Others: (a0 === b0) & (a1 === b1) & ...

(#K{...} === #L{...})  (different tag)
------------------------------------- EQL-CTR-MIS
#0
```

I.46 · forgecode_clone/hvm/docs/hvm/interactions/eql_dry.md

EQL-DRY

```
(^(af ax) === ^(bf bx))
---------------------- EQL-DRY
(af === bf) & (ax === bx)
```

I.47 · forgecode_clone/hvm/docs/hvm/interactions/eql_era.md

EQL-ERA

```
(&{} === b)
----------- EQL-ERA-L
&{}

(a === &{})
----------- EQL-ERA-R
&{}
```

I.48 · forgecode_clone/hvm/docs/hvm/interactions/eql_inc.md

EQL-INC

```
(↑a === b)
----------- EQL-INC-L
↑(a === b)

(a === ↑b)
----------- EQL-INC-R
↑(a === b)
```

I.49 · forgecode_clone/hvm/docs/hvm/interactions/eql_lam.md

EQL-LAM

```
 (λax.af === λbx.bf)
------------------- EQL-LAM
X := fresh_nam()
ax ← X
bx ← X
af === bf
```

I.50 · forgecode_clone/hvm/docs/hvm/interactions/eql_mat.md

EQL-MAT

```
 (λ{#K:ah;am} === λ{#K:bh;bm})  (same tag)
---------------------------------------- EQL-MAT-MAT
(ah === bh) & (am === bm)

(λ{#K:...} === λ{#L:...})  (different tag)
---------------------------------------- EQL-MAT-MIS
#0
```

I.51 · forgecode_clone/hvm/docs/hvm/interactions/eql_nam.md

EQL-NAM

```

```

I.52 · forgecode_clone/hvm/docs/hvm/interactions/eql_num.md

EQL-NUM

```
 (#a === #b)
------------ EQL-NUM
#(a == b)
```

I.53 · forgecode_clone/hvm/docs/hvm/interactions/eql_sup.md

EQL-SUP

```
 (&L{a0,a1} === b)
---------------------- EQL-SUP-L
! B &L = b
&L{(a0 === B₀), (a1 === B₁)}

(a === &L{b0,b1})
---------------------- EQL-SUP-R
! A &L = a
&L{(A₀ === b0), (A₁ === b1)}
```

I.54 · forgecode_clone/hvm/docs/hvm/interactions/eql_use.md

EQL-USE

```
(λ{af} === λ{bf})
----------------- EQL-USE
af === bf
```

I.55 · forgecode_clone/hvm/docs/hvm/interactions/mat_inc.md

MAT-INC

```
(λ{...} ↑x)
------------ MAT-INC
↑(λ{...} x)
```

I.56 · forgecode_clone/hvm/docs/hvm/interactions/op2_era.md

OP2-ERA

```
@@opr(&{}, y)
------------- OP2-ERA
&{}
```

I.57 · forgecode_clone/hvm/docs/hvm/interactions/op2_inc.md

OP2-INC

```
(↑x op y)
---------- OP2-INC-X
↑(x op y)

(#n op ↑y)
---------- OP2-INC-Y
↑(#n op y)
```

I.58 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_era.md

OP2-NUM-ERA

```
(x op &{}) where x is NUM
-------------- OP2-NUM-ERA
&{}
```

I.59 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_num.md

OP2-NUM-NUM

```
(#a op #b)
-------------- OP2-NUM-NUM
#(a opr b)
```

I.60 · forgecode_clone/hvm/docs/hvm/interactions/op2_num_sup.md

OP2-NUM-SUP

```
(x op &L{a,b}) where x is NUM
------------------------ OP2-NUM-SUP
! X &L = x
&L{(X₀ op a), (X₁ op b)}
```

I.61 · forgecode_clone/hvm/docs/hvm/interactions/op2_sup.md

OP2-SUP

```
@@opr(&L{a,b}, y)
------------------------ OP2-SUP
! Y &L = y
&L{@@opr(a,Y₀), @@opr(b,Y₁)}
```

I.62 · forgecode_clone/hvm/docs/hvm/interactions/or_era.md

OR-ERA

```
(&{} .|. b)
---------- OR-ERA
&{}
```

I.63 · forgecode_clone/hvm/docs/hvm/interactions/or_inc.md

OR-INC

```
(↑a | b)
-------- OR-INC
↑(a | b)
```

I.64 · forgecode_clone/hvm/docs/hvm/interactions/or_num.md

OR-NUM

```
(#0 .|. b)
---------- OR-ZER
b

(#n .|. b)   [n ≠ 0]
------------------- OR-ONE
#1
```

I.65 · forgecode_clone/hvm/docs/hvm/interactions/or_sup.md

OR-SUP

```
(&L{a0,a1} .|. b)
------------------------ OR-SUP
! B &L = b
&L{(a0 .|. B₀), (a1 .|. B₁)}
```

I.66 · forgecode_clone/hvm/docs/hvm/interactions/uns.md

UNS

```
! ${f, v}; t
------------ WNF UNS
t(λy.λ$x.y, $x)
```

I.67 · forgecode_clone/hvm/docs/hvm/interactions/use_era.md

USE-ERA

```
(λ{f} &{})
---------- USE-ERA
&{}
```

I.68 · forgecode_clone/hvm/docs/hvm/interactions/use_inc.md

USE-INC

```
(use ↑x)
--------- USE-INC
↑(use x)
```

I.69 · forgecode_clone/hvm/docs/hvm/interactions/use_sup.md

USE-SUP

```
(λ{f} &L{a,b})
----------------- USE-SUP
! F &L = f
&L{(λ{F₀} a), (λ{F₁} b)}
```

I.70 · forgecode_clone/hvm/docs/hvm/interactions/use_val.md

USE-VAL

```
(λ{f} x)
--------- USE-VAL
(f x)
```

I.71 · forgecode_clone/hvm/test/*.hvm (aggregated)

**HVM4 test suite — every .hvm program with its expected output (aggregated at build time)*

These are the repository's own test programs. Each file is quoted verbatim; a `//...` line is the expected output where the authors wrote one. `_all_.sh` is the runner.

TEST/_ALL_.SH

```
#!/usr/bin/env bash
# test/_all_.sh
# ===============
# Unified test runner for HVM4.
# Runs interpreted and AOT-compiled modes in sequence.
#
# Test format:
#   @main = <expression>
#   //<expected output>
#
# For multi-line expected output, use multiple // lines.
# Tests starting with _ are skipped.
# Per-test CLI flags can be set with one leading directive line:
#   //!--flag1 --flag2

set -uo pipefail

DIR="$(cd "$(dirname "$0")" && pwd)"
ROOT_DIR="$(cd "$(dirname "$0")/.." && pwd)"
C_BIN="$ROOT_DIR/clang/main"
C_MAIN="${C_BIN}.c"
FFI_DIR="$DIR/ffi"

# Config
# ------

# CLI options
# -----------
interpreted_only=0
while [ $# -gt 0 ]; do
  case "$1" in
    -i|--interpreted-only)
      interpreted_only=1
      ;;
    -h|--help)
      cat <<'EOF'
usage: test/_all_.sh [--interpreted-only|-i]

Options:
  -i, --interpreted-only   Run interpreted tests only (skip AOT tests)
  -h, --help               Show this help message
EOF
      exit 0
      ;;
    *)
      echo "error: unknown option: $1" >&2
      echo "run test/_all_.sh --help for usage" >&2
      exit 1
      ;;
  esac
  shift
done

TEST_TIMEOUT_INTERPRETED_SECS=2
TEST_TIMEOUT_COMPILED_SECS=20
HVM_TMP_DIR="${TMPDIR:-/tmp}"
shared_flags=()

# Allow env overrides
if [ -n "${HVM_TEST_TIMEOUT_SECS:-}" ]; then
  TEST_TIMEOUT_INTERPRETED_SECS="$HVM_TEST_TIMEOUT_SECS"
  TEST_TIMEOUT_COMPILED_SECS="$HVM_TEST_TIMEOUT_SECS"
fi
if [ -n "${HVM_TEST_FLAGS:-}" ]; then
  read -r -a shared_flags <<< "$HVM_TEST_FLAGS"
fi
if [ -n "${HVM_TMPDIR:-}" ]; then
  HVM_TMP_DIR="$HVM_TMPDIR"
fi
if ! mkdir -p "$HVM_TMP_DIR"; then
  echo "error: failed to create temp directory '$HVM_TMP_DIR'" >&2
  exit 1
fi

ffi_ext=".so"
ffi_ldflags=(-shared -fPIC)
case "$(uname -s)" in
  Darwin*)
    ffi_ext=".dylib"
```

```
    ffi_ldflags=(-dynamiclib -fPIC)
    ;;
esac

# Timeout
# -------

# Runs a command with a timeout; stores output in the named variable
run_with_timeout() {
  local out_var="$1" t="$2" marker cap_file pid status
  shift 2

  marker="${HVM_TMP_DIR}/hvm4-test-runner.timeout.marker"
  cap_file="${HVM_TMP_DIR}/hvm4-test-runner.capture.log"

  rm -f "$marker"
  : > "$cap_file" || return 1
  "$@" >"$cap_file" 2>&1 & pid=$!
  ( sleep "$t"; kill -0 "$pid" 2>/dev/null || exit 0; : > "$marker"; kill -KILL "$pid" 2>/dev/null || true ) &
  wait "$pid" 2>/dev/null; status=$?
  [ -f "$marker" ] && status=124
  printf -v "$out_var" '%s' "$(cat "$cap_file")"
  rm -f "$cap_file" "$marker"
  return $status
}

# Build
# -----

# Compiles the C binary
if [ ! -f "$C_MAIN" ]; then
  echo "error: expected C entrypoint at $C_MAIN" >&2
  exit 1
fi
(cd "$ROOT_DIR/clang" && clang -O2 -o main main.c)

# Collect
# -------

# Tracks generated artifact files for cleanup
cleanup_files=()
cleanup() {
  if [ ${#cleanup_files[@]} -gt 0 ]; then
    rm -f "${cleanup_files[@]}"
  fi
}
trap cleanup EXIT

# Collect test files
shopt -s nullglob
tests=()
for f in "$DIR"/*.hvm "$FFI_DIR"/*.hvm; do
  name="$(basename "$f")"
  case "$name" in
    _* ) continue ;;
    *  ) tests+=("$f") ;;
  esac
done
shopt -u nullglob

if [ ${#tests[@]} -eq 0 ]; then
  echo "no .hvm files found under $DIR" >&2
  exit 1
fi

# Run
# ---

# Runs all collected tests against the given binary
run_tests() {
  local bin="$1"
  local label="$2"
  local timeout_secs="$3"
  shift 3
  local status=0

  echo "=== Testing $label ==="
  for test_file in "${tests[@]}"; do
    name="$(basename "${test_file%.hvm}")"

    # Read leading //!... directive lines as CLI flags
    extra_flags=()
```

```
while IFS= read -r line; do
  if [[ "$line" == "//! "* ]]; then
    flag_line="${line#//! }"
    read -r -a more_flags <<< "$flag_line"
    extra_flags+=("${more_flags[@]}")
    continue
  fi
  if [[ "$line" == "//!-"* ]]; then
    flag_line="${line#//!}"
    read -r -a more_flags <<< "$flag_line"
    extra_flags+=("${more_flags[@]}")
    continue
  fi
  if [[ -z "$line" ]]; then
    continue
  fi
  break
done < "$test_file"

# Extract trailing // comment lines (consecutive from end of file)
expected=""
nlines_expected=0
nocollapse=0
expect_prefix=""
expect_contains=""
while IFS= read -r line; do
  if [[ "$line" == //* ]]; then
    if [[ "$line" == "//EXPECT_PREFIX:"* ]]; then
      expect_prefix="${line#//EXPECT_PREFIX:}"
      continue
    fi
    if [[ "$line" == "//EXPECT_CONTAINS:"* ]]; then
      expect_contains="${line#//EXPECT_CONTAINS:}"
      continue
    fi
    if [[ "$line" == '//!'* ]]; then
      nocollapse=1
      content="${line#//!}"
    else
      content="${line#//}"
    fi
    [ -n "$expected" ] && expected="${content}"$'\n'"$expected"
    [ -z "$expected" ] && expected="${content}"
    ((nlines_expected++))
  else
    break
  fi
done < <(tail -r "$test_file" 2>/dev/null || tac "$test_file")

# For collapse_* and enum_* tests, infer limit from expected output lines
collapse_count=""
if [[ "$name" == collapse_* || "$name" == enum_* ]]; then
  collapse_count="$nlines_expected"
fi

if [ $nlines_expected -eq 0 ] && [ -z "$expect_prefix" ] && [ -z "$expect_contains" ]; then
  echo "[FAIL] $name (missing expected result comment)" >&2
  status=1
  continue
fi

# Determine flags: all tests use -C by default unless //! is used
flags=""
if [ "$nocollapse" -eq 0 ]; then
  flags="-C"
  case "$name" in
    collapse_* | enum_* )
      [ -n "$collapse_count" ] && flags="${flags}${collapse_count}"
      ;;
  esac
fi

# Build FFI flags if applicable
ffi_flag=""
ffi_target=""
if [[ "$test_file" == "$FFI_DIR/"* ]]; then
  base="${test_file%.hvm}"
  if [ -d "$base" ]; then
    ffi_flag="--ffi-dir"
    ffi_target="$base"
    shopt -s nullglob
    c_files=("$base"/*.c)
```

```
    shopt -u nullglob
    if [ ${#c_files[@]} -eq 0 ]; then
      echo "[FAIL] $name (no .c files under $base)" >&2
      status=1
      continue
    fi
    for src in "${c_files[@]}"; do
      out="${src%.c}${ffi_ext}"
      if ! clang "${ffi_ldflags[@]}" -I "$ROOT_DIR" -o "$out" "$src"; then
        echo "[FAIL] $name (failed to build $src)" >&2
        status=1
        continue 2
      fi
      cleanup_files+=("$out")
    done
  else
    src="${base}.c"
    out="${base}${ffi_ext}"
    if [ ! -f "$src" ]; then
      echo "[FAIL] $name (missing $src)" >&2
      status=1
      continue
    fi
    if ! clang "${ffi_ldflags[@]}" -I "$ROOT_DIR" -o "$out" "$src"; then
      echo "[FAIL] $name (failed to build $src)" >&2
      status=1
      continue
    fi
    cleanup_files+=("$out")
    ffi_flag="--ffi"
    ffi_target="$out"
  fi
fi

# Assemble the command
cmd=("$bin" "$test_file")
if [ $# -gt 0 ]; then
  cmd+=("$@")
fi
if [ -n "$flags" ]; then
  cmd+=("$flags")
fi
if [ ${#extra_flags[@]} -gt 0 ]; then
  cmd+=("${extra_flags[@]}")
fi
if [ -n "$ffi_flag" ]; then
  cmd+=("$ffi_flag" "$ffi_target")
fi

# Execute and compare
run_with_timeout actual "$timeout_secs" "${cmd[@]}"
cmd_status=$?
if [ $cmd_status -eq 124 ]; then
  echo "[FAIL] $name (timeout after ${timeout_secs}s)"
  status=1
  continue
fi

# Strip ANSI escape codes for comparison
actual_clean="$(echo "$actual" | sed 's/\x1b\[[0-9;]*m//g')"
expected_clean="$(echo "$expected" | sed 's/\x1b\[[0-9;]*m//g')"

# Compare output
if [ -n "$expect_prefix" ]; then
  if [[ "$actual_clean" == "$expect_prefix"* ]]; then
    echo "[PASS] $name"
  else
    echo "[FAIL] $name"
    echo "  expected prefix: $expect_prefix"
    echo "  detected: $actual_clean"
    status=1
  fi
elif [ -n "$expect_contains" ]; then
  if [[ "$actual_clean" == *"$expect_contains"* ]]; then
    echo "[PASS] $name"
  else
    echo "[FAIL] $name"
    echo "  expected to contain: $expect_contains"
    echo "  detected: $actual_clean"
    status=1
  fi
elif [ "$expected_clean" = "PARSE_ERROR" ]; then
```

```
      if [[ "$actual_clean" == PARSE_ERROR* ]]; then
        echo "[PASS] $name"
      else
        echo "[FAIL] $name"
        echo "  expected: PARSE_ERROR"
        echo "  detected: $actual_clean"
        status=1
      fi
    elif [ "$actual_clean" = "$expected_clean" ]; then
      echo "[PASS] $name"
    else
      echo "[FAIL] $name"
      echo "  expected: $expected"
      echo "  detected: $actual"
      status=1
    fi
  done
  echo ""
  return $status
}

# Main
# ----

status=0
if [ -n "${HVM_TEST_FLAGS:-}" ]; then
  run_tests "$C_BIN" "HVM (interpreted)" "$TEST_TIMEOUT_INTERPRETED_SECS" "${shared_flags[@]}"
else
  run_tests "$C_BIN" "HVM (interpreted)" "$TEST_TIMEOUT_INTERPRETED_SECS"
fi
if [ $? -ne 0 ]; then
  status=1
fi

if [ "$interpreted_only" -eq 0 ]; then
  if [ -n "${HVM_TEST_FLAGS:-}" ]; then
    run_tests "$C_BIN" "HVM (AOT)" "$TEST_TIMEOUT_COMPILED_SECS" "${shared_flags[@]}" "--as-c"
  else
    run_tests "$C_BIN" "HVM (AOT)" "$TEST_TIMEOUT_COMPILED_SECS" "--as-c"
  fi
  if [ $? -ne 0 ]; then
    status=1
  fi
fi

if [ $status -eq 0 ]; then
  echo "All tests passed!"
  exit 0
else
  exit 1
fi
```

TEST/_BOOK_.HVM

```
 @ctru = λt. λf. t
@cfal = λt. λf. f
@cnot = λb. λt. λf. b(f, t)
@cadd = λa. λb. λs. λz. !S&B=s; a(S₀, b(S₁, z))
@cmul = λa. λb. λs. λz. a(b(s), z)
@cexp = λa. λb. b(a)
@c1   = λs. λx. s(x)
@k1   = λs. λx. s(x)
@c2   = λs. !S0&C=s; λx0.S0₀(S0₁(x0))
@k2   = λs. !S0&K=s; λx0.S0₀(S0₁(x0))
@c4   = λs. !S0&C=s; !S1&C=λx0.S0₀(S0₁(x0)); λx1.S1₀(S1₁(x1))
@k4   = λs. !S0&K=s; !S1&K=λx0.S0₀(S0₁(x0)); !S2&K=λx1.S1₀(S1₁(x1)); λx3.S2₀(S2₁(x3))
@c8   = λs. !S0&C=s; !S1&C=λx0.S0₀(S0₁(x0)); !S2&C=λx1.S1₀(S1₁(x1)); λx3.S2₀(S2₁(x3))
@k8   = λs. !S0&K=s; !S1&K=λx0.S0₀(S0₁(x0)); !S2&K=λx1.S1₀(S1₁(x1)); λx3.S2₀(S2₁(x3))
@mul2 = λ{#Z:#Z{}; λ{#S:λp.#S{#S{@mul2(p)}}; &{}}}
@add  = λ{#Z:λb.b; λ{#S:λa.λb.#S{@add(a, b)}; &{}}}
@sum  = λ{#Z:#Z{}; λ{#S:λp.!P&S=p;#S{@add(P₀, @sum(P₁))}; &{}}}
```

TEST/AOT_HELPER_NAME_COLLISION.HVM

```
// AOT helper names must not collide with user definitions like @foo_1.
@foo = λ{#A:1; λx.2}
@foo_1 = 7
@main = [@foo(#A{}), @foo(#B{}), @foo_1]
//[1,2,7]
```

TEST/APP_LAM.HVM

```
@foo = λx. λy. λz. y
@main = @foo(1, 2, 3)
//2
```

TEST/AUTODUP_1.HVM

```
// 1 use: no dup inserted
@f = λ&x. [x]
@main = @f(5)
//[5]
```

TEST/AUTODUP_2.HVM

```
// 2 uses: 1 dup inserted
@f = λ&x. [x,x]
@main = @f(5)
//[5,5]
```

TEST/AUTODUP_3.HVM

```
// 3 uses: 2 dups inserted
@f = λ&x. [x,x,x]
@main = @f(7)
//[7,7,7]
```

TEST/AUTODUP_4.HVM

```
// 4 uses: 3 dups inserted
@f = λ&x. [x,x,x,x]
@main = @f(7)
//[7,7,7,7]
```

TEST/AUTODUP_BOUND_VARS.HVM

```
// Test: cloning bound variables through complex structures
// Focus on lambdas capturing cloned vars, lists with vars, nested cloning

@test1 =
  λ&a. λ&b.                              // cloned outer vars
  ! &X & = [a, λk. [b, k]];              // X is list with lambda capturing b
  [X₀, X₀, X₁]                           // X used 3 times total

@test2 =
  λ&a. λ&b. λ&c.
  ! &P & = [a, [b, c]];                  // nested list with all cloned vars
  ! &Q & = [P₀, P₁];                     // Q captures P refs
  [[P₀, P₀], [Q₀, Q₁]]                   // mix of P and Q uses

@test3 =
  λ&f.                                   // f is a cloned function
  ! &X & = [f(1), f(2)];                 // X contains applications of f
  [X₀, X₀, X₁]                           // X used 3 times

@main = [
  @test1(10)(20),
  @test2(1)(2)(3),
  @test3(λx. x + 100)
]
//[[[10,λa.[20,a]],[10,λa.[20,a]],[10,λa.[20,a]]],[[[1,[2,3]],[1,[2,3]]],[[[1,[2,3]],[1,[2,3]]],[[1,[2,3]],[1,[2,3]]]]],[[101,102],[101,102],[101,102]]]
```

TEST/AUTODUP_DEEP.HVM

```
// Deeply nested cloned lambdas
@f = λ&a. λ&b. λ&c. [a,a,b,b,c,c]
@main = @f(1)(2)(3)
//[1,1,2,2,3,3]
```

TEST/AUTODUP_DUP_DYN_3EACH.HVM

```
// Auto-dup on dynamic dup: 3 uses each side
@main = ! &X &(0) = 5; [[X₀, X₀, X₀], [X₁, X₁, X₁]]
//[[5,5,5],[5,5,5]]
```

TEST/AUTODUP_DUP_DYN_BOTH.HVM

```
// Auto-dup on dynamic dup: both sides used multiple times
@main = ! &X &(0) = 5; [[X₀, X₀], [X₁, X₁]]
//[[5,5],[5,5]]
```

TEST/AUTODUP_DUP_DYN.HVM

```
// Auto-dup on dynamic dup: ! &X &(L) = v; [X₀, X₀, X₁]
@main = ! &X &(0) = 10; [X₀, X₀, X₁]
//[10,10,10]
```

TEST/AUTODUP_DUP_STATIC_3EACH.HVM

```
// Auto-dup on static dup: X₀ and X₁ both used 3 times
@main = ! &X & = 5; [[X₀, X₀, X₀], [X₁, X₁, X₁]]
//[[5,5,5],[5,5,5]]
```

TEST/AUTODUP_DUP_STATIC_BOTH.HVM

```
// Auto-dup on static dup: both sides used twice
@main = ! &X & = 3; [[X₀, X₀], [X₁, X₁]]
//[[3,3],[3,3]]
```

TEST/AUTODUP_DUP_STATIC.HVM

```
// Auto-dup on static dup: ! &X &L = v; [X₀, X₀, X₁]
@main = ! &X &L = 10; [X₀, X₀, X₁]
//[10,10,10]
```

TEST/AUTODUP_LET.HVM

```
// Auto-dup with let binding: ! &x = val
@main = ! &x = 2 + 3; x * x
//25
```

TEST/AUTODUP_MANY.HVM

```
// Many uses (8)
@f = λ&x. [x,x,x,x,x,x,x,x]
@main = @f(9)
//[9,9,9,9,9,9,9,9]
```

TEST/AUTODUP_MAT_LAM.HVM

```
// Auto-dup in match branch with inner lambda
@fun = λ&v. λ{
  #A: [v,v]
  λx. x
}
@main = @fun(0, #B)
//#B{}
```

TEST/AUTODUP_MIXED.HVM

```
// Mix of cloned and non-cloned variables
@f = λ&x. λy. [x,x,y]
@main = @f(1)(2)
//[1,1,2]
```

TEST/AUTODUP_NESTED_LAM.HVM

```
// Nested cloned lambdas
@f = λ&x. λ&y. [[x,x], [y,y]]
@main = @f(1)(2)
//[[1,1],[2,2]]
```

TEST/AUTODUP_NIGHTMARE.HVM

```
// Nightmare test: clone complex expressions with bound vars, lambdas, apps, nested structures
// Uses lambda variables (not applied) to create well-typed stuck applications

@nightmare =
  λ&f. λ&g. λh.                          // f, g cloned; h not
  λ&x. λ&y.                              // x, y cloned
  ! &A &(0) = λk. f(x, g(λu. u(h, k))); // lambda with apps of outer vars
  ! &B & = [A₀(y), λv. [v, A₁(x)]];     // list with A applied, lambda with A
  ! &C &(1) = [B₀, g(y), f(x, B₁)];     // list using B and outer funcs
  [C₀, C₀, C₁]

@main = @nightmare
//λa.λb.λc.λd.λe.[[[a(d,b(λf.f(c,e))),λf.[f,a(d,b(λg.g(c,d)))]],b(e),a(d,[a(d,b(λf.f(c,e))),λf.[f,a(d,b(λg.g(c,d)))]])],[[a(d,b(λf.f(c,e))),λf.
[f,a(d,b(λg.g(c,d)))]],b(e),a(d,[a(d,b(λf.f(c,e))),λf.[f,a(d,b(λg.g(c,d)))]])],[[a(d,b(λf.f(c,e))),λf.[f,a(d,b(λg.g(c,d)))]],b(e),a(d,
[a(d,b(λf.f(c,e))),λf.[f,a(d,b(λg.g(c,d)))]])]]
```

TEST/AUTODUP_OUTER_REF.HVM

```
// Cloned inner var with outer var reference
@f = λ&x. λ&y. [x,y,y]
@main = @f(1)(2)
//[1,2,2]
```

TEST/AUTODUP_STRICT.HVM

```
// Auto-dup with strict let: !! &x = val
@main = !! &x = 1 + 2; [x,x]
//[3,3]
```

TEST/AUTODUP_SUP_MIX.HVM

```
// Test: superpositions mixed with cloned bindings
// SUP branches may have different auto-dup behavior

@sup_test =
  ! &X & = 10;                           // cloned dup
  λ&y.                                   // cloned lambda
  &L{                                    // superposition
    [X₀, X₀, y, y],                      // branch 0: uses X₀ twice, y twice
    [X₁, X₁, X₁, y]                      // branch 1: uses X₁ three times, y once
  }

@main = @sup_test(99)
//[10,10,99,99]
//[10,10,10,99]
```

TEST/AUTODUP_VALUE_PATH.HVM

```
// Test: cloned refs inside dup VALUE positions (not just bodies)
// The value path uses idx+0, dep+0 which could have different behavior

@val_test =
  ! &X & = 100;                          // X at level 0, cloned
  ! A & = [X₀, X₀, X₁];                  // X refs in A's VALUE
  ! B & = λ&y. [X₀, X₀, y, y];           // X refs + cloned y in B's VALUE
  [A₀, A₁, B₀(1), B₁(2)]                 // Use all the dups

@main = @val_test
//[[100,100,100],[100,100,100],[100,100,1,1],[100,100,2,2]]
```

TEST/BIND_LAM_FORK.HVM

```
@main = λ&x. &Lλy{[x,y];[x,y]}
//λa.λb.[a,b]
//λa.λb.[a,b]
```

TEST/BIND_SHADOW_DUP_LAM.HVM

```
@main = λ&x&.λ&x.[x₀,x₀,x₁,x₁,x,x]
//λa.λb.[a,a,a,a,b,b]
```

TEST/BIND_SHADOW_LAM_DUP.HVM

```
@main = λ&x.λ&x&.[x₀,x₀,x₁,x₁,x,x]
//λa.λb.[b,b,b,b,a,a]
```

TEST/C2_NOT_T.HVM

```
#include "_book_.hvm"
@main = @c2(@cnot, @ctru)
//λa.λb.a
```

TEST/C8_K8_NOT_T.HVM

```
#include "_book_.hvm"
@main = @c8(@k8, @cnot, @ctru)
//λa.λb.a
```

TEST/CADD_C1_C1_NOT.HVM

```
#include "_book_.hvm"
@main = @cadd(@c1, @c1, @cnot)
//λa.λb.λc.a(b,c)
```

TEST/CADD_C1_C4_NOT_T.HVM

```
#include "_book_.hvm"
@main = @cadd(@c1, @c4, @cnot, @ctru)
//λa.λb.b
```

TEST/CADD_C1_C4.HVM

```
#include "_book_.hvm"
@main = @cadd(@c1, @c4)
//λa.λb.a(a(a(a(a))))
```

TEST/CADD_C2_C1.HVM

```
#include "_book_.hvm"
@main = @cadd(@c2, @c1)
//λa.λb.a(a(a(b)))
```

TEST/CADD_C2_C2.HVM

```
#include "_book_.hvm"
@main = @cadd(@c2, @c2)
//λa.λb.a(a(a(a(b))))
```

TEST/CADD_C2_C4_NOT_T.HVM

```
#include "_book_.hvm"
@main = @cadd(@c2, @c4, @cnot, @ctru)
//λa.λb.a
```

TEST/CADD_C4_C1_NOT_T.HVM

```
#include "_book_.hvm"
@main = @cadd(@c4, @c1, @cnot, @ctru)
//λa.λb.b
```

TEST/CADD_C4_C1.HVM

```
#include "_book_.hvm"
@main = @cadd(@c4, @c1)
//λa.λb.a(a(a(a(a(b)))))
```

TEST/CADD_C4_C2_NOT_T.HVM

```
#include "_book_.hvm"
@main = @cadd(@c4, @c2, @cnot, @ctru)
//λa.λb.a
```

TEST/CADD_C4_C4.HVM

```
#include "_book_.hvm"
@main = @cadd(@c4, @c4)
//λa.λb.a(a(a(a(a(a(a(a(b)))))))) 
```

TEST/CMUL_C4_C2_NOT_T.HVM

```
#include "_book_.hvm"
@main = @cmul(@c4, @c2, @cnot, @ctru)
//λa.λb.a
```

TEST/CMUL_C4_C2.HVM

```
#include "_book_.hvm"
@main = @cmul(@c4, @c2)
//λa.λb.a(a(a(a(a(a(a(a(b)))))))) 
```

TEST/CMUL_C4_C4_NOT_T.HVM

```
#include "_book_.hvm"
@main = @cmul(@c4, @c4, @cnot, @ctru)
//λa.λb.a
```

TEST/CMUL_C4_C4.HVM

```
#include "_book_.hvm"
@main = @cmul(@c4, @c4)
//λa.λb.a(a(a(a(a(a(a(a(a(a(a(a(a(a(a(a(b))))))))))))))))
```

TEST/CNOT_NOT.HVM

```
#include "_book_.hvm"
@main = @cnot(@cnot(@ctru))
//λa.λb.a
```

TEST/CNOT_TRUE.HVM

```
#include "_book_.hvm"
@main = @cnot(@ctru)
//λa.λb.b
```

TEST/COLLAPSE_0.HVM

```
#include "_book_.hvm"
@main = λt.t(&A{#S{#Z{}},#S{#S{#Z{}}}}, #S{#S{#S{#Z{}}}})
//λa.a(#S{#Z{}},#S{#S{#S{#Z{}}}})
//λa.a(#S{#S{#Z{}}},#S{#S{#S{#Z{}}}})
```

TEST/COLLAPSE_1.HVM

```
#include "_book_.hvm"
@main = λt.t(&A{#S{#Z{}},#S{#S{#Z{}}}}, &A{#S{#S{#S{#Z{}}}},#S{#S{#S{#S{#Z{}}}}}})
//λa.a(#S{#Z{}},#S{#S{#S{#Z{}}}})
//λa.a(#S{#S{#Z{}}},#S{#S{#S{#S{#Z{}}}}})
```

TEST/COLLAPSE_2.HVM

```
#include "_book_.hvm"
@main = λt.t(&A{#S{#Z{}},#S{#S{#Z{}}}}, &B{#S{#S{#S{#Z{}}}},#S{#S{#S{#S{#Z{}}}}}})
//λa.a(#S{#Z{}},#S{#S{#S{#Z{}}}})
//λa.a(#S{#Z{}},#S{#S{#S{#S{#Z{}}}}})
//λa.a(#S{#S{#Z{}}},#S{#S{#S{#Z{}}}})
//λa.a(#S{#S{#Z{}}},#S{#S{#S{#S{#Z{}}}}})
```

TEST/COLLAPSE_3.HVM

```
#include "_book_.hvm"
@main = λt.t(#S{#Z{}}, &B{#S{#S{#S{#Z{}}}},#S{#S{#S{#S{#Z{}}}}}})
//λa.a(#S{#Z{}},#S{#S{#S{#Z{}}}})
//λa.a(#S{#Z{}},#S{#S{#S{#S{#Z{}}}}})
```

TEST/COLLAPSE_4.HVM

```
#include "_book_.hvm"
@main = λa.!A&L=a;&L{A₀,A₁}
//λa.a
//λa.a
```

TEST/COLLAPSE_5.HVM

```
#include "_book_.hvm"
@main = λa.λb.!A&L=a;!B&L=b;&L{λx.x(A₀,B₀),λx.x(A₁,B₁)}
//λa.λb.λc.c(a,b)
//λa.λb.λc.c(a,b)
```

TEST/COLLAPSE_6.HVM

```
#include "_book_.hvm"
@main = #S{&A{&B{#Z{},#S{#Z{}}},&C{#S{#S{#Z{}}},#S{#S{#S{#Z{}}}}}}}
//#S{#Z{}}
//#S{#S{#Z{}}}
//#S{#S{#S{#Z{}}}}
//#S{#S{#S{#S{#Z{}}}}}
```

TEST/COLLAPSE_7.HVM

```
#include "_book_.hvm"
@main = #S{&L{#Z{},#S{#Z{}}}}
//#S{#Z{}}
//#S{#S{#Z{}}}
```

TEST/COLLAPSE_8.HVM

```
#include "_book_.hvm"
@main = #T{&A{#Z{}, #Z{}}, &A{#Z{}, #S{#S{#Z{}}}}}
//#T{#Z{},#Z{}}
//#T{#Z{},#S{#S{#Z{}}}}
```

TEST/COLLAPSE_9.HVM

```
// Tests lazy enumeration of infinite structures
// The stream @X = &L{#Z, #S{@X}} enumerates as #Z, #S{#Z}, #S{#S{#Z}}, ...
@X = &L{#Z, #S{@X}}
@main = @X
//#Z{}
//#S{#Z{}}
//#S{#S{#Z{}}}
//#S{#S{#S{#Z{}}}}
//#S{#S{#S{#S{#Z{}}}}}
//#S{#S{#S{#S{#S{#Z{}}}}}}
//#S{#S{#S{#S{#S{#S{#Z{}}}}}}}
//#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}
//#S{#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}}
//#S{#S{#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}}}
```

TEST/COLLAPSE_ERA_CTR.HVM

```
// Test: #F{&{}} collapses to ERA, which is not printed (universe destroyed)
@main = #F{&{}}
//
```

TEST/COLLAPSE_ERA_DEEP.HVM

```
// Test: λx.[0, λy.λz.z(&{})] collapses to ERA, which is not printed (universe destroyed)
@id = λx.x
@main = λx.[0, λy.λz.@id(&{})]
//
```

TEST/COLLAPSE_ERA_MULTI_CTR.HVM

```
// Test: #A{#B{&{}}} collapses to ERA, which is not printed (universe destroyed)
@main = #A{#B{&{}}}
//
```

TEST/COLLAPSE_ERA_NESTED.HVM

```
// Test: λx.λy.λz.&{} collapses to ERA, which is not printed (universe destroyed)
@main = λx.λy.λz.&{}
//
```

TEST/COLLAPSE_ERA_PAIR.HVM

```
// Test: [0, &{}] collapses to ERA, which is not printed (universe destroyed)
@main = [0, &{}]
//
```

TEST/COLLAPSE_ERA.HVM

```
// Test: ERA propagation through collapse
// λx.&{} collapses to ERA, which is not printed (universe destroyed)
@main = λx.&{}
//
```

TEST/COLLAPSE_INC_LAM.HVM

```
// Test: INC inside lambda should not leak SUP in CNF
@main = λx. ↑(&A{0, 1})
//λa.0
//λa.1
```

TEST/COLLAPSE_SUPGEN_MUL.HVM

```
@X=λ&f.f(@X(f))
@W=λ{[]:λx.x;<>:λ{0:λt.&{};1:λt.λx.@W(t,x)}}
@A=λ{[]:λys.ys;<>:λx.λxs.λys.x<>@A(xs,ys)}
@M=λ&f.λ{[]:[];<>:λx.λxs.f(x)<>@M(f,xs)}
@Y=λx.(x*2+0)%16777216
@Z=λx.(x*2+1)%16777216
@P=λF.λT.λ{#B:λ{#A:λA.λB.λv.#B{T(B),F(v)}}}
@S=λF.λT.λR.@M(@P(F,T),R)
@L=λ&L.λ{#A:λ{#N:λB&(L).λF&(L).λS&(L).λI&(L).λU&(L).λC&(L).&(L)
{@V(L,#N,B₀,F₀,S₀,I₀,U₀,C₀),@N(L,B₁,F₁,S₁,I₁,U₁,C₁)}#Z:λA.λB.λF.λS.λI.λU.λC.@V(L,A,B,F,S,I,U,C)λA.λB.λF.λS.λI.λU.λC.@V(L,A,B,F,S,I,U,C)};λT.λF.λS.λI.λU.λC
.@E(L,T,F,I,U,C)}
@V=λL.λA.λB.λF.λS.λI.λU.λC.λ&x.@L(L,B(x),F,λk.S(k,x),I,@S(λv.λk.v(k,x),λt.t(x),U),@A(C,[#B{A,x}]))
@N=λ&L.λ&B.λ&F.λ&S.λ&I.λ&U.λ&C.λ{0n:@L(@Y(L),B(0n),F,λk.S(k,0n),I,@S(λv.λk.v(k,0n),λt.t(0n),U),C);1n+:@L(@Z(L),#A{#Z{#N},λx.B(1n+x)},F,λk.λx.S(k,1n+x),I,@
A(@S(λv.λk.λx.v(k,#SUC{x}),λt.#A{#N,λx.t(#SUC{x})},U),[#B{#A{#N,λx.B(x)},λk.λx.S(k,x)}]),C)}
@E=λ&L.λT&(L).λF&(L).λI&(L).λU&(L).λC&(L).&(L){@I(L,I₀,T₀,F₀,U₀,C₀),@K(L,T₁,F₁,I₁,U₁,C₁)}
@K=λ&L.λ{#N:λF&(L).λI&(L).λU&(L).λC&(L).&(L){0n,1n+@E(L,#N,F₁,I₁,U₁,C₁)}λT.λF.λI.λU.λC.&{}}
@I=λ&L.λ{[]:λT.λF.λU.λC.@U(L,U,T,F,C);<>:λ{#B:λIT&(L).λIV&(L).λI&(L).λT&(L).λF&(L).λU&(L).λC&(L).&(L){@F(L,IT₀,T₀,F₀,I₀,U₀,C₀,IV₀),@I(L,I₁,T₁,F₁,U₁,C₁)}}}
@U=λ&L.λ{[]:λT.λF.λC.@C(L,C,T,F);<>:λ{#B:λUT&(L).λUV&(L).λU&(L).λT&(L).λ&F&(L).λC&(L).&(L){@F(L,UT₀,T₀,F₀,[],U₀,C₀,UV₀(F₀)),@U(L,U₁,T₁,F₁,C₁)}}}
@C=λ&L.λ{[]:λT.λF.&{};<>:λ{#B:λCT&(L).λCV&(L).λC&(L).λT&(L).λF&(L).&(L){@F(L,CT₀,T₀,F₀,[],[],C₀,CV₀),@C(L,C₁,T₁,F₁)}}}
@F=λ&L.λ{#A:λ&A.λB.λ&T.λ&F.λ&I.λ&U.λ&C.λv.@F(@Z(L),B(@E(@Y(L),A,F,I,U,C)),T,F,I,U,C,v(@E(@Y(L),A,F,I,U,C)))λt.λT.λF.λI.λU.λC.λv.@W([T===t],v)}
@main=!&a=λF.λ{0n:λb.b;1n+:λa.λb.1n+F(a,b)};!&F=λa.λF.@L(1,#A{#N,λx.#A{#N,λy.#N}},F,λk.k,[#B{#A{#N,λx.#A{#N,λy.#N}},a}],[],[]);@W([@X(F(@X(a)))
(2n,3n)===6n,@X(F(@X(a)))(3n,3n)===9n,@X(F(@X(a)))(4n,1n)===4n,@X(F(@X(a)))(4n,2n)===8n],F)
//λa.λb.λc.λ{0n:0n;1n+:λd.a(c,b(c,d))}
//λa.λb.λ{0n:λc.0n;1n+:λc.λd.a(d,b(c,d))}
//λa.λb.λ{0n:λ{0n:0n;1n+:λc.0n};1n+:λc.λd.a(d,b(c,d))}
//λa.λb.λ{0n:λ{0n:1n;1n+:λc.0n};1n+:λc.λd.a(d,b(c,d))}
```

TEST/DUP_AFFINE_BARE.HVM

```
// Error: bare x refers to dup binding, but outer x has no capacity
@main = λx. !x&L=x; (x₀ + x)
//PARSE_ERROR
```

TEST/DUP_AFFINE_CLONED.HVM

```
// OK: d is cloned so d₀ can be used multiple times
@main = (λx. !&d&L=x; (d₀ + d₀ + d₁))(5)
//15
```

TEST/DUP_AFFINE_CO0.HVM

```
// Error: d₀ used twice without cloning
@main = λx. !d&L=x; (d₀ + d₀)
//PARSE_ERROR
```

TEST/DUP_AFFINE_CO1.HVM

```
// Error: d₁ used twice without cloning
@main = λx. !d&L=x; (d₁ + d₁)
//PARSE_ERROR
```

TEST/DUP_AFFINE_SHADOW_OK.HVM

```
// OK: bare x falls through to cloned outer x
@main = (λ&x. !x&L=x; (x₀ + x))(5)
//10
```

TEST/DUP_AFFINE_VALID.HVM

```
// OK: each side used once
@main = (λx. !d&L=x; (d₀ + d₁))(5)
//10
```

TEST/DUP_FRESH.HVM

```
// Auto-label dup: !x& = val desugars to !x&N = val with fresh N
@main = !x& = 5; x₀ + x₁
//10
```

TEST/DUP_REF.HVM

```
#include "_book_.hvm"
@main = !F&LL=@mul2; F₀(F₁(#S{#S{#Z{}}}))
//#S{#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}}
```

TEST/DYN_DUP_NESTED.HVM

```
@main = !X&(1)=10; !Y&(2)=X₀; λf.f(X₁,Y₀,Y₁)
//λa.a(10,10,10)
```

TEST/DYN_DUP_SUP.HVM

```
@main =
  ! X &(1) = &(1){10,20};
  [X₀, X₁]
//[10,20]
```

TEST/ENUM_BITS.HVM

```
#include "_book_.hvm"

// Bits = { #E{} #O{p} #I{p} }

@two = #S{#S{#Z{}}}

@bits = λ{
  #Z: #E{} ;
  #S: λp.!P&L=p ; &L{ #O{@bits(P₀)}, #I{@bits(P₁)} }
}

@main = @bits(@two)
//#O{#O{#E{}}}
//#O{#I{#E{}}}
//#I{#O{#E{}}}
//#I{#I{#E{}}}
```

TEST/ENUM_NAT.HVM

```
// data Nat { #Z #S{pred} }

@if = λ{
  #T: λt.λf.t
  #F: λt.λf.f
}

@add = λ{ #Z: λb.b ; #S: λp.λb.#S{@add(p, b)}; }
@mul = λ{ #Z: λb.#Z{} ; #S: λp.λb. !B&A=b ; @add(B₀, @mul(p, B₁)); }

@eq = λ{
  #Z: λ{
    #Z: #T{}
    #S: λbp. #F{}
  };
  #S: λap.λ{
    #Z: #F{}
    #S: λbp. @eq(ap, bp)
  };
}

@X = &Z{#Z{}, #S{@X}}

@two = #S{#S{#Z{}}}
@four = #S{#S{#S{#S{#Z{}}}}}

// solves x + 2 = 4
@main = @if(@eq(@add(@X, @two), @four), @X, &{})
//#S{#S{#Z{}}}
```

TEST/ENUM_PRIMES.HVM

```
@if = λ{
  0:    λt.λf.f
  _: λc.λt.λf.t
}

@u32 = λ{
  #O: λp. 2*@u32(p) + 0
  #I: λp. 2*@u32(p) + 1
  #E: 0
}

@bin = λ{
  0: λ  &n. #E
  _: λs,&n. @bin_go(n%2, s-1, n)
}

@bin_go = λ{
  0:    λp,n. #O{@bin(p, n/2)}
  _: λe.λp,n. #I{@bin(p, n/2)}
}

@eq = λ{
  #E: λ{
    #O: λbp.0
    #I: λbp.0
    #E: 1
  }
  #O: λap,{
    #O: λbp.@eq(ap bp)
    #I: λbp.0
    #E: 0
  }
  #I: λap.λ{
    #O: λbp.0;
    #I: λbp.@eq(ap,bp);
    #E: 0;
  }
}

@inc = λ{
  #O: λp. #I{p}
  #I: λp. #O{@inc(p)}
  #E: #E
}

@dec = λ{
  #O: λp. #O{@dec(p)}
  #I: λp. #I{p}
  #E: #E
}

@add = λ{
  #O: λ&ap. λ{
    #O: λbp. #O{@add(ap, bp)}
    #I: λbp. #I{@add(ap, bp)}
    #E: #E
  }
  #I: λ&ap. λ{
    #O: λbp. #I{@add(ap, bp)}
    #I: λbp. #O{@inc(@add(ap, bp))}
    #E: #I{ap}
  }
  #E: λb. #E
}

@mul = λ&a. λ{
  #O: λbp. #O{@mul(a, bp)}
  #I: λbp. @add(a, #O{@mul(a, bp)})
  #E: #E
}

@cat = λ{
  #O: λap, b. #O{@cat(ap, b)}
  #I: λap, b. #I{@cat(ap, b)}
  #E: λb.b
}

@all1 = λ{
  0: #E;
```

```
  _: λs. !p&L = s-1; &L{
    #O{@all1(p₀)}
    #I{@all1(p₁)}
  }
}

@all2 = λ{
  0: #E;
  _: λs. !p&M = s-1; &M{
    #O{@all2(p₀)}
    #I{@all2(p₁)}
  }
}

@K_B = 9
@H_B = 10
@S_B = 20
@X_B = @cat(@all1(@H_B) @bin(@H_B, 0))
@Y_B = @cat(@all2(@H_B) @bin(@H_B, 0))
@P_B = #I{#I{#I{#I{#O{#I{#I{#O{#I{#I{#O{#O{#O{#O{#I{#I{#I{#E}}}}}}}}}}}}}}}}}

@main = @if(@eq(@mul(@X_B @Y_B), @P_B), [@u32(@X_B), @u32(@Y_B)], &{})
//[853,947]
//[947,853]
```

TEST/EQL_APP_FUN.HVM

```
// Test: structural equality involving function application
@add = λx.λy.(x + y)
@main = @add(1, 2) === 3
//1
```

TEST/EQL_CTR_EMPTY.HVM

```
// Test: structural equality of nullary constructors
@main = #Unit{} === #Unit{}
//1
```

TEST/EQL_CTR_F1.HVM

```
// Test: structural equality of constructors (same tag, different fields)
@main = #Pair{1,2} === #Pair{1,3}
//0
```

TEST/EQL_CTR_F2.HVM

```
// Test: structural equality of constructors (different tag)
@main = #Pair{1,2} === #Triple{1,2}
//0
```

TEST/EQL_CTR_T.HVM

```
// Test: structural equality of constructors (same tag, same fields)
@main = #Pair{1,2} === #Pair{1,2}
//1
```

TEST/EQL_DUP_LAM_CTR_LEFT.HVM

```
// Test: structural equality should preserve both cloned branches when a
// duplicated lambda compares a constructor on the left.
@main = !f& = λa. #A{} === a; f₀(#A{})
//1
```

TEST/EQL_DUP.HVM

```
// Test: structural equality with duplicated variables
@main = !x &a = 42; (x₀ === x₁)
//1
```

TEST/EQL_LAM_F.HVM

```
// Test: structural equality of lambdas (not equal)
@main = (λx.x) === (λy.0)
//0
```

TEST/EQL_LAM_NESTED_F.HVM

```
// Test: structural equality of nested lambdas (different)
@main = (λx.λy.x) === (λa.λb.b)
//0
```

TEST/EQL_LAM_NESTED.HVM

```
// Test: structural equality of nested lambdas
@main = (λx.λy.x) === (λa.λb.a)
//1
```

TEST/EQL_LAM_T.HVM

```
// Test: structural equality of lambdas (equal - identical bodies)
@main = (λx.x) === (λy.y)
//1
```

TEST/EQL_LIST_F.HVM

```
// Test: structural equality of lists (different)
@main = [1,2,3] === [1,2,4]
//0
```

TEST/EQL_LIST_T.HVM

```
// Test: structural equality of lists (equal)
@main = [1,2,3] === [1,2,3]
//1
```

TEST/EQL_MIXED_F.HVM

```
// Test: structural equality of different types (num vs ctr)
@main = 42 === #Box{42}
//0
```

TEST/EQL_NAT_F.HVM

```
// Test: structural equality of nat sugar (different)
@main = 3n === 4n
//0
```

TEST/EQL_NAT_T.HVM

```
// Test: structural equality of nat sugar
@main = 3n === 3n
//1
```

TEST/EQL_NESTED.HVM

```
// Test: structural equality of nested structures
@main = #Pair{#Pair{1,2}, 3} === #Pair{#Pair{1,2}, 3}
//1
```

TEST/EQL_NUM_F.HVM

```
// Test: structural equality of numbers (not equal)
@main = 42 === 7
//0
```

TEST/EQL_NUM_T.HVM

```
// Test: structural equality of numbers (equal)
@main = 42 === 42
//1
```

TEST/EQL_REF.HVM

```
// Test: structural equality via reference
@id = λx.x
@main = @id === @id
//1
```

TEST/EQL_STR_F.HVM

```
// Test: structural equality of strings (different)
@main = "hello" === "world"
//0
```

TEST/EQL_STR_T.HVM

```
// Test: structural equality of strings
@main = "hello" === "hello"
//1
```

TEST/EQL_SUP.HVM

```
// Test: structural equality with superposition (eql-sup-l, collapse flattens)
@main = &a{1,2} === 1
//1
//0
```

TEST/FACTORIAL.HVM

```
#include "_book_.hvm"

@one = #S{#Z{}}
@two = #S{#S{#Z{}}}

@add = λ{ #Z: λb.b ; #S: λp.λb.#S{@add(p, b)}; }
@mul = λ{ #Z: λb.#Z{} ; #S: λp.λb. !B&A=b ; @add(B₀, @mul(p, B₁)); }
@fac = λ{ #Z: #S{#Z{}} ; #S: λp. !P&A=p ; @mul(#S{P₀}, @fac(P₁)); }

@main = @fac(@add(@two, @one))
//#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}
```

TEST/FIB_SMALL.HVM

```
@fib = λ{
  0: 0;
  1: 1;
  _: λ&n. @fib(n - 1) + @fib(n - 2)
}

@main = @fib(34)
//5702887
```

TEST/FIB.HVM

```
#include "_book_.hvm"
@fib = λ{#Z:#Z{}; λ{#S:λ{#Z:#S{#Z{}}; λ{#S:λn.!F&=@fib; !N&=n; @add(F₀(N₀), F₁(#S{N₁})); &{}}}; &{}}}

@main = @fib(#S{#S{#S{#S{#S{#Z{}}}}}})
//#S{#S{#S{#S{#S{#Z{}}}}}}
```

TEST/FIBO.HVM

```
// @add = λ{0: λb.b ; 1+: λa.λb. 1+(@add a b)}
// @fib = λ{0:0; 1+:λ{0: 1; 1+:λp. ! P & L = p ; (@add (@fib (1+P₀)) (@fib P₁))}}"

@add = λ{ #Z: λb.b ; #S: λp.λb. #S{@add(p, b)}; }
@fib = λ{ #Z: #Z{} ; #S: λ{ #Z: #S{#Z{}} ; #S: λp. !P&L=p ; @add(@fib(#S{P₀}), @fib(P₁))}}

@two = #S{#S{#Z{}}}
@three = #S{#S{#S{#Z{}}}}

@main = @fib(@add(@three, @three))
//#S{#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}}
```

TEST/FORK_DYNAMIC.HVM

```
// Fork notation with dynamic label: &(L)λx,y{A;B}
@main = (λ&L.&(L)λx,y{(x+y);(x*y)})(1)(&(1){2,100},&(1){3,100})
//5
//10000
```

TEST/FORK_STATIC.HVM

```
// Fork notation with static label: &Lλx,y{A;B}
@main = (&Lλx,y{(x+y);(x*y)})(&L{2,100},&L{3,100})
//5
//10000
```

TEST/FUSE_INC.HVM

```
// Fuse_Inc: A minimal example of Optimal Evaluation
// Computes ~1m calls to inc(x) in 6876 interactions

@O = λx. λo. λi. λe. o(x)
@I = λx. λo. λi. λe. i(x)
@E =     λo. λi. λe. e

@inc = λxs.
  λo. λi. λe. xs(i, λp.o(@inc(p)), e)

@rep = λ{
  #E: λf. λx. x
  #O: λp. λf. λx.
    ! F &A = f;
    @rep(p, λk.F₀(F₁(k)), x)
  #I: λp. λf. λx.
    ! G &A = f;
    ! F &A = G₀;
    @rep(p, λk.F₀(F₁(k)), G₁(x))
}

@sbin_to_bin = λxs.
  xs( λp.#O{@sbin_to_bin(p)}
    , λp.#I{@sbin_to_bin(p)}
    , #E{})

@sbin_zero_32 =
  @rep(#O{#O{#O{#O{#O{#I{#E{}}}}}}}, λx.@O(x), @E)

@bin_1048576 =
  #O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#I{#E{}}}}}}}}}}}}}}}}}}}}}}

@main =
  @sbin_to_bin(@rep(@bin_1048576, @inc, @sbin_zero_32))
//#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#I{#O{#O{#O{#O{#O{#O{#O{#O{#O{#O{#E{}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}
```

TEST/GEN_MUL4.HVM

```
@Y    = λ&f. f(@Y(f))
@when = λ{[]: λx.x; <>: λ{0: λt. &{}; 1: λt,x. @when(t, x)}}

@muln = λ&L,&K. λF. λ{
  0n: 0n
  1n+: λp. @expr(L, K, F, p)
}

@expr = λ&L. λ{
  0n: λf. λx. x;
  1n+: λ&K. λf. λx.
    ! f &(L) = f
    ! x &(L) = x
    &(L){f₀(x₀), 1n+@expr(L+1,K,f₁,x₁)}
}

@main =
  ! &F = λF. @muln(1, 100000n, F)
  ! e0 = @Y(F)(3n) === 12n
  ! e1 = @Y(F)(4n) === 16n
  @when([e0,e1], [F])
//[λa.λ{0n:0n;1n+:λb.4n+a(b)}]
```

TEST/LAM_DUP_AUTO.HVM

```
@main = (λx&.[x₀,x₁])(7)
//[7,7]
```

TEST/LAM_DUP_DYN.HVM

```
@main = (λx&(0).[x₀,x₁])(&(0){1,2})
//[1,2]
```

TEST/LAM_DUP_STATIC.HVM

```
@main = (λx&L.[x₀,x₁])(7)
//[7,7]
```

TEST/LAM_DUP_SUP.HVM

```
@main = (λx&L.[x₀,x₁])(&L{1,2})
//[1,2]
```

TEST/LAMBDA_ENV0.HVM

```
// lambda_env0.hvm
// ---------------
// Baseline standalone environment-based lambda-calculus interpreter.
//
// Terms:
// - #Var{idx}
// - #Lam{body}
// - #App{fun, arg}
//
// Values:
// - #VLam{env, body}
// - #VNeu{neu}
//
// Neutrals:
// - #NVar{lev}
// - #NApp{fun, arg}

// Subst
// -----

@env_get = λ{
  []: λidx.
    #VNeu{#NVar{idx + 1000}};
  <>: λh. λt. λidx.
    @env_get_0(idx, h, t)
}

@env_get_0 = λ{
  0: λh. λt.
    h
  λn. λh. λt.
    @env_get(t, n - 1)
}

@subst = λenv. λbod. λarg.
  @eval_0(arg <> env, bod)

// Eval
// ----

@eval = λtrm.
  @eval_0([], trm)

@eval_0 = λ&env. λ{
  #Var: λidx.
    @env_get(env, idx)
  #Lam: λbod.
    #VLam{env, bod}
  #App: λfun. λarg.
    ! e &D = env
    @wnf_app(@eval_0(e₀, fun), @eval_0(e₁, arg))
}

// Quote
// -----

@quote_wnf = λ{
  #VLam: λenv. λbod. λ&lvl.
    ! l &D = lvl
    #Lam{@quote_wnf_body(env, 1, l₀ + 1, bod)}
  #VNeu: λneu. λlvl.
    @quote_wnf_neu(neu, lvl)
}

@quote_wnf_body = λ&env. λ&cut. λ&lvl. λ{
  #Var: λidx.
    ! i &D = idx
    ! c &D = cut
    @quote_wnf_var(i₀ < c₀, env, c₁, lvl, i₁)
  #Lam: λbod.
    #Lam{@quote_wnf_body(env, cut + 1, lvl + 1, bod)}
  #App: λfun. λarg.
    ! e &D = env
    ! c &D = cut
    ! l &D = lvl
    #App{@quote_wnf_body(e₀, c₀, l₀, fun), @quote_wnf_body(e₁, c₁, l₁, arg)}
}

@quote_wnf_var = λ{
```

```
  1: λ&env. λ&cut. λ&lvl. λidx.
    #Var{idx}
  0: λ&env. λ&cut. λ&lvl. λidx.
    @quote_wnf(@env_get(env, idx - cut), lvl)
}

@quote_wnf_neu = λ{
  #NVar: λlev. λlvl.
    #Var{lvl - lev - 1}
  #NApp: λfun. λarg. λ&lvl.
    ! l &D = lvl
    #App{@quote_wnf_neu(fun, l₀), @quote_wnf(arg, l₁)}
}

@quote = λ{
  #VLam: λenv. λbod. λ&lvl.
    ! l &D = lvl
    ! x = #VNeu{#NVar{l₀}}
    #Lam{@quote(@subst(env, bod, x), l₁ + 1)}
  #VNeu: λneu. λlvl.
    @quote_neu(neu, lvl)
}

@quote_neu = λ{
  #NVar: λlev. λlvl.
    #Var{lvl - lev - 1}
  #NApp: λfun. λarg. λ&lvl.
    ! l &D = lvl
    #App{@quote_neu(fun, l₀), @quote(arg, l₁)}
}

// WNF
// ---

@wnf = λtrm.
  @quote_wnf(@eval(trm), 0)

@wnf_app = λ{
  #VLam: λenv. λbod. λarg.
    @subst(env, bod, arg)
  #VNeu: λneu. λarg.
    #VNeu{#NApp{neu, arg}}
}

// SNF
// ---

@snf = λtrm.
  @quote(@eval(trm), 0)

// Terms
// -----

@c2  = #Lam{#Lam{#App{#Var{1}, #App{#Var{1}, #Var{0}}}}}
@c8  = #Lam{#Lam{#App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #Var{0}}}}}}}}}}
@t   = #Lam{#Lam{#Var{1}}}
@f   = #Lam{#Lam{#Var{0}}}
@not = #Lam{#App{#App{#Var{0}, @f}, @t}}
@xor = #Lam{#Lam{#App{#App{#Var{1}, #App{@not, #Var{0}}}, #Var{0}}}}
@tog = #Lam{#App{#App{@xor, #Var{0}}, @t}}

// Main
// ----

@main = @snf(#App{#App{#App{@c8, @c2}, @tog}, @t})
//#Lam{#Lam{#Var{1}}}
```

TEST/LAMBDA_ENV1.HVM

```
// lambda_env1.hvm
// ---------------
// Tuple-threaded standalone environment-based lambda-calculus interpreter.
//
// Terms:
// - #Var{idx}
// - #Lam{body}
// - #App{fun, arg}
//
// Values:
// - #VLam{env, body}
// - #VNeu{neu}
//
// Neutrals:
// - #NVar{lev}
// - #NApp{fun, arg}
//
// Why this is faster than lambda_env0.hvm:
//
// On HVM, duplications (ex: variables that use &) are very costly, in special
// when the cloned structure is large. To avoid that, a trick we often use is
// to pass that variable linearly, using a tuple (&T{x,y}) and tuple projections
// (! x_ret &T = ...), and avoiding cross-branch dups by moving the
// pattern-matched value to the beginning of the argument list.
//
// Bad:
//
//   @foo = λ&big. λ{
//     #Foo: λx. λy. @foo(x,big) + @foo(y,big)
//     #Bar: λx. λy. @foo(x,big) + @foo(y,big)
//     ...
//   }
//
// Here, big is duplicated for two reasons:
// - to pass it to both branches
// - to pass it to both recursive calls
//
// Better:
//
//   @foo = λ{
//     #Foo: λx. λy. λbig.
//       ! x_ret &T = @foo(x,big);
//       ! y_ret &T = @foo(y,x_ret₀);
//       &T{y_ret₀, x_ret₁ + y_ret₁}
//     #Bar: λx. λy. λbig. ...
//   }
//
// Here, we avoid all dups of big:
// - by moving it to the last argument, so it gets its own λ on each branch
// - by using tuples to pass leftover big's from recursive calls in a chain-like fashion

// Subst
// -----

@env_nil = λidx.
  #VNeu{#NVar{idx + 1000}}

@env_get = λidx. λ&env.
  ! e &D = env
  &T{e₀, e₁(idx)}

@env_ext = λenv. λarg. λ{
  0: arg
  λn.
    env(n - 1)
}

@subst = λenv. λbod. λarg.
  ! ret &T = @eval_0(bod, @env_ext(env, arg))
  ret₁

// Eval
// ----

@eval = λtrm.
  ! ret &T = @eval_0(trm, @env_nil)
  ret₁

@eval_0 = λ{
```

```
  #Var: λidx. λenv.
    @env_get(idx, env)
  #Lam: λbod. λ&env.
    ! e &D = env
    &T{e₀, #VLam{e₁, bod}}
  #App: λfun. λarg. λenv.
    ! fun_ret &T = @eval_0(fun, env)
    ! arg_ret &T = @eval_0(arg, fun_ret₀)
    @wnf_app(fun_ret₁, arg_ret₁, arg_ret₀)
}

// Quote
// -----

@quote_wnf = λ{
  #VLam: λenv. λbod. λ&lvl.
    ! l &D = lvl
    ! ret &T = @quote_wnf_body(bod, env, 1, l₀ + 1)
    ret₁
  #VNeu: λneu. λlvl.
    @quote_wnf_neu(neu, lvl)
}

@quote_wnf_body = λ{
  #Var: λ&idx. λenv. λ&cut. λlvl.
    ! i &D = idx
    ! c &D = cut
    @quote_wnf_var(i₀ < c₀, i₁, env, c₁, lvl)
  #Lam: λbod. λenv. λcut. λ&lvl.
    ! l &D = lvl
    ! ret &T = @quote_wnf_body(bod, env, cut + 1, l₀ + 1)
    &T{ret₀, #Lam{ret₁}}
  #App: λfun. λarg. λenv. λcut. λ&lvl.
    ! c &D = cut
    ! l &D = lvl
    ! fun_ret &T = @quote_wnf_body(fun, env, c₀, l₀)
    ! arg_ret &T = @quote_wnf_body(arg, fun_ret₀, c₁, l₁)
    &T{arg_ret₀, #App{fun_ret₁, arg_ret₁}}
}

@quote_wnf_var = λ{
  1: λidx. λenv. λcut. λlvl.
    &T{env, #Var{idx}}
  0: λidx. λenv. λcut. λlvl.
    ! ret &T = @env_get(idx - cut, env)
    &T{ret₀, @quote_wnf(ret₁, lvl)}
}

@quote_wnf_neu = λ{
  #NVar: λlev. λlvl.
    #Var{lvl - lev - 1}
  #NApp: λfun. λarg. λ&lvl.
    ! l &D = lvl
    #App{@quote_wnf_neu(fun, l₀), @quote_wnf(arg, l₁)}
}

@quote = λ{
  #VLam: λenv. λbod. λ&lvl.
    ! l &D = lvl
    ! x = #VNeu{#NVar{l₀}}
    #Lam{@quote(@subst(env, bod, x), l₁ + 1)}
  #VNeu: λneu. λlvl.
    @quote_neu(neu, lvl)
}

@quote_neu = λ{
  #NVar: λlev. λlvl.
    #Var{lvl - lev - 1}
  #NApp: λfun. λarg. λ&lvl.
    ! l &D = lvl
    #App{@quote_neu(fun, l₀), @quote(arg, l₁)}
}

// WNF
// ---

@wnf = λtrm.
  @quote_wnf(@eval(trm), 0)

@wnf_app = λ{
  #VLam: λenv. λbod. λarg. λout.
    &T{out, @subst(env, bod, arg)}
```

```
  #VNeu: λneu. λarg. λout.
    &T{out, #VNeu{#NApp{neu, arg}}}
}

// SNF
// ---

@snf = λtrm.
  @quote(@eval(trm), 0)

// Terms
// -----

@c2  = #Lam{#Lam{#App{#Var{1}, #App{#Var{1}, #Var{0}}}}}
@c8  = #Lam{#Lam{#App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #Var{0}}}}}}}}}}
@t   = #Lam{#Lam{#Var{1}}}
@f   = #Lam{#Lam{#Var{0}}}
@not = #Lam{#App{#App{#Var{0}, @f}, @t}}
@xor = #Lam{#Lam{#App{#App{#Var{1}, #App{@not, #Var{0}}}, #Var{0}}}}
@tog = #Lam{#App{#App{@xor, #Var{0}}, @t}}

// Main
// ----

@main = @snf(#App{#App{#App{@c8, @c2}, @tog}, @t})
//#Lam{#Lam{#Var{1}}}
```

TEST/LAMBDA_EVAL.HVM

```
// lambda_eval.hvm
// ---------------
// Native HVM lambda benchmark.

// Terms
// -----

@c2  = λ&f. λx. f(f(x))
@c8  = λ&f. λx. f(f(f(f(f(f(f(f(x)))))))
@t   = λt. λf. t
@f   = λt. λf. f
@not = λb. b(@f, @t)
@xor = λa. λ&b. a(@not(b), b)
@tog = λb. @xor(b, @t)

// Main
// ----

@main = @c8(@c2, @tog, @t)
//λa.λb.a
```

TEST/LAMBDA_HOAS.HVM

```
// lambda_hoas.hvm
// ---------------
// Minimal standalone HOAS lambda-calculus interpreter.
//
// Terms:
// - #Lam{λ&x. body}
// - #App{fun, arg}

// Subst
// -----

@subst = λbod. λarg.
  bod(arg)

// WNF
// ---

@wnf = λ{
  #App: λfun. λarg.
    @wnf_app(@wnf(fun), arg)
  λx.
    x
}

@wnf_app = λ{
  #Lam: λbod. λarg.
    @wnf(@subst(bod, @wnf(arg)))
  λ&fun. λarg.
    #App{fun, arg}
}

// SNF
// ---

@snf = λtrm.
  @snf_0(@wnf(trm))

@snf_0 = λ{
  #Lam: λbod.
    #Lam{λ&x. @snf(bod(#Sub{x}))}
  #App: λfun. λarg.
    #App{@snf(fun), @snf(arg)}
  #Sub: λx.
    x
  λx.
    x
}

// Terms
// -----

@c2  = #Lam{λ&f. #Lam{λ&x. #App{f, #App{f, x}}}}
@c8  = #Lam{λ&f. #Lam{λ&x. #App{f, #App{f, #App{f, #App{f, #App{f, #App{f, #App{f, #App{f, x}}}}}}}}}}
@t   = #Lam{λ&t. #Lam{λ&f. t}}
@f   = #Lam{λ&t. #Lam{λ&f. f}}
@not = #Lam{λ&b. #App{#App{b, @f}, @t}}
@xor = #Lam{λ&a. #Lam{λ&b. #App{#App{a, #App{@not, b}}, b}}}
@tog = #Lam{λ&b. #App{#App{@xor, b}, @t}}

// Main
// ----

@main = @snf(#App{#App{#App{@c8, @c2}, @tog}, @t})
//#Lam{λa.#Lam{λb.a}}
```

TEST/LAMBDA_SUBS.HVM

```
// lambda_subs.hvm
// ---------------
// Minimal standalone explicit-substitution lambda-calculus interpreter.
//
// Terms:
// - #Var{idx}
// - #Lam{body}
// - #App{fun, arg}

// Subst
// -----

@lift = λcut. λtrm.
  @lift_0(cut, trm)

@lift_0 = λ&cut. λ{
  #Var: λ&idx.
    ! c &D = cut
    ! i &D = idx
    @lift_var(i₀ < c₀, i₁, c₁)
  #Lam: λbod.
    #Lam{@lift_0(cut + 1, bod)}
  #App: λfun. λarg.
    ! c &D = cut
    #App{@lift_0(c₀, fun), @lift_0(c₁, arg)}
}

@lift_var = λ{
  1: λidx. λcut.
    #Var{idx}
  0: λidx. λcut.
    #Var{idx + 1}
}

@subst = λval. λbod.
  @subst_0(0, val, bod)

@subst_0 = λ&dep. λ&val. λ{
  #Var: λ&idx.
    ! d &D = dep
    ! i &D = idx
    ! x &D = i₀
    @subst_var(x₀ < d₀, x₁ > d₁, i₁, val)
  #Lam: λbod.
    #Lam{@subst_0(dep + 1, @lift(0, val), bod)}
  #App: λfun. λarg.
    ! d &D = dep
    ! v &D = val
    #App{@subst_0(d₀, v₀, fun), @subst_0(d₁, v₁, arg)}
}

@subst_var = λ{
  1: λgt. λidx. λval.
    #Var{idx}
  0: λgt. λidx. λval.
    @subst_var_0(gt, idx, val)
}

@subst_var_0 = λ{
  1: λidx. λval.
    #Var{idx - 1}
  0: λidx. λval.
    val
}

// Eval
// ----

@eval = λtrm.
  trm

// Quote
// -----

@quote_wnf = λtrm. λlvl.
  trm

@quote = λtrm. λlvl.
  trm
```

```
// WNF
// ---

@wnf = λtrm.
  @wnf_0(@eval(trm))

@wnf_0 = λ{
  #App: λfun. λarg.
    @wnf_app(@wnf(fun), arg)
  λtrm.
    @eval(trm)
}

@wnf_app = λ{
  #Lam: λbod. λarg.
    @wnf(@subst(@wnf(arg), bod))
  λ&fun. λarg.
    #App{fun, arg}
}

// SNF
// ---

@snf = λtrm.
  @snf_0(@quote(@wnf(trm), 0), 0)

@snf_0 = λ{
  #Var: λidx. λlvl.
    #Var{idx}
  #Lam: λbod. λ&lvl.
    ! l &D = lvl
    #Lam{@snf_0(@quote(@wnf(bod), l₀ + 1), l₁ + 1)}
  #App: λfun. λarg. λ&lvl.
    ! l &D = lvl
    ! x &D = l₀
    ! y &D = l₁
    #App{@snf_0(@quote(@wnf(fun), x₀), x₁), @snf_0(@quote(@wnf(arg), y₀), y₁)}
}

// Terms
// -----

@c2  = #Lam{#Lam{#App{#Var{1}, #App{#Var{1}, #Var{0}}}}}
@c8  = #Lam{#Lam{#App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #App{#Var{1}, #Var{0}}}}}}}}}}
@t   = #Lam{#Lam{#Var{1}}}
@f   = #Lam{#Lam{#Var{0}}}
@not = #Lam{#App{#App{#Var{0}, @f}, @t}}
@xor = #Lam{#Lam{#App{#App{#Var{1}, #App{@not, #Var{0}}}, #Var{0}}}}
@tog = #Lam{#App{#App{@xor, #Var{0}}, @t}}

// Main
// ----

@main = @snf(#App{#App{#App{@c8, @c2}, @tog}, @t})
//#Lam{#Lam{#Var{1}}}
```

TEST/LET.HVM

```
// Let sugar: !x = val; body  →  (λx.body)(val)
// Strict let: !!x = val; body  →  (λ{λx.body(x)})(val)
// Cloned let: ! &x = val allows multiple uses of x
@lazy = ! &x = 1 + 2; x * x
@strict = !! &x = 1 + 2; x * x
@nested = !a = 1; !b = 2; !c = 3; a + (b + c)
@main = [@lazy, @strict, @nested]
//[9,9,6]
```

TEST/LONG_NAME_CTOR_NO_COLLISION.HVM

```
@main = [#mse{} === #c_mse{}, #mse{}, #c_mse{}]
//[0,#mse{},#c_mse{}]
```

TEST/LONG_NAME_DEF.HVM

```
@my_very_long_function_name = λx. [1,x]
@other_very_long_function_name = λx. [2,x]
@main = [@my_very_long_function_name(42), @other_very_long_function_name(42)]
//[[1,42],[2,42]]
```

TEST/LONG_NAME_LABEL_NO_COLLISION.HVM

```
@main = !x&mse = &c_mse{1,2}; [x₀,x₁]
//PARSE_ERROR
```

TEST/LONG_NAME_MATCH_NO_COLLISION.HVM

```
@main = λ{ #mse: #t{}; _: λx.#f{} }(#c_mse{})
//#f{}
```

TEST/LONG_NAME_NO_COLLISION.HVM

```
@my_very_long_ref = λx. x
@main = @other_undefined_ref
//@other_undefined_ref
```

TEST/LONG_NAME_UNDEF.HVM

```
@main = @this_function_is_not_defined
//@this_function_is_not_defined
```

TEST/MAP.HVM

```
#include "_book_.hvm"

@inc = λx.#S{x}
@map = λf.λ{#Nil: #Nil{} ; #Cons: λh.λt. !F&L=f ; #Cons{F₀(h), @map(F₁, t)} }

@main = @map(@inc, #Cons{#Z{}, #Cons{#S{#Z{}}, #Nil{}}})
//#Cons{#S{#Z{}},#Cons{#S{#S{#Z{}}},#Nil{}}}
```

TEST/MATCH_0.HVM

```
// Chained SWI and MAT with defaults
@swi = λ{0:10; 1:20; 2:30; λ&x.x * x}
@mat = λ{#A:1; #B:λx.x; #C:λx.λy.x + y; λz.0}
@main = [
  @swi(0),
  @swi(2),
  @swi(5),
  @mat(#A{}),
  @mat(#B{7}),
  @mat(#C{3,4}),
  @mat(#D{})]
//[10,30,25,1,7,7,0]
```

TEST/MATCH_1.HVM

```
// Nat and List sugar patterns
@len = λ{[]:0; <>:λh.λt.1 + @len(t)}
@sum = λ{0n:0; 1n+:λp.1 + @sum(p)}
@main = [
  @len([]),
  @len([1,2,3]),
  @sum(0n),
  @sum(5n)]
//[0,3,0,5]
```

TEST/MATCH_2.HVM

```
// Nested matches and fallthrough
@f = λ{0:λ{0:0; 1:1; λy.2}; 1:λ{0:3; 1:4; λy.5}; λx.λy.x + y}
@g = λ{#A:λ{#X:1; #Y:2; λx.0}; #B:λ{#X:3; #Y:4; λx.0}; λa.λb.9}
@main = [
  @f(0,0),
  @f(0,1),
  @f(1,0),
  @f(5,3),
  @g(#A{},#X{}),
  @g(#B{},#Y{}),
  @g(#C{},#Z{})]
//[0,1,3,8,1,4,9]
```

TEST/MATCH_3.HVM

```
// Mixed: MAT on list structure with nesting
@classify = λ{
  []: 0;
  <>: λh.λt.λ{
    []: 1;
    <>: λh2.λt2.λ{[]: 2; λx.3}(t2)
  }(t)
}
@main = [
  @classify([]),
  @classify([1]),
  @classify([1,2]),
  @classify([1,2,3])]
//[0,1,2,3]
```

TEST/MATCH_4.HVM

```
// SWI with sparse values and MAT extracting fields
@sparse = λ{0:1; 10:2; 100:3; 1000:4; λx.0}
@get = λ{#None:0; λsome.λ{#Some:λx.x}(some)}
@pair = λ{#P:λa.λb.a + b; λx.0}
@main = [
  @sparse(0),
  @sparse(100),
  @sparse(50),
  @get(#None{}),
  @get(#Some{42}),
  @pair(#P{3,7})]
//[1,3,0,0,42,10]
```

TEST/MATCH_5.HVM

```
// Default case with _: syntax
@swi = λ{0:10; 1:20; _:λ&x.x * x}
@mat = λ{#A:1; #B:2; _:λx.0}
@lst = λ{[]:0; <>:λh.λt.h + @lst(t); _:0}
@main = [@swi(0), @swi(1), @swi(5), @mat(#A{}), @mat(#C{}), @lst([1,2,3])]
//[10,20,25,1,0,6]
```

TEST/MATCH_CHR_ESCAPE.HVM

```
@main = (λ{
  #CHR: λc. (λ{
    '\n': 1;
    0
  })(c);
})('\n')
//1
```

TEST/MATCH_CHR_SWITCH.HVM

```
@main = (λ{
  #CHR: λc. (λ{
    'λ': 1;
    0
  })(c);
})('λ')
//1
```

TEST/MATCH_CTR_MISS.HVM

```
@pick = λ{#A: 10; λx. 20}
@main = @pick(#B{})
//20
```

TEST/MATCH_CTR_PAIR.HVM

```
@sum_pair = λ{#P: λa. λb. a + b; λx.0}
@main = @sum_pair(#P{20,22})
//42
```

TEST/MATCH_RECURSIVE_ARG.HVM

```
@mat_self = λ{
  0n: #Z{};
  1n+: λd. (λ{ #Z: #Z{}; #S: λp. @mat_self(p)})(@mat_self(d));
}

@main = @mat_self(100000n)
//#Z{}
```

TEST/MATCH_U32.HVM

```
@succ_or_100 = λ{0:100; λx. x + 1}
@main = @succ_or_100(5)
//6
```

TEST/MUL2_4.HVM

```
#include "_book_.hvm"
@main = @mul2(#S{#S{#S{#S{#Z{}}}}})
//#S{#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}}
```

TEST/MULTIARG_CLONED.HVM

```
// λ&x,&y,&z.f - all cloned (auto-dup)
@main = (λ&x,&y,&z.[x,x,y,y,z,z])(1,2,3)
//[1,1,2,2,3,3]
```

TEST/MULTIARG_INLINE_DUP.HVM

```
// λx&L,y&M.f - each arg has inline dup
@main = (λx&L,y&M.[x₀,x₁,y₀,y₁])(1,2)
//[1,1,2,2]
```

TEST/MULTIARG_MIXED.HVM

```
// λ&x,y&L,z.f - mixed: cloned, static dup, simple
@main = (λ&x,y&L,z.[x,x,y₀,y₁,z])(1,2,3)
//[1,1,2,2,3]
```

TEST/MULTIARG_SIMPLE.HVM

```
// λx,y,z.f → λx.λy.λz.f
@main = (λx,y,z.[x,y,z])(1,2,3)
//[1,2,3]
```

TEST/NAT_EQL.HVM

```
@eql = λ{
  #ZER: λ{
    #ZER: 1
    #SUC: λb. 0
  }
  #SUC: λa. λ{
    #ZER: 0
    #SUC: λb. @eql(a, b)
  }
}

@main = @eql(7n, 7n)
//1
```

TEST/PAIR_SWAP.HVM

```
#include "_book_.hvm"

// Pair = { #P{a, b} }

@swap = λ{ #P: λa.λb. #P{b, a} }

@main = @swap(#P{#Z{}, #S{#Z{}}})
//#P{#S{#Z{}},#Z{}}
```

TEST/PARSER.HVM

```
@def = λa.a

@ref = @def(λa.a)

@stuck = λa.^a

@dry = ^(λa.a λb.b)

@era = &{}

@sup = &L{λa.a, λb.b}

@dyn_sup = λL.&(L){λa.a, λb.b}

@dup = λa.!A&L=a ; 1

@co0 = λa.!A&L=a ; A₀

@co1 = λa.!A&L=a ; A₁

@ctr = #Ctor{}

@nest_ctr = #Ctor{#Another{}}

@match_ctr = λ{
  #Z: #Z{}    ;
  #S: λs.#Z{} ;
}

@num_mat = λ{
  0: 0 ;
  1: 1 ;
  λx.x
}

@use = λ{λx.x}

@era2 = λ{}

@lam = λx.x

@paren = (λx.x) // NOT an APP

@app = λx.x(2)

@num = 102

@plus = 10 + 10
@sub  = 10 - 10
@mul  = 10 * 10
@mod  = 10 % 10
@and  = 1 && 1
@or   = 1 || 1
@xor  = 1 ^ 1
@lshf = 1 << 1
@rshf = 1 >> 1
@lt   = 2 < 1
@lte  = 2 <= 1
@gt   = 1 > 1
@gte  = 2 >= 1
@eq   = 1 == 1
@neq  = 1 != 1

@strct_eq = λx.x === λa.a

@unsc = ! a = λ b ; λa.2

// sugars

@autodup_lam = λ&x.x + x

@dup_lam = λx&L.x₀ + x₁

@dyn_dup_lam = λL.λx&(L).x₀ + x₁

@dyn_auto_dup_lam = λ&x&.x₀ * x₁

@multi_arg_lam = λx,y,z.y
```

```
@multi_arg_autodup_lam = λ&x,&y,z.y * y + x + x

@multi_arg_inl_dup = λx&L,y&D.x₀ + x₁ + y₀ + y₁

@list_mat = λ{
  []: #Z{} ;
  <>: λh.λt. #Z{} ;
}

@def_case = λ{
  0n: #Z{} ;
  λsuc.suc
}

@nat_mat = λ{
  0n: #Z{} ;
  1n+: λs.#S{s} ;
}

@frk = &Lλx{x ; x}

@frk_dyn = λ&L.&(L) λx,y {x ; y}

@frk_brnch = &L λx,y {
  &₀: x ;
  &₁: y ;
}

@frk_brnch_dyn = λ&L. &(L) λx, y {
  &₀: x ;
  &₁: y ;
}

@let =
  !x = 1 ;
  x

@stri_let =
  !!x = λx.x(1)
  x

@dup_let = λx.
  !&y = x * 2 ;
  y + y

@stri_dup_let = λx.
  !!&y = x * 2 ;
  y * y

@dup_auto_lab = λx.
  !X& = x ;
  X₀ + X₁

@auto_dup_auto_lab = λx.
  !&X& = x ;
  X₀ + X₁

@auto_dup_static_lab = λx.
  !&X&L = x ;
  X₀

@auto_dup_dyn_lab = λL.λx.
  !&X&(L) = x ;
  X₁

@list_lit = [1, 2, 3]

@list_cons = 1 <> 2 <> []

@string_lit = "hello"

@nat_lit = 4n

@suc_lit = 1n+ 3n

@main = 1
//1
```

TEST/PARTIAL_FALLBACK.HVM

```
@foo = λx. λy. λz. y
@main = !F&=@foo(1); F₀
//λa.λb.a
```

TEST/PRIM_ADD.HVM

```
@main = 1 + 2 + 3 + 4
//10
```

TEST/PRIM_AND.HVM

```
@main = 12 && 10
//8
```

TEST/PRIM_DIV.HVM

```
@main = 20 / 4
//5
```

TEST/PRIM_DUP.HVM

```
@main = λx. !X&(5)=x; !Y&(4)=X₀; λy. y(X₁,Y₀,Y₁)
//λa.λb.b(a,a,a)
```

TEST/PRIM_EQ_F.HVM

```
@main = 42 == 43
//0
```

TEST/PRIM_EQ_T.HVM

```
@main = (2 + 3) == (1 * 5)
//1
```

TEST/PRIM_GE_F.HVM

```
@main = 5 >= 10
//0
```

TEST/PRIM_GE_T.HVM

```
@main = 5 >= 5
//1
```

TEST/PRIM_GT_F.HVM

```
@main = 5 > 10
//0
```

TEST/PRIM_GT_T.HVM

```
@main = 10 > 5
//1
```

TEST/PRIM_LE_F.HVM

```
@main = 10 <= 5
//0
```

TEST/PRIM_LE_T.HVM

```
@main = 5 <= 5
//1
```

TEST/PRIM_LOG_ARITY.HVM

```
@main = %log
//PARSE_ERROR
```

TEST/PRIM_LOG_BASIC.HVM

```
@main = %log("hi")
//!hi
//![]
```

TEST/PRIM_LOG_NEWLINE.HVM

```
@main = %log("foo\nbar")
//!foo
//!bar
//![]
```

TEST/PRIM_LOG_REVERSE.HVM

```
@rev = λxs. @rev_go(xs, [])
@rev_go = λxs.λ&acc. λ{
  []: acc;
  <>: λh.λt. @rev_go(t, h <> acc)
}(xs)

@main = %log(@rev("dlrow\nolleh"))
//!hello
//!world
//![]
```

TEST/PRIM_LSH.HVM

```
@main = 3 << 4
//48
```

TEST/PRIM_LT_F.HVM

```
@main = 10 < 5
//0
```

TEST/PRIM_LT_T.HVM

```
@main = (2 * 3) < (3 * 3)
//1
```

TEST/PRIM_MOD.HVM

```
@main = 17 % 5
//2
```

TEST/PRIM_MUL.HVM

```
@main = 2 * 3 + 4 * 5
//26
```

TEST/PRIM_NE_F.HVM

```
@main = 42 != 42
//0
```

TEST/PRIM_NE_T.HVM

```
@main = 42 != 43
//1
```

TEST/PRIM_NOT.HVM

```
// NOT is implemented as 0 ~ x (bitwise complement)
@main = (0 ~ 0) && 255
//255
```

TEST/PRIM_OR.HVM

```
@main = 12 || 10
//14
```

TEST/PRIM_RSH.HVM

```
@main = 48 >> 4
//3
```

TEST/PRIM_SP0.HVM

```
@main = ((1 * 16293) + 1) && 65535
//16294
```

TEST/PRIM_SP1.HVM

```
@main = ((1 * 32677) + 3) && 65535
//32680
```

TEST/PRIM_SUB.HVM

```
@main = 100 - 20 - 50 + 40
//70
```

TEST/PRIM_SUP.HVM

```
@main = &(1){&(2){1, 2}, 3}
//3
//1
//2
```

TEST/PRIM_XOR.HVM

```
@main = 12 ^ 10
//6
```

TEST/PRINT_CHR_ESCAPE.HVM

```
@main = #CHR{10}
//'\n'
```

TEST/PRINT_MAT_DEFAULT.HVM

```
// Test that MAT with default prints correctly
@main = λ{#A:1;#B:2;99}
//λ{#A:1;#B:2;99}
```

TEST/PRINT_MAT_LST.HVM

```
// Test that list sugar in MAT prints correctly
@main = λ{[]:1;<>:2}
//λ{[]:1;<>:2}
```

TEST/PRINT_MAT_NAT.HVM

```
// Test that nat sugar in MAT prints correctly
@main = λ{0n:1;1n+:2}
//λ{0n:1;1n+:2}
```

TEST/PRINT_MAT.HVM

```
// Test that MAT chains print in sugared form
@main = λ{#A:1;#B:2;#C:3}
//λ{#A:1;#B:2;#C:3}
```

TEST/PRINT_STR_ESCAPE.HVM

```
@main = [#CHR{64}, #CHR{64}, #CHR{10}, #CHR{64}]
//"@@\n@"
```

TEST/PRINT_SWI.HVM

```
// Test that SWI chains print in sugared form
@main = λ{0:10;1:20;2:30}
//λ{0:10;1:20;2:30}
```

TEST/REGRESSION_AUTODUP_3USES_RECURSIVE.HVM

```
// Regression test: auto-dup with 3 uses in recursive context
// Bug: DP1 in dup chain had wrong De Bruijn index (1 instead of 0)
// This caused crashes or wrong results with 3+ uses of auto-dup var

@app = λ{
  []: [];
  <>: λx. λxs. λ&f. f(x) <> @app(xs, f);
}

@lam = λ&L. λ{
  #All: λ{
    #Boo: λB. λS. λR.
      λ&x. @lam(L, B(x), S(x), @app(R,x))
    λA. λB. λS. λR. &{}
  }
  λT. λS. λR. S
}

@main = @lam(0, #All{#Boo, λx. #Boo}, ^F, [])
//λa.F(a)
```

TEST/REGRESSION_MT_DUP_RACE.HVM

```
//!-T8 -S
@Leaf = λl.λn.l
@Node = λa.λb.λl.λn.n(a)(b)
@Y = λ&f.(λ&x.f(x(x)))(λ&x.f(x(x)))
@mkTree = λ&rec. λ{
  0n : @Leaf
  1n+:λ&p. @Node(rec(p))(rec(p))
}
@main = @Y(@mkTree)(14n)
//!
//EXPECT_PREFIX:- Itrs:
```

TEST/SCOPED_VAR_BOTH.HVM

```
// Tests both X₀ and X₁ used in correct positions
// X₀ in position 0, X₁ in position 1
@main = λx. !X&L=x ; &L{X₀, X₁}
//λa.a
//λa.a
```

TEST/SCOPED_VAR.HVM

```
// Tests that properly scoped variables resolve correctly
// Using X₁ in position 1 of same-labeled SUP is correct
@main = λx. !X&L=x ; &L{1, X₁}
//λa.1
//λa.a
```

TEST/SNF_C2_K2.HVM

```
@c2   = λs. !S0&C=s; λx0.S0₀(S0₁(x0))
@k2   = λs. !S0&K=s; λx0.S0₀(S0₁(x0))
@main = @c2(@k2)
//!λa.λb.A₀;!A&K=B₀(B₁(&K{A₁,b}));!B&K=a;
```

TEST/SNF_UNSCOPED.HVM

```
@main = λk.
  ! F = λx;
  ! K &L = k;
  #Tup{x, F, K₀, K₁, λv.v}
//!λa.#Tup{b,λc.λb.c,A₀,A₁,λd.d};!A&L=a;
```

TEST/SUGAR_CHR.HVM

```
@main = 'x'
//'x'
```

TEST/SUGAR_CONS.HVM

```
@main = 1 <> [2,3]
//[1,2,3]
```

TEST/SUGAR_LST.HVM

```
@main = [1,2,3]
//[1,2,3]
```

TEST/SUGAR_MAT_LST.HVM

```
@head = λ{[]: 0; <>: λh.λt. h}
@main = @head([1,2])
//1
```

TEST/SUGAR_MAT_NAT.HVM

```
@pred = λ{0n: 0n; 1n+: λp. p}
@main = @pred(3n)
//2n
```

TEST/SUGAR_NAT_PLUS.HVM

```
@main = 2n+3
//2n+3
```

TEST/SUGAR_NAT.HVM

```
@main = 3n
//3n
```

TEST/SUGAR_NIL.HVM

```
@main = []
//[]
```

TEST/SUGAR_STR_BT.HVM

```
@main = `hi`
//"hi"
```

TEST/SUGAR_STR.HVM

```
@main = "hi"
//"hi"
```

TEST/SUM_4.HVM

```
#include "_book_.hvm"
@main = @sum(#S{#S{#S{#S{#Z{}}}}})
//#S{#S{#S{#S{#S{#S{#S{#S{#S{#S{#Z{}}}}}}}}}}}
```

TEST/UNICODE_CHAR.HVM

```
// Unicode character literals
@a = 'λ'
@b = '世'
@c = '□'
@d = 'α'
@main = [@a, @b, @c, @d]
//"λ世□α"
```

TEST/UNICODE_CJK.HVM

```
// CJK characters (Chinese, Japanese, Korean)
@main = "你好世界"
//"你好世界"
```

TEST/UNICODE_EMOJI.HVM

```
// Emoji (4-byte UTF-8)
@main = "□□□□"
//"□□□□"
```

TEST/UNICODE_GREEK.HVM

```
// Greek letters commonly used in math/CS
@main = "αβγδλμπσ"
//"αβγδλμπσ"
```

TEST/UNICODE_MIXED.HVM

```
// Mixed ASCII and Unicode
@hello = "Hello"
@world = "世界"
@arrow = "→"
@main = [@hello, @arrow, @world]
//["Hello","→","世界"]
```

TEST/UNICODE_STRING.HVM

```
// Unicode strings with various scripts
@main = "λx.世界□"
//"λx.世界□"
```

TEST/UNSCOPED_CALLCC.HVM

```
@Seq = λ{λa.λ{λb.a}}
@CC_lang =
  λprogram.
  ! callccf = λ callccv ;
  ! garbagef = λ garbagev ;
  ! holef = λ holev ;
  ! callcc = λcallback. garbagef(holev, callback(holef(0)));
  ! result = program(callcc);
  @Seq(result, garbagev)

@main = @CC_lang(λcallcc.(10 + callcc(λk.k(42) + 1729)))
//52
```

TEST/UNSCOPED_LAMBDA_DIRECT.HVM

```
@main = (λ $x,y. (x + y))(10, 20)
//30
```

TEST/UNSCOPED_MUT_REF.HVM

```
@mut =
  λref. λfn.
  ! newf = λ newv ;
  !! new = fn(ref(newv));
  newf(0, new)

@spt =
  λref. λfn.
  ! yf = λ yv ;
  ! zf = λ zv ;
  fn(yf(ref(zv)), zf(yv))

@main =
  ! xf = λ xv ;
  ! X0f = λ X0v ;
  ! X1f = λ X1v ;
  ! X = xf(0);
  !! a = @spt(X, X0f(X1f(0)));
  !! a = @mut(X0v, λx.x+1);
  !! a = @mut(X1v, λx.x+1);
  xv
//2
```

TEST/UNSCOPED_NESTED.HVM

```
// Tests unscoped variable in nested lambda context
// Inner lambda uses B₀ in wrong position, outer A₁ is correct
@main = λa. !A&L=a; &L{λb. !B&L=b; &L{1, B₀}, A₁}
//λa.a
//λa.λb.1
//λa.λb.b
```

TEST/UNSCOPED_QUEUE.HVM

```
@Qnew = λx.x
@Qadd = λx.λq.λk.q(λc.c(x, k))
@Qrem = ! kf = λ kv ; λq.q(kv, λx.λxs.λp.p(x, kf(xs)))
@queue = @Qadd(3, @Qadd(2, @Qadd(1, @Qnew)))
@main =
  @Qrem(@queue, λv0.λq0.
  @Qrem(q0, λv1.λq1.
  @Qrem(q1, λv2.λq2.
  [v0, v1, v2])))
//[1,2,3]
```

TEST/UNSCOPED_VAR.HVM

```
// Tests that unscoped variables print as _ instead of garbage
// Using X₀ in position 1 of same-labeled SUP causes world mismatch
@main = λx. !X&L=x ; &L{1, X₀}
//λa.1
//λa.a
```

TEST/UNSCOPED.HVM

```
@main = ! f = λ v ; #P{f(1, 2), v}
//#P{1,2}
```

TEST/USE_MATCH_DEPTH.HVM

```
// USE+MAT stress: recursively peels a constructor argument through
// a strict USE wrapper, controlled by numeric depth.
@mk = λ{
  0n: #Z{}
  1n+: λp. #S{@mk(p)}
}

@peel = λ{
  0n: λx. x
  1n+: λd. λx.
    ((λ{λv. λ{
      #Z: #Z{}
      #S: λp. @peel(d, p)
    }(v)})(x))
}

@run = λ&d. @peel(d, @mk(d))
@main = @run(100000n)
//#Z{}
```

TEST/USE_SUP_ID.HVM

```
// Test: USE over a superposition must preserve both cloned branches.
@main = λ{λx.x}(&a{1,2})
//1
//2
```

PART K · Workspace context: Bend experiments and docs-site design notes

Orientation material found while scanning the workspace: the bend-runtime README (where the HVM5 work lives), two small Bend 2 experiments about sharing vs copying (bend-branch-graph, bend-branch-lab), and the design/proposal notes for this very docs site (content model, page conventions) which are the starting point for the later course document.

Source root: `bend-runtime` · `bend-branch-graph` · `bend-branch-lab` · `bend-docs/openspec` · **5 files · 15.7 KB**

▼ In this part (file list)

- **K.1** · `bend-runtime/README.md`
- **K.2** · `bend-branch-graph/README.md`
- **K.3** · `bend-branch-lab/README.md`
- **K.4** · `bend-docs/openspec/changes/bend-docs-site/proposal.md`
- **K.5** · `bend-docs/openspec/changes/bend-docs-site/design.md`

K.1 · `bend-runtime/README.md`

bend-runtime — HVM5 v1

HVM5 v1 is a reference runtime for compiled interaction nets. A smart contract is written in Bend 2; deploy compiles it to an inet package; the runtime reduces that package on BendRT. The runtime never parses contract source at send time.

```
contracts/counter.bend
  -> deploy: compile_for_deploy (automatic)
  -> inet package (Term)
  -> parse/reduce in Bend 2
  -> state / receipt
```

The reducer is Bend 2 compiled by BendRT. The deployed contract is inet data, so fork/test/merge and patches operate on the same `Term` representation.

LAYOUT

Path	What
`docs/`	copied Bend tutorial/reference content
`knowledge/`	renamed research base (`bend2-knowledge`)
`hvm5/reference/`	early-draft HVM5 port, kept as history
`hvm5/version1/spec/`	normative v1 spec
`hvm5/version1/plan/`	phases, slices, gates, risks
`hvm5/version1/maps/`	goal/system/type/space/lifecycle/code/risk maps
`hvm5/version1/compiler/`	compiler bootstrap
`hvm5/version1/contracts/`	Bend 2 contract sources
`hvm5/version1/src/`	deploy compiler, inet parser, reducer, simulator
`examples/bend2-basics/`	original small Bend app
`deploy/`	v0 Worker prototype, not the v1 engine

Path	What
`lattice/` absent; see `hvm5/version1/legacy/lattice-v0/`	

QUICKSTART

```
# Check a contract
bend hvm5/version1/contracts/counter.bend --check-only

# Deploy it: source is compiled to inet automatically
bend hvm5/version1/src/deploy.bend hvm5/version1/contracts/counter.bend

# Full ten-node model
bend hvm5/version1/src/sim.bend

# Benchmark
bend hvm5/version1/bench/bench.bend

# Proof gate for the closed topology laws
bend hvm5/version1/PROOF.bend
```

Expected simulation:

```
NODES 10
PUB Counter{4}
PRIV Counter{3}
N4 Counter{3}
ISOLATED True
PUB-OK True
PRIV-OK True
CATCHUP-OK True
REPLICAS-OK True
QUORUM-OK True
BRIDGE-OK True
DIGEST 2815417948
PASS
```

STATUS

Implemented in Bend 2:

- Bend 2 source parser/compiler for the v1 counter contract, invoked automatically by deploy;
- inet package parser and reducer;
- public/private ten-node model with three replicas each;
- kill/heal/catch-up, quorum loss, isolation, bridge negative test;
- fork/test/merge over compiled inet terms;
- benchmark and proof gate;
- native BendRT builds of the reducer and simulator.

Open gaps:

- official checked core IR emitter + generic `bendc.bend` lowering for arbitrary Bend 2; no second frontend;
- full HVM4 rules (DUP-LAM, different-label DUP/SUP);
- native Ed25519/SHA-256 and signed deploy envelopes;
- a real process/network cluster and WAL persistence;
- Worker/WASM target.

The authoritative status is `hvm5/version1/spec/11-acceptance.md`.

K.2 · `bend-branch-graph/README.md`

bend-branch-graph

A persistent program graph for "copy, change a small part, keep the rest", the git-branch idea, written as Bend 2 data. A branch is a new root. Its patch shares every untouched subtree with the parent root.

STATUS

- `LAWS.bend` : four laws drafted. Waiting for your review. No proofs yet.
- `main.bend` : header only until the laws are accepted.
- `spike/` : measurement scaffold, used to check that the design is feasible. It is not the implementation, and it will be replaced after review.

WHAT THE SPIKE MEASURED

Bend 2.0.25, native build, one thread, 12-core machine. Each run commits a chain of patches to a perfect tree of depth `d` .

Variant	Depth	Commits	Time per commit
share: untouched siblings reused through `+`	12	200,000	0.054 µs
share	20	200,000	0.111 µs
copy: untouched siblings copied	12	20,000	22.8 µs
copy	16	20,000	450.8 µs

- Sharing cost grows with the depth of the path. Copying cost grows with the tree size, about 2^depth.
- Persistence check (`spike/demo.bend`): the old root reads 0 after a patch, the new root reads 1, and an untouched path reads 0. Output `(0n, 1n, 0n)` .
- Emitted C (`bend x.bend -o x.c` , compared with a program that shares nothing): the sharing variant adds two `ctr_take` sites and two `term_peek` sites. `term_keep` and `rfc_seal` do not change. The two `ctr_take` sites are the shared siblings. Removing them means copying.

The spike uses `/tmp/opencode/bend-graph` as its working directory and reads a baseline C file from `/tmp/opencode/bend-verify` . Those paths are not part of the repository.

LAWS AND NON-LAWS

Laws, once accepted and proved:

- `read_back` : a write at a fitting path reads back.
- `isolated` : a write leaves every other fitting path unchanged.
- `last_write_wins` : the second write at a path is the value a reader sees.
- `sharing_is_invisible` : the sharing patch equals the copying patch.

Not laws, because the type system does not state cost: time, memory, the number of nodes allocated, and whether two roots share memory.

WHAT BEND 2 CANNOT DO YET

- **Sharing between interaction nets.** BendRT compiles to C and does not reduce interaction nets, so Bend 2 cannot share sub-nets across programs. The Interaction Calculus does this with DUP and SUP (chapter 14 of the tutorial). Its reference binary shows the rules at small scale.
- **HVM5 and "Bend in Bend".** I found no public source. A web search returned nothing, and the local docs and compiler have no reference. A grep matched a random string inside the compiler binary, which is not a mention. I have not designed around either.

DECISIONS FOR REVIEW

1. Accept the four laws, or change them.
2. Start with a generic tree of `Nat` values (as in the spike), or go straight to an `Expr` program tree with `eval` , which is the homoiconic step?
3. Paths: keep binary paths with a `Fits` precondition, or move to depth-indexed trees?
4. Keep the Interaction Calculus experiments as a separate track?

COMMANDS AFTER ACCEPTANCE

```
bend main.bend --check-only
bend main.bend
bend PROOF.bend
bend main.bend -o main && ./main --threads 1
```

K.3 · bend-branch-lab/README.md

bend-branch-lab

A small Bend 2 project for the laws-first workflow in the tutorial's chapter 10 (`bend-docs/src/content/tutorial/10-branching-and-self-change.md`).

STATUS

- `main.bend` : header only. Purpose, expected output, and commands. The implementation waits for review of the laws.
- `LAWS.bend` : six laws drafted, waiting for review. They type-check as statements (against a throwaway stub outside this project they report `Error: 6 TODOs found.` , the expected state before proofs), and they hold on a 25 x 25 x 25 grid of stock, request, and reserve values.
- `PROOF.bend` : not written. It starts after the laws are accepted.

Until `main.bend` defines `Desk.decide` and `Desk.Policy` , `bend LAWS.bend` reports undefined names. That is expected at this stage.

LAYOUT

Follows the Bend demos:

- `main.bend` : the program, under a header comment.
- `LAWS.bend` : imports `main.bend` as `Desk` and states the laws. Human-owned. An accepted law is never deleted or weakened.
- `PROOF.bend` : imports both, proves each law with `def Laws.<name>` . Agent-written after acceptance.

COMMANDS (AFTER ACCEPTANCE)

```
bend main.bend --check-only
bend main.bend
bend PROOF.bend
bend main.bend -o main && ./main --threads 1 && ./main --threads 12
```

`bend PROOF.bend` is the gate. Read its output, not only its exit status: Bend 2.0.25 can exit 0 when a proof depends on `@unsafe` code.

REQUIREMENTS AND WHAT THEY MAP TO

Requirement	Status	Law or note
Main keeps working	law	`plain_grants_min` pins the main policy
A new branch is tested	law	`reserve_keeps_back` pins the reserve branch
A fork that changes nothing behaves like main	law	`zero_reserve_is_plain`
A branch never grants more than main	law	`reserve_never_more` (difference law)
A branch keeps its reserve	law	`reserve_is_kept`
Every future policy stays safe	law	`never_oversells` quantifies over every `Desk.Policy`
SupGen	not a law	not in the released compiler; the site calls it private
Copy-on-write forks with `sup` and `dup`	not a law	no `sup` syntax; `Array.fork` is `@unsafe`

Requirement	Status	Law or note
Homoiconic self-change	not yet	an `Expr` interpreter would be the next step, after review
Cheap forks	not a law	cost is measured, not proved
Test branches cheaply and fast	not a law	the gate checks the laws; run time is measured with native builds

DECISIONS TO REVIEW

1. The domain: a stock desk with `Plain` and `Reserve` policies. Replace it with your own domain if you have one.
2. Parallel forks: if this project adds a parallel tree, the midpoint must be `i + (L << p)`, where `L` is the leaf size. `i + (1 << p)` overlaps leaves that cover more than one index.
3. `--verdict` needs Bend 2.0.36. Bend 2.0.25, the version installed when this was written, does not have it.

K.4 · bend-docs/openspec/changes/bend-docs-site/proposal.md

WHY

Bend (massively parallel language), HVM/Interaction Calculus (its runtime), and the unreleased Bend2 are scattered across `bend2.dev` notes, Taelin's 261 personal repos, and HigherOrderCO org repos. No single source explains the main concept simply first, then deep. A from-scratch Astro docs site, deployed reproducibly via Alchemy to Cloudflare, fixes that.

WHAT CHANGES

- Research-first content sourcing: deep-read all 14 `bend2.dev` pages (release tracker, 7 notes, 5 learn pages) plus Taelin/HigherOrderCO source repos (Bend2, Interaction-Calculus, hvm4-formal, Bend, HVM, Kind) and distill verified concepts with citations and `Last updated` dates.
- New docs site in `bend-docs/`: concept-first pages (what/why in plain language, then deep dives), runnable-style Bend examples clearly marked by verification status, diagrams for interaction nets / HVM reduction.
- Astro static build + Alchemy `Cloudflare.Website.Astro` (`output: static`, assets-only) deploy to the user's Cloudflare account; Cloudflare Speed Brain for prefetch, no custom prefetch code.
- OpenSpec stays the long-term record: specs + changes versioned in this repo for every future improve/delete decision.

CAPABILITIES

NEW CAPABILITIES

- `content-model`: what the docs cover and how (concept-first structure, verification labels, citation + last-updated rules, `index.md` mirrors, `llms.txt`).
- `docs-site`: observable site behavior (routes, navigation, themes, search/ToC, fonts, zero-JS default, offline/print behavior).
- `cloudflare-deploy`: reproducible Alchemy deploy (static assets-only Worker, custom domain handling, Speed Brain + cache expectations, credential/profile flow).

MODIFIED CAPABILITIES

- None (greenfield; `openspec/specs/` is empty).

IMPACT

- New code: Astro site (`src/`, `public/`), `alchemy.run.ts`, package setup (bun). No changes to existing code (repo currently only holds `openspec/` + `.opencode/`).
- External systems: Cloudflare account (Workers Assets, optional custom domain, Speed Brain toggle). Requires user login (`alchemy login` OAuth or API token) — blocked on user until credentials provided.
- Sources are third-party and pre-release: content must never claim unverified Bend2 code compiles; every claim cites canonical URL + date.

K.5 · bend-docs/openspec/changes/bend-docs-site/design.md

CONTEXT

See `proposal.md` for motivation. Current state: `bend-docs/` holds only `openspec/ + .opencode/` (greenfield). Sources verified so far: `bend2.dev` serves 14 pages as Astro static on Cloudflare (`cf-cache-status: HIT` , `~28kB` , `speculation-rules: /cdn-cgi/speculation` → Speed Brain `conservative` prefetch); VictorTaelin has 261 repos with key ones `Bend2` (WIP), `Interaction-Calculus` (948 stars), `hvm4-formal` , plus HigherOrderCO `Bend` / `HVM` / `Kind` ; Alchemy provides `Cloudflare.Website.Astro` with `astro: { output: "static" }` assets-only deploys and profile-based login (`alchemy login` , `~/.alchemy/profiles.json`).

GOALS / NON-GOALS

Goals:

- Research-first: read before writing; every claim cited with source URL + date.
- Concept-first content (plain idea, then depth) with honest verification labels.
- Reproducible edge deploy a second person can repeat from docs alone.

Non-Goals:

- No Bend2 compiler/runtime implementation or fork of Bend/HVM code.
- No SSR, auth, comments, or analytics beyond a minimal privacy-friendly beacon.
- No visual clone of `bend2.dev` ; original minimal paper-style design.

DECISIONS

- **Astro `output: static` , no adapter in config**: `bend2.dev` proves the shape works (zero JS framework, inline CSS, preloaded subset fonts). Alternative `output: server` rejected — no dynamic need; server bundle adds cost and complexity.
- **Alchemy `StaticSite` over the `Astro` resource and raw wrangler**: single `alchemy.run.ts` declares build + assets (`notFoundHandling: 404-page`); no `wrangler.jsonc` hand-maintenance, content-hashed deploys skip no-op rebuilds. The dedicated `Astro` resource was tried first and rejected: its builder injects a rolldown plugin that crashes with Astro 7's bundled Vite (`esmExternalRequirePlugin is not a function`), while plain `bun run build` succeeds. Revisit if alchemy/astro versions realign.
- **Cloudflare Speed Brain over Astro/inline prefetch**: the analyzed instant-navigation effect comes from the edge `Speculation-Rules` header, not site code. Ship zero prefetch JS; document the dashboard toggle. Alternative `astro:prefetch` / inline speculation rules rejected to keep bundles at zero.
- **Research notes live in the change, specs version long-term**: per-page notes under the change during research; only durable behavior contracts graduate to `openspec/specs/` at archive. Keeps the record useful for future improve/delete calls.

RISKS / TRADE-OFFS

- [Bend2 is pre-release; facts decay fast] → every page shows source date + own verification date; research tasks re-check sources before drafting.
- [261 personal repos + org repos is too much to read fully] → bound scope to key repos + READMEs first; gap notes for the rest instead of pretending full coverage.
- [Cloudflare credentials not yet provided] → design and site build proceed without them; deploy task stays blocked until user runs `alchemy login` .
- [Bend examples may mislead if presented as runnable] → verification labels are a spec requirement, not a nice-to-have.

MIGRATION PLAN

Greenfield, no migration. Deploy: `bun install` → `bun run build` → `alchemy login` (user) → `alchemy deploy` . Rollback: redeploy prior commit via Alchemy; static assets make this a cache-purge at most. Custom domain is a later additive step.

OPEN QUESTIONS

- Cloudflare account + domain: reuse an existing zone or new subdomain? (Deferrable; deploy works on `workers.dev` first.)
- Content language: English only for v1? (Assumed yes; recorded in tasks.)

- Speed Brain on `workers.dev`: the `speculation-rules` header is absent on the workers.dev URL (platform zone, no user toggle); verify it appears once a custom domain is attached and Speed Brain is enabled there.

PART L · HVM4 at HEAD `6defdfc` (2026-05-29), verbatim

> The current upstream HVM4 documentation. Part I holds the March 2026 snapshot (`2d7f28e` , multi-file `clang/` tree with AOT/FFI/threads); since commit `4bc2336` the runtime is one sequential C file, `src/hvm.c` , and these docs describe it. Read Part L for the present machine, Part I for history. Copied byte for byte; only the banner is added.

Source: `https://github.com/HigherOrderCO/HVM4` · commit `6defdfc7dae2a3cca5dd6e74ed0612385b5646a8 Fri May 29 23:42:54 2026 -0300` · retrieved 2026-11-06 · **69 files**

▼ In this part (file list)

- **L.1** · `HVM4/README.md`
- **L.2** · `HVM4/AGENTS.md`
- **L.3** · `HVM4/docs/primer.md`
- **L.4** · `HVM4/docs/theory/interaction_calculus.md`
- **L.5** · `HVM4/docs/hvm/core.md`
- **L.6** · `HVM4/docs/hvm/memory.md`
- **L.7** · `HVM4/docs/hvm/syntax.md`
- **L.8** · `HVM4/docs/hvm/collapser.md`
- **L.9** · `HVM4/docs/hvm/interactions/alo_cop.md`
- **L.10** · `HVM4/docs/hvm/interactions/alo_dry.md`
- **L.11** · `HVM4/docs/hvm/interactions/alo_dup.md`
- **L.12** · `HVM4/docs/hvm/interactions/alo_lam.md`
- **L.13** · `HVM4/docs/hvm/interactions/alo_nam.md`
- **L.14** · `HVM4/docs/hvm/interactions/alo_nod.md`
- **L.15** · `HVM4/docs/hvm/interactions/alo_var.md`
- **L.16** · `HVM4/docs/hvm/interactions/and_era.md`
- **L.17** · `HVM4/docs/hvm/interactions/and_inc.md`
- **L.18** · `HVM4/docs/hvm/interactions/and_num.md`
- **L.19** · `HVM4/docs/hvm/interactions/and_sup.md`
- **L.20** · `HVM4/docs/hvm/interactions/app_ctr.md`
- **L.21** · `HVM4/docs/hvm/interactions/app_dry.md`
- **L.22** · `HVM4/docs/hvm/interactions/app_era.md`
- **L.23** · `HVM4/docs/hvm/interactions/app_inc.md`
- **L.24** · `HVM4/docs/hvm/interactions/app_lam.md`
- **L.25** · `HVM4/docs/hvm/interactions/app_mat_ctr.md`
- **L.26** · `HVM4/docs/hvm/interactions/app_mat_num.md`
- **L.27** · `HVM4/docs/hvm/interactions/app_mat_sup.md`
- **L.28** · `HVM4/docs/hvm/interactions/app_nam.md`
- **L.29** · `HVM4/docs/hvm/interactions/app_sup.md`
- **L.30** · `HVM4/docs/hvm/interactions/ddu_era.md`
- **L.31** · `HVM4/docs/hvm/interactions/ddu_inc.md`
- **L.32** · `HVM4/docs/hvm/interactions/ddu_num.md`
- **L.33** · `HVM4/docs/hvm/interactions/ddu_sup.md`
- **L.34** · `HVM4/docs/hvm/interactions/dsu_era.md`
- **L.35** · `HVM4/docs/hvm/interactions/dsu_inc.md`
- **L.36** · `HVM4/docs/hvm/interactions/dsu_num.md`
- **L.37** · `HVM4/docs/hvm/interactions/dsu_sup.md`
- **L.38** · `HVM4/docs/hvm/interactions/dup_dry.md`
- **L.39** · `HVM4/docs/hvm/interactions/dup_lam.md`
- **L.40** · `HVM4/docs/hvm/interactions/dup_nam.md`
- **L.41** · `HVM4/docs/hvm/interactions/dup_nod.md`

- **L.42** · HVM4/docs/hvm/interactions/dup_sup.md
- **L.43** · HVM4/docs/hvm/interactions/eql_any.md
- **L.44** · HVM4/docs/hvm/interactions/eql_ctr.md
- **L.45** · HVM4/docs/hvm/interactions/eql_dry.md
- **L.46** · HVM4/docs/hvm/interactions/eql_era.md
- **L.47** · HVM4/docs/hvm/interactions/eql_inc.md
- **L.48** · HVM4/docs/hvm/interactions/eql_lam.md
- **L.49** · HVM4/docs/hvm/interactions/eql_mat.md
- **L.50** · HVM4/docs/hvm/interactions/eql_nam.md
- **L.51** · HVM4/docs/hvm/interactions/eql_num.md
- **L.52** · HVM4/docs/hvm/interactions/eql_sup.md
- **L.53** · HVM4/docs/hvm/interactions/eql_use.md
- **L.54** · HVM4/docs/hvm/interactions/mat_inc.md
- **L.55** · HVM4/docs/hvm/interactions/op2_era.md
- **L.56** · HVM4/docs/hvm/interactions/op2_inc.md
- **L.57** · HVM4/docs/hvm/interactions/op2_num_era.md
- **L.58** · HVM4/docs/hvm/interactions/op2_num_num.md
- **L.59** · HVM4/docs/hvm/interactions/op2_num_sup.md
- **L.60** · HVM4/docs/hvm/interactions/op2_sup.md
- **L.61** · HVM4/docs/hvm/interactions/or_era.md
- **L.62** · HVM4/docs/hvm/interactions/or_inc.md
- **L.63** · HVM4/docs/hvm/interactions/or_num.md
- **L.64** · HVM4/docs/hvm/interactions/or_sup.md
- **L.65** · HVM4/docs/hvm/interactions/uns.md
- **L.66** · HVM4/docs/hvm/interactions/use_era.md
- **L.67** · HVM4/docs/hvm/interactions/use_inc.md
- **L.68** · HVM4/docs/hvm/interactions/use_sup.md
- **L.69** · HVM4/docs/hvm/interactions/use_val.md

L.1 · HVM4/README.md

HVM

HVM is a high-performance runtime for the Interaction Calculus.

NOTE: you're here before launch. Use at your own risk.

BUILDING AND RUNNING

```
# Build
clang -O2 -o src/hvm src/hvm.c

# Run a file (use collapse mode by default)
hvm devs/test/file.hvm -s -C10

# Run all tests
./devs/test/_all_.sh

# Run a benchmark file
hvm devs/bench/u32_fib.hvm -s
```

Flags:

- `-s` shows performance stats
- `-D` prints each intermediate reduction step with interaction labels
- `-C10` collapses and flattens superpositions (limit to 10 lines)

EXAMPLES

```
@main = ((@add 1) 2)
//3
```

```
@main = (&{1, 2} + 10)
//11
//12
```

```
@main = (! x &A= 3; (x₀ + x₁))
//6
```

DOCUMENTATION

- Theory: docs/theory/interaction_calculus.md
- Core language: docs/hvm/core.md
- Memory layout: docs/hvm/memory.md
- Interaction rules: docs/hvm/interactions/

L.2 · `HVM4/AGENTS.md`

AGENTS.md

QUICK ONBOARDING

HVM is a runtime for the Interaction Calculus (IC), a lambda-calculus extension with explicit duplication (DUP) and superposition (SUP). These forms enable optimal sharing for lazy evaluation, even inside lambdas. This repo is the C runtime: parse source -> build static book terms -> lazily allocate dynamic heap terms -> reduce with WNF/SNF interactions -> print results.

Key terms:

- static/book term: immutable definition stored in the book (de Bruijn levels).
- dynamic term: mutable heap term used during evaluation (linked by pointers).
- WNF: weak normal form (head reduction with interactions).
- SNF: strong normal form (full reduction).
- CNF: collapsed normal form (full lambda-calculus readback).
- interaction: a rewrite rule (APP-LAM, DUP-SUP, etc.).

BUILD AND TEST

```
# Build
clang -O2 -o src/hvm src/hvm.c

# Run a file
hvm devs/test/file.hvm -s -C10

# Run all tests
./devs/test/_all_.sh
```

DOCS MAP

- `README.md` : entry point, build/run examples, links.
- `docs/primer.md` : quick intro to the language and runtime usage.
- `docs/theory/interaction_calculus.md` : IC theory + examples.
- `docs/hvm/core.md` : core term AST and grammar.
- `docs/hvm/syntax.md` : parser syntax, precedence, and desugaring rules.
- `docs/hvm/memory.md` : term layout, heap representation, linked/quoted terms.
- `docs/hvm/collapser.md` : CNF readback and collapse algorithm.

- `docs/hvm/interactions/*.md` : one file per WNF interaction; mirrors the sequent calculus comments in the WNF section of `src/hvm.c` .

CODE MAP (C RUNTIME)

`src/hvm.c` is the only C runtime source. It is organized as broad sections, inspired by Bend's `Core.ts` :

- `Types` : scalar aliases, term tags, parser/runtime structs, globals.
- `Term` : term packing, tag/ext/val helpers, constructors, clone helpers, OP2.
- `Heap` : allocation, heap access, and substitutions.
- `Nick` , `System` , `Table` , `Print` : names, file/path helpers, intern table, and dynamic/static pretty-printing.
- `Runtime` : process setup, program preparation, entry lookup, and @main eval.
- `Parse` : lexer, bindings, syntax sugar, includes, and top-level definitions.
- `WNF` : stack evaluator and interaction rules.
- `Data` : small runtime data structures used by evaluation.
- `CNF` : one-step collapsed normal form readback and SUP lifting.
- `Eval` : SNF traversal and CNF branch enumeration.
- `CLI` : option parsing and program entry point.

There is no external call layer. HVM now runs pure programs and prints the result of evaluating `@main` .

L.3 · HVM4/docs/primer.md

HVM Primer

HVM is a runtime for the Interaction Calculus, a model of computation that extends lambda calculus to achieve **optimal reduction**: no work is ever duplicated, even when copying functions. It's lazy like Haskell, but sharing extends inside lambdas, not just at the data level.

RUNNING HVM

```
hvm file.hvm -s -C10 # run with collapse
```

Flags:

- `-s` : show performance stats
- `-D` : print intermediate reduction steps
- `-C10` : collapse superpositions (limit output to 10 lines)

Modes:

- *Normalize*: reduce to SNF. Keeps SUPs and DUPs in output.
- *Collapse*: eliminate SUPs and DUPs, enumerate as pure lambda terms.

GRAMMAR

```
File     ::= (Include | Def)*
Include  ::= "#include" '"' Path '"'
Def      ::= "@" Name "=" Term

Term ::=
  | Name                                    -- variable
  | Name "₀" | Name "₁"                     -- dup variable (first/second)
  | Integer                                 -- number literal
  | "@" Name                                -- reference to definition
  | "*"                                     -- wildcard (erased)

  -- Lambda & Application
  | "λ" Params "." Term                     -- lambda: λx.body
  | "λ" "&" Name "." Term                   -- cloned lambda: λ&x.body
  | "λ" Name "&" Label "." Term             -- lambda with inline dup
  | Term "(" Args ")"                       -- application: f(x,y)

  -- Pattern Matching
  | "λ" "{" Cases "}"                       -- match lambda
  | "λ" "{" Term "}"                        -- use (strict unbox)

  -- Duplication & Superposition
  | "!" Name "&" Label "=" Term ";" Term    -- dup: !x&L=val;body
  | "!" Name "&" "(" Term ")" "=" Term ";" Term -- dynamic dup
  | "&" Label "{" Term "," Term "}"         -- superposition: &L{a,b}
  | "&" "(" Term ")" "{" Term "," Term "}"  -- dynamic superposition
  | "&" "{" "}"                             -- erasure

  -- Let Bindings (sugar)
  | "!" Name "=" Term ";" Term              -- let: !x=val;body
  | "!" "&" Name "=" Term ";" Term          -- cloned let
  | "!!" Name "=" Term ";" Term             -- strict let

  -- Constructors
  | "#" Name "{" Args "}"                   -- constructor: #Pair{a,b}
  | "#" Name "{" "}"                        -- nullary: #ZER{} or #True{}

  -- Operators
  | Term Op Term                            -- binary: (a + b), (x == y)
  | Term "===" Term                         -- structural equality
  | Term ".&." Term                         -- short-circuit AND
  | Term ".|." Term                         -- short-circuit OR

  -- Special
  | "↑" Term                                -- priority (collapse order)
  | "&" Label "λ" Names "{" Term ";" Term "}"   -- fork
  | "[" Args "]"                            -- list: [a,b,c]
  | '"' Chars '"'                           -- string
  | "'" Char "'"                            -- char: 'x' -> #CHR{120}
  | Integer "n"                             -- nat: 2n -> #SUC{#SUC{#ZER}}

Cases    ::= (Pattern ":" Term ";")* [Default]
Pattern  ::= "#" Name | Integer | "[]" | "<>" | Integer "n" | Integer "n+"

Operators (low to high): || && == != < <= > >= << >> + - * / % ^
```

KEY CONCEPTS

AFFINE VARIABLES

Variables are **affine**: used at most once.

```
// WRONG: x used twice
@bad = λx. (x + x)

// RIGHT: use cloned binder
@square = λ&x. (x * x)
@main = @square(5)
//25
```

The `&` prefix tells the parser to auto-insert duplication nodes.

Common affinity errors:

1. Using a variable in multiple match branches:

```
// WRONG: x appears in both branches
@bad = λx. λ{
  #T: x
  #F: x
}

// RIGHT: match first, then bind x in each branch
@good = λ{
  #T: λx. x
  #F: λx. x
}
@main = @good(#T{}, 42)
//42
```

2. Using unnecessary clones:

```
// BAD: inline λ{...}(x) is poor style

@filter = λ&f. λ{
  []: []
  <>: λ&h. λ&t.
    λ{
      #T: h <> @filter(f, t)
      #F: @filter(f, t)
    }(f(h))
}

// GOOD: split into two functions

@keep = λ{
  #T: λh. λt. h <> t
  #F: λh. λt. t
}

@filter = λ&f. λ{
  []: []
  <>: λ&h. λt. @keep(f(h), h, @filter(f, t))
}
```

It is very important, on HVM, to avoid dups as much as possible. As such, it is considered a good style to format all top-level definitions as case-trees, rearranging arguments to avoid unecessary clones, and avoiding inline λ-match applications (such as `λ{...}(x)`).

SUPERPOSITION (SUP)

A SUP `&L{a,b}` represents two values in one location:

```
@main = (&X{10, 20} + 5)
//15
//25
```

When collapsed (`-C` flag), each branch prints separately.

LABEL MECHANICS

Labels control how DUPs interact with SUPs:

- **Same label**: DUP and SUP annihilate (pair extraction)
- **Different labels**: they commute (cross product)

To observe this, use explicit DUP syntax `!x&L = val; body` which binds x_0 and x_1:

```
// Same label: pairwise extraction
@main =
  !x&A = &A{1, 2};
  [x₀, x₁]
//[1,2]
```

```
// Different labels: the DUP stays inside each SUP branch
@main =
  !x&A = &B{1, 2};
  [x₀, x₁]
//[1,1]
//[2,2]
```

```
// Cross product from two SUPs with different labels (4 results)
@main = [&A{1, 2}, &B{10, 20}]
//[1,10]
//[1,20]
//[2,10]
//[2,20]
```

Prefer `λ&x` for general functions. Use explicit DUPs only when you need label control (pairwise extraction or explicit SUP/DUP interactions).

PATTERN MATCHING

Match on constructors with `λ{...}`:

```
@not = λ{
  #T: #F{}
  #F: #T{}
}
@main = @not(#T{})
//#F{}
```

```
@pred = λ{
  0n:  0n
  1n+: λp. p
}
@main = @pred(3n)
//2n
```

Match cases bind constructor fields as lambda arguments.

PEANO NATURALS

`Nn` is sugar for `#SUC{...#SUC{#ZER}...}`. Patterns: `0n:` matches `#ZER`, `1n+:` matches `#SUC` and binds predecessor:

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}
@main = @add(2n, 3n)
//5n
```

CHARACTERS

A char literal `'x'` desugars to `#CHR{120}` (the constructor `#CHR` wrapping the Unicode codepoint as a number). Escape sequences: `'\n'`, `'\t'`, `'\r'`, `'\0'`, `'\\'`, `'\''`, `'\"'`.

```
@main = 'A'
//'A'
```

To match on a character, match the `#CHR` constructor to extract the codepoint, then switch on the numeric value:

```
@is_a = λ{
  #CHR: λ{'A': #T; λn. #F}
}

@main = @is_a('A')
//#T{}
```

LISTS

Lists use the built-in constructors `#CON` and `#NIL` (all caps).

Sugar:

- `[]` desugars to `#NIL{}`
- `[a, b, c]` desugars to `#CON{a, #CON{b, #CON{c, #NIL{}}}}`
- `a <> b` desugars to `#CON{a, b}` (cons)

Pattern sugar:

- `[]:` matches `#NIL`
- `<>:` matches `#CON`

```
@len = λ{
  []: 0
  <>: λh. λt. (1 + @len(t))
}
@main = @len([1, 2, 3, 4, 5])
//5
```

STRINGS

Strings are lists of `#CHR`. `"hi"` desugars to:

#CON{#CHR{104}, #CON{#CHR{105}, #NIL{}}}

Backtick can be used instead of quotes.

```
@main = "hi"
//"hi"
```

Since strings are just lists, list functions work on them directly:

```
...
@main = @len("hello")
//5
```

NUMERIC PATTERNS

Switch on machine ints with `0:`, `1:`, etc., and `λn.` for default:

```
@fib = λ{
  0: 0
  1: 1
  λ&n. (@fib((n - 1)) + @fib((n - 2)))
}
@main = @fib(10)
//55
```

FORK NOTATION

`&Lλx,y{A;B}` is sugar for paired branching. A uses x_0, y_0; B uses x_1, y_1:

```
@main = (&Lλx,y{(x + y); (x * y)})(3, 4)
//7
//12
```

This desugars to:

```
@main =
  !x&L = 3;
  !y&L = 4;
  &L{(x₀ + y₀), (x₁ * y₁)}
//7
//12
```

UNSCOPED LAMBDAS

Unscoped lambdas allow a variable to be used outside its binder's lexical scope. You can write them directly as:

```
λ$x. body
```

This is sugar for:

```
! f = λ x ; body
```

This declares in `body`:

- `f = λy. λ$x. y` — a function that wraps its argument in an unscoped lambda
- `x = $x` — an unscoped variable usable anywhere

So `λ$x.body` is equivalent to constructing it as `f(body)`:

```
@main = (λ$x. (x + 1))(10)
//11
```

Applying the unscoped lambda to `10` binds `x` to `10`, yielding `11`. (Equivalent sugar: `! f = λ x ; f(x + 1)(10)`.)

Another example using the unscoped variable in a pair:

```
@main = ! f = λ v ; #P{f(1, 2), v}
//#P{1,2}
```

Here `f(1, 2)` is `f(1)(2)` which produces `λ$v.1` then applies it to `2`, binding `v` to `2`. The pair contains `1` and the captured value `2`.

This is useful for continuation-passing or "exporting" a binding out of a nested context.

EXAMPLES

FACTORIAL WITH PEANO NATURALS

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}

@mul = λ{
  0n:  λb. 0n
  1n+: λa. λ&b. @add(b, @mul(a, b))
}

@fac = λ{
  0n:  1n
  1n+: λ&p. @mul(1n+p, @fac(p))
}

@main = @fac(3n)
//6n
```

LIST MAP

```
@map = λ&f. λ{
  []: []
  <>: λh. λt. (f(h) <> @map(f, t))
}
@main = @map(λx. (x * 2), [1, 2, 3])
//[2,4,6]
```

ENUMERATING BINARY STRINGS

Superpositions can enumerate multiple results; each branch prints separately in collapse mode:

```
@bits = λ{
  0n:  #E{}
  1n+: λ&p.
    &B{#O{@bits(p)}, #I{@bits(p)}}
}
@main = @bits(2n)
//#O{#O{#E{}}}
//#O{#I{#E{}}}
//#I{#O{#E{}}}
//#I{#I{#E{}}}
```

CHURCH NUMERALS

Standard lambda encoding with a cloned binder for `s`:

```
 @c2   = λ&s. λz. s(s(z))
@cmul = λa. λb. λs. λz. a(b(s), z)
@main = @cmul(@c2, @c2)
//λa.λb.a(a(a(a(b))))
```

SOLVING EQUATIONS VIA ENUMERATION

```
@add = λ{
  0n:  λb. b
  1n+: λa. λb. 1n+@add(a, b)
}

@eq = λ{
  0n:  λ{
    0n:  #T{}
    1n+: λp. #F{}
  }
  1n+: λa. λ{
    0n:  #F{}
    1n+: λb. @eq(a, b)
  }
}

@if = λ{
  #T: λt. λf. t
  #F: λt. λf. f
}

// All naturals: 0, 1, 2, ...
@X = &N{0n, 1n+@X}

// Solve X + 2 = 4
@main = @if(@eq(@add(@X, 2n), 4n), @X, &{})
//2n
```

FILTER

```
@even = λ{
  0: #T{}
  1: #F{}
  λp. @even((p - 2))
}

@filter = λ&f. λ{
  []: []
  <>: λ&h. λt. @keep(f(h), h, @filter(f, t))
}

@keep = λ{
  #T: λh. λr. (h <> r)
  #F: λh. λr. r
}

@main = @filter(@even, [0, 1, 2, 3, 4, 5, 6])
//[0,2,4,6]
```

COMMON PATTERNS

- `λ&x.` : Cloned binder (auto-dup)
- `!x&L = v; body` : Explicit dup with label L
- x_0 , x_1 : Dup variable branches
- `&L{a,b}` : Superposition
- `&{}` : Erasure (discard value)
- `a <> b` : List cons
- `@name` : Reference to definition

Constructor match:

```
λ{
  #C: ...
}
```

Peano natural match:

```
λ{
  0n:  ...
  1n+: λp. ...
}
```

Numeric switch (machine ints):

```
λ{
  0:  ...
  1:  ...
  λn. ...
}
```

GOTCHAS

1. **No whitespace application**: use `f(x)` not `f x`
2. **Variables are affine**: use `&` for multiple uses
3. **Subscripts required**: dup-bound vars need `₀` or `₁`
4. **Labels matter**: same labels annihilate, different multiply branches
5. **Constructor syntax**: always `#Name{...}`, even nullary `#ZER{}`
6. **Pattern arms are lambdas**: `1n+: λpred. body` "binds" the field
7. **No inline match application**: avoid writting `λ{...}(x)`

REDUCTION MODES

- **WNF**: weak normal form (head only)
- **SNF**: strong normal form (full, keeps SUPs/DUPs)
- **CNF**: collapsed normal form (enumerate as pure lambda terms)

Use `-C` flag for collapse mode, omit for normalize mode.

L.4 · HVM4/docs/theory/interaction_calculus.md

Interaction Calculus

The Interaction Calculus is a model of computation that extends the Lambda Calculus with **duplications** and **superpositions**, two core forms that enable optimal lazy evaluation.

CONTENTS

- Optimal Lazy Evaluation
- Duplications
- Superpositions
- Duplicating Lambdas
- Dup/Sup Labels
- The Four Core Interactions
- Relation to Interaction Combinators
- Complete Examples

OPTIMAL LAZY EVALUATION

Programs can be evaluated **strictly** (compute everything immediately) or **lazily** (compute only when needed). Lazy evaluation avoids unnecessary work, but has a flaw: when a value is used twice, its computation might happen twice. Haskell solves this with *memoization* (thunks), caching results so repeated access doesn't recompute. This works for data, but breaks down inside lambdas.

The Interaction Calculus achieves **optimal sharing**: work is never duplicated, even inside lambdas. Two dual forms make this possible:

- **Duplications**: allow a single value to exist in multiple locations
- **Superpositions**: allow multiple values to exist in a single location

These correspond to Lafont's "fan nodes" in Interaction Combinators (1997), with opposite polarities. Together, they provide the mechanics for lazy cloning that extends into lambda bodies.

DUPLICATIONS

Duplications allow a **single value** to exist in **two locations**. The `! x &= v; t` construct duplicates `v`, making it available as `x₀` and `x₁` in `t`:

```
! x &= 2;
(x₀ + x₁)
--------- DUP-NUM
(2 + 2)
--------- OP2-NUM
4
```

Here, the number `2` was duplicated, then added to itself.

Duplication is incremental - it happens layer by layer, on demand:

```
! x &= [1, 2, 3];
(x₀, x₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [2, 3];
(1 <> t₀, 1 <> t₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [3];
(1 <> 2 <> t₀, 1 <> 2 <> t₁)
------------------------------------ DUP-CTR, DUP-NUM
! t &= [];
(1 <> 2 <> 3 <> t₀, 1 <> 2 <> 3 <> t₁)
------------------------------------ DUP-CTR
([1, 2, 3], [1, 2, 3])
```

The list was cloned element by element. Each step peels off one layer - duplicating the head and creating a duplication for the tail. The tail duplication only triggers when both copies are accessed.

SUPERPOSITIONS

Superpositions allow **two values** to exist in **one location**. The `&{a, b}` construct creates a superposition of `a` and `b`:

```
(&{1, 2} + 10)
--------------------- OP2-SUP
! x &= 10;
&{(1 + x₀), (2 + x₁)}
--------------------- DUP-NUM
&{(1 + 10), (2 + 10)}
--------------------- OP2-NUM, OP2-NUM
&{11, 12}
```

Here, we added `10` to a superposition of `1` and `2`. The addition applied to both values, producing a superposition of results. Notice that `10` had to be duplicated; SUPs generate DUP nodes (DP0/DP1) as byproducts, and vice-versa.

Superpositions and duplications are duals. When a DUP meets a SUP, they annihilate, extracting the two values:

```
! x &= &{1, 2};
(x₀ + x₁)
--------------- DUP-SUP
(1 + 2)
--------------- OP2-NUM
3
```

This is like a pair projection: `x₀` gets the first element, `x₁` gets the second. SUPs and DUP nodes create and eliminate each other, just like LAMs and APPs.

DUPLICATING LAMBDAS

Lambdas are also duplicated incrementally. When that happens, the bound variable becomes superposed, and temporarily escapes its scope:

```
! f &= λx.(x + 1);
(f₀(10), f₁(20))
---------------------------- DUP-LAM
! b &= (&{$x0, $x1} + 1);
((λ$x0.b₀)(10), (λ$x1.b₁)(20))
---------------------------- APP-LAM, APP-LAM
! b &= (&{10, 20} + 1);
(b₀, b₁)
---------------------------- OP2-SUP, DUP-NUM
! b &= &{(10 + 1), (20 + 1)};
(b₀, b₁)
---------------------------- OP2-NUM, OP2-NUM
! b &= &{11, 21};
(b₀, b₁)
---------------------------- DUP-SUP
(11, 21)
```

Notice how, on the first step, the variables `$x0` and `$x1` are bound *outside* the lambda's body. This is why the Interaction Calculus needs globally scoped variables. It's also what enables **optimal sharing inside lambdas**: the body is now shared by `b`, so any computation there only happens once.

DUP/SUP LABELS

Consider this example:

```
(&A{1, 2} + &A{10, 20})
---------------------- OP2-SUP
! x &A= &A{10, 20};
&A{(1 + x₀), (2 + x₁)}
---------------------- DUP-SUP (same label: annihilate)
&A{(1 + 10), (2 + 20)}
---------------------- OP2-NUM, OP2-NUM
&A{11, 22}
```

Here, the superpositions *annihilated*: the first element paired with the first, and the second with the second. But what if we wanted all combinations instead?

Labels control this behavior. When a DUP meets a SUP with the *same* label, they annihilate. With *different* labels, they commute:

```
(&A{1, 2} + &B{10, 20})
------------------------------------------------ OP2-SUP
! x &A= &B{10, 20};
&A{(1 + x₀), (2 + x₁)}
------------------------------------------------ DUP-SUP (different labels: commute)
! a &A= 10;
! b &A= 20;
&A{(1 + &B{a₀, b₀}), (2 + &B{a₁, b₁})}
------------------------------------------------ DUP-NUM, DUP-NUM
&A{(1 + &B{10, 20}), (2 + &B{10, 20})}
------------------------------------------------ OP2-SUP, OP2-SUP
&A{&B{(1 + 10), (1 + 20)}, &B{(2 + 10), (2 + 20)}}
------------------------------------------------ OP2-NUM (x4)
&A{&B{11, 21}, &B{12, 22}}
```

Now we get a nested superposition containing all four results. Labels let us control whether superpositions collapse together or stay independent.

THE FOUR CORE INTERACTIONS

The minimal Interaction Calculus has just four rules. Two create computation (APP-LAM, DUP-SUP) and two propagate it (APP-SUP, DUP-LAM):

APP-LAM - Application eliminates lambda:

```
(λx.body)(arg)
-------------- APP-LAM
x ← arg
body
```

DUP-SUP - Duplication eliminates superposition (same label):

```
! x &L= &L{a, b}; t
------------------- DUP-SUP
x₀ ← a
x₁ ← b
t
```

APP-SUP - Application propagates through superposition:

```
(&L{a, b})(c)
------------------- APP-SUP
! x &L= c;
&L{a(x₀), b(x₁)}
```

DUP-LAM - Duplication propagates through lambda:

```
! f &L= λx.body; t
------------------ DUP-LAM
f₀ ← λ$X0.B₀
f₁ ← λ$X1.B₁
x  ← &L{$X0, $X1}
! B &L= body;
t
```

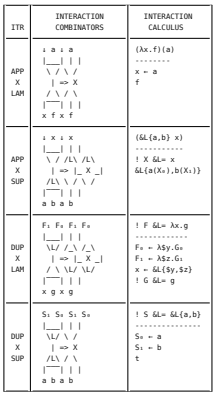
When labels differ, DUP-SUP commutes instead of annihilating:

```
! x &L= &R{a, b}; t
-------------------- DUP-SUP (L ≠ R)
! A &L= a;
! B &L= b;
x₀ ← &R{A₀, B₀}
x₁ ← &R{A₁, B₁}
t
```

These four rules form a complete system. Every other interaction in HVM is an extension for practical constructs: numbers, constructors, pattern-matching.

RELATION TO INTERACTION COMBINATORS

The Interaction Calculus is similar to Interaction Combinators, the graph rewriting model described by Lafont (1997). This similarity can be visualized as:



It can also be seen as a completion of the λ-Calculus, giving a computational meaning to previously undefined expressions: applying a pair and projecting a lambda. The Interaction Calculus provides sensible reduction rules for these, inspired by its Interaction Combinator equivalence: applying a pair (superposition) distributes over both elements (APP-SUP), and duplicating a lambda creates two lambdas with a superposed bound variable (DUP-LAM). This makes every possible interaction well-defined.

COMPLETE EXAMPLES

Here's an example demonstrating optimal sharing. We duplicate a lambda containing `(2 + 2)` and apply the copies to different arguments:

```
! F &= (λx. λy. ! z &= x; ((z₀ + z₁), y) 2);
(F₀(10), F₁(20))
---------------------------------------------- APP-LAM
! z &= 2;
! F &= λy.((z₀ + z₁), y);
(F₀(10), F₁(20))
---------------------------------------------- DUP-NUM
! F &= λy.((2 + 2), y);
(F₀(10), F₁(20))
---------------------------------------------- DUP-LAM
! B &= ((2 + 2), &{y0, y1});
((λy0.B₀ 10), (λy1.B₁ 20))
---------------------------------------------- APP-LAM (x2)
! B &= ((2 + 2), &{10, 20});
(B₀, B₁)
---------------------------------------------- DUP-CTR
! H &= (2 + 2);
! T &= &{10, 20};
((H₀, T₀), (H₁, T₁))
---------------------------------------------- OP2-NUM (shared!)
! H &= 4;
! T &= &{10, 20};
((H₀, T₀), (H₁, T₁))
---------------------------------------------- DUP-NUM, DUP-SUP
((4, 10), (4, 20))
```

Notice that `(2 + 2)` was computed only **once**, even though the lambda was duplicated and each copy was applied to a different argument. The result `4` flowed to both copies through the DUP-NUM interaction.

CHURCH 2^2

Here's a more complex example - computing 2^2 using Church numerals:

```
(λf. ! F &L= f; λx.(F₀ (F₁ x)) λg. ! G &K= g; λy.(G₀ (G₁ y)))
---------------------------------------------------------- APP-LAM
! F &L= λg. ! G &K= g; λy.(G₀ (G₁ y));
λx.(F₀ (F₁ x))
---------------------------------------------------------- DUP-LAM
! G &K= &L{g0, g1};
! F &L= λy.(G₀ (G₁ y));
λx.((λg0.F₀) (λg1.F₁ x))
---------------------------------------------------------- APP-LAM
! G &K= &L{(λg1.F₁ x), g1};
! F &L= λy.(G₀ (G₁ y));
λx.F₀
---------------------------------------------------------- DUP-LAM
! G &K= &L{(λg1.λy1.F₁ x), g1};
! F &L= (G₀ (G₁ &L{y0, y1}));
λx.λy0.F₀
---------------------------------------------------------- DUP-SUP (L ≠ K)
! A &K= (λg1.λy1.F₁ x);
! B &K= g1;
! F &L= (&L{A₀, B₀} (&L{A₁, B₁} &L{y0, y1}));
λx.λy0.F₀
---------------------------------------------------------- APP-SUP
! A &K= (λg1.λy1.F₁ x);
! B &K= g1;
! U &L= (&L{A₁, B₁} &L{y0, y1});
! F &L= &L{(A₀ U₀), (B₀ U₁)};
λx.λy0.F₀
---------------------------------------------------------- DUP-SUP (L = L)
! A &K= (λg1.λy1.(B₀ U₁) x);
! B &K= g1;
! U &L= (&L{A₁, B₁} &L{y0, y1});
λx.λy0.(A₀ U₀)
---------------------------------------------------------- APP-LAM
! A &K= λy1.(B₀ U₁);
! B &K= x;
! U &L= (&L{A₁, B₁} &L{y0, y1});
λx.λy0.(A₀ U₀)
---------------------------------------------------------- DUP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= (&L{λy11.A₁, B₁} &L{y0, &K{y10, y11}});
λx.λy0.((λy10.A₀) U₀)
---------------------------------------------------------- APP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= (&L{λy11.A₁, B₁} &L{y0, &K{U₀, y11}});
λx.λy0.A₀
---------------------------------------------------------- APP-SUP
! A &K= (B₀ U₁);
! B &K= x;
! V &L= &L{y0, &K{U₀, y11}};
! U &L= &L{((λy11.A₁) V₀), (B₁ V₁)};
λx.λy0.A₀
---------------------------------------------------------- DUP-SUP (L = L), APP-LAM
! A &K= (B₀ U₁);
! B &K= x;
! U &L= &L{(A₁ y0), (B₁ &K{U₀, y11})};
λx.λy0.A₀
---------------------------------------------------------- DUP-SUP (L = L)
! A &K= (B₀ (B₁ &K{(A₁ y0), y11}));
! B &K= x;
λx.λy0.A₀
```

At this point, the computation is done. The result is a lambda `λx.λy0.A₀` connected to a small network of nodes. This compressed form represents the answer but is hard to read directly. To see the familiar Church numeral 4, we can **collapse** the Interaction Calculus term back into a proper λ-Term, by applying 2 extra interactions, DUP-VAR and DUP-APP:

```
...
-------------------------------- DUP-VAR
! A &K= (x (x &K{(A₁ y0), y11}));
λx.λy0.A₀
-------------------------------- DUP-APP, DUP-VAR
! X &K= (x &K{((x X₁) y0), y11});
λx.λy0.(x X₀)
-------------------------------- DUP-APP, DUP-VAR
! Y &K= &K{((x (x Y₁)) y0), y11};
λx.λy0.(x (x Y₀))
-------------------------------- DUP-SUP (K = K)
λx.λy0.(x (x ((x (x y11)) y0)))
-------------------------------- APP-LAM (x2)
λx.λy0.(x (x (x (x y0))))
```

The Church numeral 2 (`λf.λx.(f (f x))`) was applied to itself, yielding 4 (`λf.λx.(f (f (f (f x))))`). Despite temporarily escaping variables, the system correctly computes the result. This is the kind of symbolic computation where sharing matters most, and is often used as a benchmark in the literature. HVM completes it in 14 interactions, which is optimal.

L.5 · `HVM4/docs/hvm/core.md`

HVM Core Terms

This document defines the core surface syntax for HVM terms. These terms are parsed into static (book) terms and later instantiated as dynamic terms during execution. See `docs/hvm/memory.md` for the dynamic/static memory layout.

GRAMMAR

```
Term ::=
  | Var  Name                                    -- variable
  | Dp0  Name "₀"                                -- first dup variable
  | Dp1  Name "₁"                                -- second dup variable
  | Ref  "@" Name                                -- reference
  | Nam  "^" Name                                -- name (stuck head)
  | Dry  "^" "(" Term " " Term ")"               -- dry (stuck application)
  | Era  "&{}"                                   -- erasure
  | Sup  "&" Label "{" Term "," Term "}"         -- superposition
  | Dup  "!" Name "&" Label "=" Term ";" Term    -- duplication term
  | Ctr  "#" Name "{" Term,* "}"                 -- constructor
  | Mat  "λ" "{" "#" Name ":" Term ";" Term "}"  -- pattern match
  | Swi  "λ" "{" Num ":" Term ";" Term "}"       -- number switch
  | Use  "λ" "{" Term "}"                        -- use (unbox)
  | Lam  "λ" Name "." Term                       -- lambda
  | App  "(" Term " " Term ")"                   -- application
  | Num  integer                                 -- number literal
  | Op2  "(" Term Oper Term ")"                  -- binary operation
  | Eql  "(" Term "==" Term ")"                  -- equality test
  | And  "(" Term ".&." Term ")"                 -- short-circuit AND
  | Or   "(" Term ".|." Term ")"                 -- short-circuit OR
  | DSu  "&" "(" Term ")" "{" Term "," Term "}"  -- dynamic superposition
  | DDu  "!" Name "&" "(" Term ")" "=" Term ";" Term -- dynamic duplication term
  | Inc  "↑" Term                                -- priority wrapper
  | Alo  "@" "{" Name,* "}" Term                 -- allocation
  | Uns  "!" "$" "{" Name "," Name "}" ";" Term  -- unscoped binding

Name  ::= [_A-Za-z0-9]+
Label ::= Name
Oper  ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||"
        | "^" | "~" | "<<" | ">>" | "==" | "!="
        | "<" | "<=" | ">" | ">="
```

NOTES

- Variables are affine: each variable is used at most once.
- Variables are global: a variable can occur outside its binder's lexical scope.
- Labels determine how duplications and superpositions interact; equal labels annihilate, different labels commute.
- Surface sugar accepts `λ$x. body` as an unscoped lambda, equivalent to `! f = λ x ; f(body)` with fresh `f` (see `docs/hvm/syntax.md`).

L.6 · HVM4/docs/hvm/memory.md

HVM Memory Layout

HVM represents every term as a 64-bit word. The same layout is used for dynamic terms (mutable heap entries) and static terms (immutable book entries).

TERM POINTER LAYOUT (64-BIT)

```
TAG (8 bits) | EXT (24 bits) | VAL (32 bits)
```

- `TAG` : constructor variant (APP, LAM, SUP, etc.); the high bit is the SUB flag.
- `EXT` : metadata (dup label, ctor/ref name, level, op code, flags).
- `VAL` : payload (heap loc, immediate, or 0).

TERM MEMORY TABLE

TERM	TAG	EXT	VAL	NOTES
application	APP	0	node: [func, arg]	dynamic or static
linked variable	VAR	0	lam body or subst	dynamic only; follows SUB cells
linked duplicate 0	DP0	label	dup expr or subst	dynamic only; twin of DP1
linked duplicate 1	DP1	label	dup expr or subst	dynamic only; twin of DP0
linked lambda	LAM	level+flags	node: [body]	dynamic only; binder for VAR
quoted lam	LAM	level+flags	node: [body]	quoted; level stored in EXT
superposition	SUP	label	node: [tm0, tm1]	dynamic or static
linked duplication term	DUP	label	node: [expr, body]	dynamic or static; binder for DP0/DP1
quoted duplication term	DUP	label	node: [expr, body]	expr typically `&L{BJ0,BJ1}` in quoted mode
number literal	NUM	0	unboxed u32	
constructor arity 0	C00	ctor name	0	tag encodes arity
constructor arity N	C01-16	ctor name	node: [field0..fieldN-1]	N = tag - C00
pattern match	MAT	ctor name	node: [hit, miss]	match on constructor
number switch	SWI	number literal	node: [zero, succ]	parser/printer distinction from MAT
use (unbox)	USE	0	node: [fun]	forces argument, then applies
binary op	OP2	op code	node: [lhs, rhs]	EXT stores OP_ADD, OP_MUL, etc.
equality	EQL	0	node: [lhs, rhs]	structural equality
logical AND	AND	0	node: [lhs, rhs]	short-circuit AND
logical OR	OR	0	node: [lhs, rhs]	short-circuit OR
priority wrapper	INC	0	node: [term]	collapse priority wrapper
name literal	NAM	name id	0	literal ^name

TERM	TAG	EXT	VAL	NOTES
stuck application	DRY	0	node: [fun, arg]	literal ^(f x)
reference	REF	name id	0	book reference @name
allocation	ALO	bind list length	direct or packed pair	len=0: VAL=book term; len>0: VAL->(low32=book term, high32=bind list head)
unscoped binding	UNS	0	node: [body]	helper to construct unscoped lams
wildcard	ANY	0	0	duplicates itself, equals anything
quoted lam var	BJV	0	de Bruijn level	quoted lam-bound var
quoted dup var 0	BJ0	label	de Bruijn level	quoted dup-bound var
quoted dup var 1	BJ1	label	de Bruijn level	quoted dup-bound var
dynamic superposition	DSU	0	node: [lab, tm0, tm1]	label computed dynamically
dynamic duplication	DDU	0	node: [lab, val, body]	label computed dynamically

SUBSTITUTION CELLS (SUB BIT)

When a linked binder interacts (APP-LAM, DUP-*), its body or expr slot is replaced by a substitution term with the SUB bit set. Any VAR/DP0/DP1 that points to that heap location must read the substitution instead of the original binder. This is how linked variables resolve their binders without extra maps.

DUP NODES VS DUP TERMS

A DUP term is the syntactic binder `[expr, body]` with tag `DUP`. A DUP node is the shared expr slot that DP0/DP1 point to; it has no body and no parent, and it is substituted when a duplication interaction occurs.

LINKED VS QUOTED BINDERS

Linked binders are dynamic lams/dups whose vars point to heap locations. Linked LAM uses `EXT = level | flags` and its VARs point to the body slot. Linked DUP uses DP0/DP1 vars that point to the shared expr slot.

Quoted binders are terms encoded with de Bruijn levels in `EXT` (and for BJ0/BJ1, the dup label). Their variables are BJV/BJ0/BJ1 indices, so there are no heap links and no interaction with APP-LAM or DUP-SUP.

During collapse, linked LAM/DUP binders are converted into quoted LAM/DUP binders, and free VAR/DP0/DP1 become BJV/BJ0/BJ1 at the current level. This is used to turn interaction nets into full lambda terms.

DYNAMIC VS STATIC TERMS

Dynamic terms live on the heap and are mutable: binders can be replaced by substitutions, and DUP nodes evolve as duplication is forced. Static terms live in the book and are immutable; they store de Bruijn levels and are never updated in place.

Allocation (ALO) terms bridge the two: they reference a static book term and a bind list, lazily expanding one layer into a dynamic term when forced. This keeps static definitions compact while still allowing dynamic sharing during execution.

ALO RUNTIME ENCODINGS

- `ALO` with `len == 0` (empty substitution list) is stored directly:
  - `ALO.val = book_term_loc`
  - no extra heap allocation for an ALO pair node.
- `ALO` with `len > 0` stores one packed ALO pair word at `ALO.val`:
  - `high 32 bits = bind list head location`
  - `low 32 bits = book term location`

This keeps `len > 0` ALO pairs at one heap word while preserving the full 32-bit location range for both bind-list heads and book terms.

ALO BIND-LIST NODES

For `len > 0`, the bind list is a linked list of 2-slot nodes:

- `node[0]`: bound term cell (the substitution target term lives here).
- `node[1]`: `NUM(next_node_loc)`; `0` means end of list.

This means the pointer used by variables/copiers is the bind-node location itself (slot `0`), not a separate allocation.

- In `ALO-LAM`, `node[0]` is the lambda body slot.
- In `ALO-DUP`, `node[0]` is the shared DUP expr slot.

LAM EXT FLAGS

- `LAM_ERA_MASK` (0x800000): binder is unused in lambda body (erasing lambda).

L.7 · HVM4/docs/hvm/syntax.md

HVM Parser Syntax

Desugarings use the core surface syntax from `docs/hvm/core.md`. Core application is written there as `(f x)`, so this doc uses that notation for desugaring only; the parser itself does not accept whitespace application.

PARSING MODEL (HOW TERMS GROUP)

- Base unit: an atom.
- Postfix call: `f(...)` binds tightest.
- `<>` binds tighter than infix operators.
- `===`, `.&.`, `.|.` bind tighter than infix operators (same precedence class as `==`, `&&`, `||`, but separate nodes).
- Infix operators are left-associative, by the precedence table below.
- No whitespace application: `f x` is not an application; use `f(x)`.

LEXICAL RULES

- Whitespace: space, tab, newline, `\r`.
- Line comments: `// ...`.
- Names: `_a-zA-Z0-9$`.
  - A term starting with a digit is parsed as a number literal, not a name.

ATOMS

- Variable: `x`
- Dup variables: `x₀` or `x₁` (see “Dup variables”)
- Grouping: `(term)`
- Constructor: `#Name` or `#Name{a,b,c}` (`,` and `;` are equivalent, trailing separator ok)
- Reference: `@Name`
- Name head: `^Name`
- Dry app: `^(f x)`
- Priority wrapper: `↑atom`
- Wildcard: `*`
- Lambda / dup / sup / match / literals (see below)

APPLICATION

- `f(a, b, c)` desugars to `((f a) b) c`.
  - `,` and `;` are equivalent, and separators are optional: `f(a b c)` is the same.
  - `f()` is allowed and desugars to `f`.

LITERALS

NUMBERS

- `123` is a numeric literal.

NATURALS (PEANO)

Using built-in constructors `#Zer` and `#Suc`:

- `Nn` desugars to `#Suc{...#Suc{#Zer}...}` (N times)
- `Nn+term` desugars to `#Suc{...#Suc{term}...}` (N times)
  - `n+` must be contiguous; `3n + x` is not a natural literal.

Examples:

```
3n    => #Suc{#Suc{#Suc{#Zer}}}
2n+xs => #Suc{#Suc{xs}}
```

CHARACTERS

- `'c'` desugars to `#Chr{#<codepoint>}`.

Escapes:

- Standard escapes: `\n`, `\t`, `\r`, `\0`, `\\`, `\'`, `\"`.
- Other backslashes escape the next byte verbatim.
- Otherwise UTF-8 sequences are decoded into a numeric codepoint.

STRINGS

Strings are lists of `#Chr`:

- `"str"` desugars to `#Con{#Chr{#s}, #Con{#Chr{#t}, #Con{#Chr{#r}, #Nil}}}`.

Same escape rules as characters.

LISTS

Lists use built-in `#Nil` and `#Con`:

- `[]` desugars to `#Nil`.
- `[a, b, c]` desugars to `#Con{a, #Con{b, #Con{c, #Nil}}}`.
  - `,` and `;` are equivalent; trailing separator allowed.
- `a <> b` desugars to `#Con{a, b}` (cons sugar).

LAMBDAS AND BINDERS

LAMBDA

- `λx. body` is core `Lam`.
- `λx,y,z. body` desugars to `λx. λy. λz. body`.
  - `,` and `;` are equivalent between binders.
- `λ$x. body` is an unscoped lambda binder (see “Unscoped lambda (UNS)”).

CLONED BINDERS (AUTO-DUP)

Prefix `&` allows multiple uses; the parser inserts dup nodes.

- `λ&x. body` desugars to `λx. body` plus auto-duplication of `x`.
- Auto-duplication expands a cloned variable with N uses into N-1 nested dups using fresh labels. Example for three uses: `body[x,x,x]` becomes `!d0&=x; !d1&=d0₁; body[d0₀,d1₀,d1₁]`.

INLINE DUPLICATION ON LAMBDAS

Short form for inserting a dup on the bound variable:

- `λx&L. body` desugars to `λx. !x&L = x; body`.
- `λx&. body` uses a fresh label.
- `λx&(lab). body` desugars to `λx. !x&(lab) = x; body`.

All of these combine with cloning: `λ&x&L. ...`.

DUPLICATION AND LET (!)

LET SUGAR

- `!x = v; body` desugars to `((λx. body) v)`.
- `!!x = v; body` desugars to `((λ{λx. body}) v)` (strict let).
- Cloned forms `!&x = ...` and `!!&x = ...` allow multiple uses of `x` and insert auto-dup as needed.

DUPLICATION (STATIC LABEL)

- `!x&L = v; body` is core `Dup`.
- `!x& = v; body` uses a fresh label.
- Optional `,` or `;` after `v` is allowed.
- Cloned dup binder: `!&x&L = v; body` (or `!&x& = ...`).

DUP DESTRUCTURING SUGAR

Short form for binding each side of a duplicated value as a plain variable:

- `! &L{x,y} = v; body` desugars like `!x_y&L = v; body[x := x_y₀, y := x_y₁]`.
- `! &{x,y} = v; body` uses a fresh label.
- `! &(lab){x,y} = v; body` uses a dynamic label.
- Branch binders can be cloned independently: `! &L{&x,y} = v; body`.
- `,` and `;` are equivalent between branch binders; trailing separator allowed.

DUPLICATION (DYNAMIC LABEL)

- `!x&(lab) = v; body` is core `DDu`.
- `!&x&(lab) = v; body` is the cloned form.

UNSCOPED LAMBDA (UNS)

- `! f = λ x ; body` desugars to `!${f,x}; body`.
- `λ$x. body` desugars to `! f = λ x ; f(body)` with fresh `f`.
  - `,` and `;` are allowed between binders: `λ$x,y. body` (x unscoped, y scoped).
  - The unscoped variable is referenced as plain `x` inside the body.

SUPERPOSITION (&) AND FORK

SUPERPOSITION AND ERASURE

- `&L{a, b}` is core `Sup`.
- `&(lab){a, b}` is core `DSu`.
- `&{}` is core `Era`.
  - `,` and `;` are equivalent; trailing separator allowed.

AUTO-FORK SUGAR

```
&L!{A;B}
```

desugars to `&L[&x,&y,&z,...]{A;B}` where `x,y,z,...` are all variables currently in scope (lambda/let bindings and fork bindings, excluding raw dup bindings that require subscripts). All captured variables are automatically cloned. Dynamic labels are supported: `&(lab)![A;B}`.

This is a convenience for the common pattern of forking the entire context into a superposition's two branches.

FORK SUGAR

```
&Lλx,y{A;B}
```

desugars to:

```
λx&L. λy&L. &L{A', B'}
```

where `A'` uses x_0, y_0 and `B'` uses x_1, y_1.

Inside each branch you can omit $_0$ / $_1$ and the parser chooses the side automatically. You can also prefix branches with $\&_0$: or $\&_1$: to force the side for all un-subscripted dup variables in that branch. `,` and `;` are equivalent between branches; trailing separator allowed.

Dynamic labels are supported: `&(lab)λx,y{A;B}`.

MATCH / SWITCH / USE (`λ{...}`)

ERASURE

- `λ{}` desugars to `&{}`.

USE / UNBOX

- `λ{term}` is core `Use`.
- `λ{_: term}` also desugars to `λ{term}` if it is the only entry.

PATTERN MATCH AND NUMERIC SWITCH

Syntax:

```
λ{
  pat1: t1;
  pat2: t2;
  ...
  _ : default
}
```

Patterns:

- `#Name:` match constructor `#Name`
- `[]:` match `#Nil`
- `<>:` match `#Con`
- `0n:` match `#Zer`
- `1n+:` match `#Suc` (digits before `n+` are ignored)
- `N:` numeric switch on literal `N`
- `_:` default branch

Desugaring is right-nested:

```
λ{#A: t1; #B: t2; d}
=> λ{#A: t1; λ{#B: t2; d}}

λ{0: t0; 1: t1; d}
=> λ{0: t0; λ{1: t1; d}}
```

`,` and `;` are equivalent between cases; trailing separator allowed, including after the final default. The final default can be written as `_ : d` or as a bare `d`.

NAMES AND REFERENCES

- `@Name` is a book reference.
- `^Name` is a stuck name head.
- `^(f x)` is a dry application.

PRIORITY WRAPPER AND WILDCARD

- `↑atom` is core `Inc` (binds to a single atom).
- `*` is core `Any`.

INFIX OPERATORS

All infix operators are left-associative.

Precedence (higher binds tighter):

- 8: `^`
- 7: `* / %`
- 6: `+ -`
- 5: `<< >>`
- 4: `< <= > >=`
- 3: `== !=`
- 2: `&&`
- 1: `||`

Notes:

- `~` parses as `OP_NOT` but is infix. Evaluation ignores the left operand and uses only the right one (`~b`). For unary not, write `0 ~ x`.

STRUCTURAL EQUALITY AND SHORT-CIRCUIT OPS

These are separate nodes (not `Op2`):

- `a === b` structural equality
- `a .&. b` short-circuit AND
- `a .|. b` short-circuit OR

DUP VARIABLES AND SUBSCRIPTS

- Variables bound by `Dup` (or inline `λx&L`) must be referenced as x_0 or x_1.
- Inside fork branches, the parser can auto-select the side if you omit the subscript.
- Dynamic dup variables (`!x&(lab)=...`) always require a subscript.

NOT PARSED DIRECTLY

Some internal terms have print forms but are not parsed directly (for example `@{...}` for `ALO`). The list above reflects what the parser accepts today.

L.8 · HVM4/docs/hvm/collapser.md

Collapser (CNF Readback)

Interaction Calculus is affine and uses explicit duplication (DUP) and superposition (SUP). That is great for optimal sharing but hard for humans to read. The collapser is the readback procedure: it turns one IC term into a stream of ordinary lambda terms, i.e., collapsed normal form (CNF).

Two insights make this possible:

- Quoting removes DUPs. When a branch is ready to print, we run `cnf(term)`. Linked vars become quoted vars (BJV/BJO/BJ1). DUP interactions are defined on quoted dup vars, so DUP nodes are cloned away and disappear from the printed term.
- Lifting removes SUPs. We lift the first SUP to the top and enumerate its branches. Same-label SUPs annihilate pairwise; different-label SUPs commute and create a cross product of branches.

ALGORITHM (AS IMPLEMENTED)

- `cnf` (`src/hvm.c`, CNF section): reduce to WNF, then lift the first SUP to the top and return immediately. ERA propagates upward; INC is left in place for the flattener.
- `eval_collapse` (`src/hvm.c`, Eval section): breadth-first traversal with a sequential priority queue. Lower numeric keys are popped first; SUP increases key, INC decreases key. Branches with the same key are popped FIFO for deterministic ordering. When a branch has no SUP, it prints `cnf(term)`.

LABEL BEHAVIOR (PAIRWISE VS CROSS PRODUCT)

Different labels commute (cross product):

```
[&A{1,2}, &B{3,4}]
```

```
[1,3]
[1,4]
[2,3]
[2,4]
```

Same labels annihilate pairwise:

```
[&A{1,2}, &A{3,4}]
```

```
[1,2]
[3,4]
```

WHERE TO LOOK

- `src/hvm.c`, `CNF` section: SUP lifting rules.
- `src/hvm.c`, `Eval` section: branch enumeration + SNF quoting for output.

L.9 · HVM4/docs/hvm/interactions/alo_cop.md

ALO-COP

```
@{s} n₀
------- ALO-DP0
s[n]₀ or n₀ when substitution missing (n is a de Bruijn level)

@{s} n₁
------- ALO-DP1
s[n]₁ or n₁ when substitution missing (n is a de Bruijn level)
```

L.10 · HVM4/docs/hvm/interactions/alo_dry.md

ALO-DRY

```
@{s} ^(f x)
------------- ALO-DRY
^(@{s}f @{s}x)
```

L.11 · HVM4/docs/hvm/interactions/alo_dup.md

ALO-DUP

```
@{s} ! x &L = v; t
------------------ ALO-DUP
x' ← fresh
! x' &L = @{s} v
@{x',s} t
```

L.12 · HVM4/docs/hvm/interactions/alo_lam.md

ALO-LAM

```
@{s} λx.f
------------ ALO-LAM
x' ← fresh
λx'.@{x',s}f
```

L.13 · HVM4/docs/hvm/interactions/alo_nam.md

ALO-NAM

```
@{s} ^n
------- ALO-NAM
^n
```

L.14 · HVM4/docs/hvm/interactions/alo_nod.md

ALO-NOD

```
@{s} T{a,b,...}
---------------- ALO-NOD
T{@{s}a, @{s}b, ...}
```

L.15 · HVM4/docs/hvm/interactions/alo_var.md

ALO-VAR

```
@{s} n
------ ALO-VAR
s[n] or n when substitution missing (n is a de Bruijn level)
```

L.16 · HVM4/docs/hvm/interactions/and_era.md

AND-ERA

```
(&{} .&. b)
----------- AND-ERA
&{}
```

L.17 · HVM4/docs/hvm/interactions/and_inc.md

AND-INC

```
(↑a & b)
--------- AND-INC
↑(a & b)
```

L.18 · HVM4/docs/hvm/interactions/and_num.md

AND-NUM

```
(#0 .&. b)
---------- AND-ZER
#0

(#n .&. b)   [n ≠ 0]
------------------- AND-ONE
b
```

L.19 · HVM4/docs/hvm/interactions/and_sup.md

AND-SUP

```
(&L{a0,a1} .&. b)
------------------------- AND-SUP
! B &L = b
&L{(a0 .&. B₀), (a1 .&. B₁)}
```

L.20 · HVM4/docs/hvm/interactions/app_ctr.md

APP-CTR

```
(ctr a)
----------- APP-CTR (stuck)
^(ctr a)
```

L.21 · HVM4/docs/hvm/interactions/app_dry.md

APP-DRY

```
(^(f x) a)
----------- APP-DRY
^(^(f x) a)
```

L.22 · HVM4/docs/hvm/interactions/app_era.md

APP-ERA

```
(&{} a)
------- APP-ERA
&{}
```

L.23 · HVM4/docs/hvm/interactions/app_inc.md

APP-INC

```
(↑f x)
-------- APP-INC
↑(f x)
```

L.24 · HVM4/docs/hvm/interactions/app_lam.md

APP-LAM

```
(λx.f a)
-------- APP-LAM
x ← a
f
```

L.25 · HVM4/docs/hvm/interactions/app_mat_ctr.md

APP-MAT-CTR

```
(λ{#K:h; m} #K{a,b})
-------------------- APP-MAT-CTR-MAT
(h a b)

(λ{#K:h; m} #L{a,b})
-------------------- APP-MAT-CTR-MIS
(m #L{a,b})
```

L.26 · HVM4/docs/hvm/interactions/app_mat_num.md

APP-MAT-NUM

```
(λ{#a:h; m} #a)
--------------- APP-MAT-NUM-MAT
h

(λ{#a:h; m} #b)
--------------- APP-MAT-NUM-MIS
(m #b)
```

L.27 · HVM4/docs/hvm/interactions/app_mat_sup.md

APP-MAT-SUP

```
(λ{#K:h; m} &L{a,b})
-------------------- APP-MAT-SUP
! H &L = h
! M &L = m
&L{(λ{#K:H₀; M₀} a)
  ,(λ{#K:H₁; M₁} b)}
```

L.28 · HVM4/docs/hvm/interactions/app_nam.md

APP-NAM

```
(name a)
--------- APP-NAM
^(name a)
```

L.29 · HVM4/docs/hvm/interactions/app_sup.md

APP-SUP

```
(&L{f,g} a)
----------------- APP-SUP
! A &L = a
&L{(f A₀),(g A₁)}
```

L.30 · HVM4/docs/hvm/interactions/ddu_era.md

DDU-ERA

```
! X &(&{}) = v; b
----------------- DDU-ERA
&{}
```

L.31 · HVM4/docs/hvm/interactions/ddu_inc.md

DDU-INC

```
! X &(↑x) = v; b
---------------- DDU-INC
↑(! X &(x) = v; b)
```

L.32 · HVM4/docs/hvm/interactions/ddu_num.md

DDU-NUM

```
! X &(#n) = v; b
---------------- DDU-NUM
! X &n = v
b(X₀, X₁)
```

L.33 · HVM4/docs/hvm/interactions/ddu_sup.md

DDU-SUP

```
! X &(&L{x,y}) = v; b
------------------------------ DDU-SUP
! V &L = v
! B &L = b
&L{! X &(x) = V₀; B₀, ! X &(y) = V₁; B₁}
```

L.34 · HVM4/docs/hvm/interactions/dsu_era.md

DSU-ERA

```
&(&{}){a, b}
------------ DSU-ERA
&{}
```

L.35 · HVM4/docs/hvm/interactions/dsu_inc.md

DSU-INC

```
&(↑x){a, b}
------------ DSU-INC
↑(&(x){a, b})
```

L.36 · HVM4/docs/hvm/interactions/dsu_num.md

DSU-NUM

```
&(#n){a, b}
----------- DSU-NUM
&n{a, b}
```

L.37 · HVM4/docs/hvm/interactions/dsu_sup.md

DSU-SUP

```
&(&L{x,y}){a, b}
------------------------- DSU-SUP
! A &L = a
! B &L = b
&L{&(x){A₀,B₀}, &(y){A₁,B₁}}
```

L.38 · HVM4/docs/hvm/interactions/dup_dry.md

DUP-DRY

```
! X &L = ^(f x)
--------------- DUP-DRY
! F &L = f
! A &L = x
X₀ ← ^(F₀ A₀)
X₁ ← ^(F₁ A₁)
```

L.39 · HVM4/docs/hvm/interactions/dup_lam.md

DUP-LAM

```
! F &L = λx.f
---------------- DUP-LAM
F₀ ← λ$x0.G₀
F₁ ← λ$x1.G₁
x  ← &L{$x0,$x1}
! G &L = f
```

L.40 · HVM4/docs/hvm/interactions/dup_nam.md

DUP-NAM

```
! X &L = name
------------ DUP-NAM
X₀ ← name
X₁ ← name
```

L.41 · HVM4/docs/hvm/interactions/dup_nod.md

DUP-NOD

```
! X &L = T{a,b,...}
------------------- DUP-NOD
! A &L = a
! B &L = b
...
X₀ ← T{A₀,B₀,...}
X₁ ← T{A₁,B₁,...}
```

L.42 · HVM4/docs/hvm/interactions/dup_sup.md

DUP-SUP

```
! X &L = &R{a,b}
---------------- DUP-SUP
if L == R:
  X₀ ← a
  X₁ ← b
else:
  ! A &L = a
  ! B &L = b
  X₀ ← &R{A₀,B₀}
  X₁ ← &R{A₁,B₁}
```

L.43 · HVM4/docs/hvm/interactions/eql_any.md

EQL-ANY

```
(* === b)
----------- EQL-ANY-L
1

(a === *)
----------- EQL-ANY-R
1
```

L.44 · HVM4/docs/hvm/interactions/eql_ctr.md

EQL-CTR

```
(#K{a0,a1...} === #K{b0,b1...})  (same tag)
---------------------------------------- EQL-CTR-MAT
For SUC (1n+): ↑(pred === pred)
For CON (<>): ↑((head === head) & ↑(tail === tail))
Others: (a0 === b0) & (a1 === b1) & ...

(#K{...} === #L{...})  (different tag)
------------------------------------ EQL-CTR-MIS
#0
```

L.45 · HVM4/docs/hvm/interactions/eql_dry.md

EQL-DRY

```
(^(af ax) === ^(bf bx))
----------------------- EQL-DRY
(af === bf) & (ax === bx)
```

L.46 · HVM4/docs/hvm/interactions/eql_era.md

EQL-ERA

```
(&{} === b)
---------- EQL-ERA-L
&{}

(a === &{})
---------- EQL-ERA-R
&{}
```

L.47 · HVM4/docs/hvm/interactions/eql_inc.md

EQL-INC

```
(↑a === b)
---------- EQL-INC-L
↑(a === b)

(a === ↑b)
---------- EQL-INC-R
↑(a === b)
```

L.48 · HVM4/docs/hvm/interactions/eql_lam.md

EQL-LAM

```
(λax.af === λbx.bf)
------------------ EQL-LAM
X := fresh_nam()
ax ← X
bx ← X
af === bf
```

L.49 · HVM4/docs/hvm/interactions/eql_mat.md

EQL-MAT

```
(λ{#K:ah;am} === λ{#K:bh;bm})  (same tag)
---------------------------------------- EQL-MAT-MAT
(ah === bh) & (am === bm)

(λ{#K:...} === λ{#L:...})  (different tag)
---------------------------------------- EQL-MAT-MIS
#0
```

L.50 · HVM4/docs/hvm/interactions/eql_nam.md

EQL-NAM

L.51 · HVM4/docs/hvm/interactions/eql_num.md

EQL-NUM

```
(#a === #b)
------------ EQL-NUM
#(a == b)
```

L.52 · HVM4/docs/hvm/interactions/eql_sup.md

EQL-SUP

```
(&L{a0,a1} === b)
---------------------- EQL-SUP-L
! B &L = b
&L{(a0 === B₀), (a1 === B₁)}

(a === &L{b0,b1})
---------------------- EQL-SUP-R
! A &L = a
&L{(A₀ === b0), (A₁ === b1)}
```

L.53 · HVM4/docs/hvm/interactions/eql_use.md

EQL-USE

```
(λ{af} === λ{bf})
----------------- EQL-USE
af === bf
```

L.54 · HVM4/docs/hvm/interactions/mat_inc.md

MAT-INC

```
(λ{...} ↑x)
------------ MAT-INC
↑(λ{...} x)
```

L.55 · HVM4/docs/hvm/interactions/op2_era.md

OP2-ERA

```
@@opr(&{}, y)
------------- OP2-ERA
&{}
```

L.56 · HVM4/docs/hvm/interactions/op2_inc.md

OP2-INC

```
(↑x op y)
---------- OP2-INC-X
↑(x op y)

(#n op ↑y)
---------- OP2-INC-Y
↑(#n op y)
```

L.57 · HVM4/docs/hvm/interactions/op2_num_era.md

OP2-NUM-ERA

```
(x op &{}) where x is NUM
------------- OP2-NUM-ERA
&{}
```

L.58 · HVM4/docs/hvm/interactions/op2_num_num.md

OP2-NUM-NUM

```
(#a op #b)
------------- OP2-NUM-NUM
#(a opr b)
```

L.59 · HVM4/docs/hvm/interactions/op2_num_sup.md

OP2-NUM-SUP

```
(x op &L{a,b}) where x is NUM
------------------------ OP2-NUM-SUP
! X &L = x
&L{(X₀ op a), (X₁ op b)}
```

L.60 · HVM4/docs/hvm/interactions/op2_sup.md

OP2-SUP

```
@@opr(&L{a,b}, y)
------------------------ OP2-SUP
! Y &L = y
&L{@@opr(a,Y₀), @@opr(b,Y₁)}
```

L.61 · HVM4/docs/hvm/interactions/or_era.md

OR-ERA

```
(&{} .|. b)
---------- OR-ERA
&{}
```

L.62 · HVM4/docs/hvm/interactions/or_inc.md

OR-INC

```
(↑a | b)
--------- OR-INC
↑(a | b)
```

L.63 · HVM4/docs/hvm/interactions/or_num.md

OR-NUM

```
(#0 .|. b)
---------- OR-ZER
b

(#n .|. b)   [n ≠ 0]
------------------- OR-ONE
#1
```

L.64 · HVM4/docs/hvm/interactions/or_sup.md

OR-SUP

```
(&L{a0,a1} .|. b)
------------------------- OR-SUP
! B &L = b
&L{(a0 .|. B₀), (a1 .|. B₁)}
```

L.65 · HVM4/docs/hvm/interactions/uns.md

UNS

```
! ${f, v}; t
------------ WNF UNS
t(λy.λ$x.y, $x)
```

L.66 · HVM4/docs/hvm/interactions/use_era.md

USE-ERA

```
(λ{f} &{})
---------- USE-ERA
&{}
```

L.67 · HVM4/docs/hvm/interactions/use_inc.md

USE-INC

```
(use ↑x)
--------- USE-INC
↑(use x)
```

L.68 · HVM4/docs/hvm/interactions/use_sup.md

USE-SUP

```
(λ{f} &L{a,b})
----------------- USE-SUP
! F &L = f
&L{(λ{F₀} a), (λ{F₁} b)}
```

L.69 · HVM4/docs/hvm/interactions/use_val.md

USE-VAL

```
(λ{f} x)
--------- USE-VAL
(f x)
```

L.70 · HVM4/devs/test/ (generated listing, 218 programs)

The test programs themselves are code (each is `@main = …` followed by `//expected` lines); the March versions are in Part I. Run `./devs/test/_all_.sh` (217 tests; all pass at HEAD with `HVM_TEST_TIMEOUT_SECS=20`).

book , app_lam , autodup_1 , autodup_2 , autodup_3 , autodup_4 , autodup_bound_vars , autodup_deep , autodup_dup_dyn , autodup_dup_dyn_3each , autodup_dup_dyn_both , autodup_dup_static , autodup_dup_static_3each , autodup_dup_static_both , autodup_let , autodup_many , autodup_mat_lam , autodup_mixed , autodup_nested_lam , autodup_nightmare , autodup_outer_ref , autodup_strict , autodup_sup_mix , autodup_value_path , bind_lam_fork , bind_shadow_dup_lam , bind_shadow_lam_dup , c2_not_t , c8_k8_not_t , cadd_c1_c1_not , cadd_c1_c4 , cadd_c1_c4_not_t , cadd_c2_c1 , cadd_c2_c2 , cadd_c2_c4_not_t , cadd_c4_c1 , cadd_c4_c1_not_t , cadd_c4_c2_not_t , cadd_c4_c4 , cmul_c4_c2 , cmul_c4_c2_not_t , cmul_c4_c4 , cmul_c4_c4_not_t , cnot_not , cnot_true , collapse_0 , collapse_1 , collapse_2 , collapse_3 , collapse_4 , collapse_5 , collapse_6 , collapse_7 , collapse_8 , collapse_9 , collapse_era , collapse_era_ctr , collapse_era_deep , collapse_era_multi_ctr , collapse_era_nested , collapse_era_pair , collapse_inc_lam , collapse_supgen_mul , dup_affine_bare , dup_affine_cloned , dup_affine_co0 , dup_affine_co1 , dup_affine_shadow_ok , dup_affine_valid , dup_fresh , dup_pattern_affine , dup_pattern_basic , dup_pattern_cloned , dup_pattern_dynamic , dup_pattern_fork , dup_pattern_fresh , dup_ref , dyn_dup_nested , dyn_dup_sup , enum_bits , enum_nat , enum_primes , eql_app_fun , eql_ctr_empty , eql_ctr_f1 , eql_ctr_f2 , eql_ctr_t , eql_dup , eql_dup_lam_ctr_left , eql_lam_f , eql_lam_nested , eql_lam_nested_f , eql_lam_t , eql_list_f , eql_list_t , eql_mixed_f , eql_nat_f , eql_nat_t , eql_nested , eql_num_f , eql_num_t , eql_ref , eql_str_f , eql_str_t , eql_sup , factorial , fib , fib_small , fibo , fork_dynamic , fork_static , fuse_inc , gen_mul4 , lam_dup_auto , lam_dup_dyn , lam_dup_static , lam_dup_sup , lambda_env0 , lambda_env1 , lambda_eval , lambda_hoas , lambda_subs , let , long_name_ctor_no_collision , long_name_def , long_name_label_no_collision , long_name_match_no_collision , long_name_no_collision , long_name_undef , map , match_0 , match_1 , match_2 , match_3 , match_4 , match_5 , match_chr_escape , match_chr_switch , match_ctr_miss , match_ctr_pair , match_recursive_arg , match_u32 , mul2_4 , multiarg_cloned , multiarg_inline_dup , multiarg_mixed , multiarg_simple , nat_eql , op2_add , op2_and , op2_div , op2_dup , op2_eq_f , op2_eq_t , op2_ge_f , op2_ge_t , op2_gt_f , op2_gt_t , op2_le_f , op2_le_t , op2_lsh , op2_lt_f , op2_lt_t , op2_mod , op2_mul , op2_ne_f , op2_ne_t , op2_not , op2_or , op2_rsh , op2_sp0 , op2_sp1 , op2_sub , op2_sup , op2_xor , pair_swap , parser , partial_fallback , print_chr_escape , print_mat , print_mat_default , print_mat_lst , print_mat_nat , print_str_escape , print_swi , regression_autodup_3uses_recursive , scoped_var , scoped_var_both , separators , snf_c2_k2 , snf_unscoped , spin , sugar_chr , sugar_cons , sugar_lst , sugar_mat_lst , sugar_mat_nat , sugar_nat , sugar_nat_plus , sugar_nil , sugar_str , sugar_str_bt , sum_4 , unicode_char , unicode_cjk , unicode_emoji , unicode_greek , unicode_mixed , unicode_string , unscoped , unscoped_callcc , unscoped_lambda_direct , unscoped_mut_ref , unscoped_nested , unscoped_queue , unscoped_var , use_match_depth , use_sup_id

APPENDIX R · Audit ledger (Claude edition, 2026-11-06)

Findings against the first edition of both documents, and against upstream where it matters to readers. Severity: blocker / major / minor / polish. Every finding names its evidence; every fix is applied in this edition.

Id	Sev.	Finding	Evidence	Fix
R-01	blocker	The coursebook described HVM4 as a multi-file `clang/` runtime with AOT, FFI and threads. At HEAD that tree does not exist.	`git log 2d7f28e..6defdfc` (17 commits); `4bc2336` deletes `clang/**` and adds `src/hvm.c`; `--help`: "A sequential Interaction Calculus runtime."	Part L added; coursebook Part 8 rewritten from `src/hvm.c` at `6defdfc`.
R-02	major	Build command outdated (`cd clang && clang -O2 -o main main.c -ldl`).	upstream README/AGENTS at HEAD	`clang -O2 -o src/hvm src/hvm.c` everywhere.
R-03	major	Flags `--to-c`, `--as-c`, FFI and thread options documented; absent at HEAD.	`src/hvm --help` lists `-s -S -C[=N] -D -d -v -h` only	Coursebook §8.7–8.8; history table.
R-04	major	The `#N` printed after each collapsed answer was called a "collapse-queue key". It is the running interaction count.	`src/hvm.c:6193`: `printf(" #%llu", wnf_itrs_total())`; e.g. `11 #3`, `12 #4`, total 4	Coursebook §0.5, §6.1, §8.6.
R-05	major	`3n` was presented as sugar for `#S{#S{#S{#Z{}}}}`. They are distinct forms with distinct pattern families; mixing fails.	`@add` (chain patterns) `(1n, 2n)` → `RUNTIME_ERROR: cannot apply a number`	Coursebook §7.3 table, E7.4.
R-06	major	A runnable-looking fence (`@X` solver) was pseudo-code (`--` comments, `...`). HVM4 comments are `//`.	parse failure; fixed program → `#S{#S{#Z{}}}`, ltrs 536	Full program in §6.5; all fences re-run.
R-07	minor (upstream)	HVM4's README examples `((@add 1) 2)` and `(&{1, 2} + 10)` do not parse at HEAD.	`PARSE_ERROR … expected: ) / expected: }`	Noted in coursebook §5.7; use `f(a)` and labelled sups.
R-08	minor (upstream)	`docs/hvm/interactions/eql_nam.md` is empty (20 bytes), although `wnf_eql_nam` exists.	`wc -c`; `grep wnf_eql_nam src/hvm.c`	Noted in §8.10 and Appendix B.
R-09	major	DUP-NOD was described as copying only childless nodes. The sheet copies any node `T{a,b,…}` one layer, re-duplicating its children.	`dup_nod.md` (Part L)	Coursebook §5.3, §8.10.
R-10	major	The sharing showpiece's baseline was misidentified. "8 interactions" is calling `@f` twice (two `OP2-NUM-NUM`); the literal `[[(2+2),10],[(2+2),20]]` takes 2.	runs at HEAD: 11 / 8 / 2; `-D` OP2 count 1 vs 2	Coursebook §7.5.

Id	Sev.	Finding	Evidence	Fix
R-11	minor	The asterisk trap: `*` is erasure in the IC reference and the wildcard `ANY` (tag 40) in HVM4.	`@main = [*, 1]` → `[*,1]`, 0 itrs; tag table `src/hvm.c:38–88`	§5.7, D4.
R-12	minor	"Never reuse a label" was too strong. `@cmul(@c2,@c2)` gives 4 with equal labels; only a fan meeting a same-labelled non-partner fan is wrong (`@c2(@c2)`).	14 itrs, `λa.λb.a(a(a(a(b))))`; `@c2(@c2)` → `λa.λb.a`, 9 itrs	§7.2 rule restated; E7.3.
R-13	polish	The c2/k2 accident is also a deliberate HVM5 kernel test.	G.51 status: "K3 shown (same label gives 1, not 4)"; `version2/tests/kernel_rules.bend:110`	§10.5.
R-14	major	The compendium's HVM4 part was a March snapshot without notice. HEAD changed README (16 diff lines), memory.md (24), syntax.md (53: dup destructuring, auto-fork, separators), collapser.md (18: sequential queue), interaction_calculus.md (8); tests: `aot_helper_name_collision`, `ffi` removed; `dup_pattern_*` added.	`diff` old clone vs HEAD	Part L added; Part I labelled as history.
R-15	minor	"1 timeout (`fib_small`)" in the test suite is a timing artifact of the 2 s default.	`HVM_TEST_TIMEOUT_SECS=20 ./devs/test/_all_.sh` → 217 PASS, "All tests passed!"	Coursebook Appendix A/C.
R-16	polish	Quotes attributed to Taelin and to the HVM5 documents were not systematically checked.	script: 41 quoted passages, whitespace-normalized substring match against the cited files → 41/41	Coursebook §11 and every blockquote.
R-17	minor	§3.8's hand reduction drew the commutation copies on the wrong sides in a draft; cell counts must be re-checked.	rule §3.6: γ-copies to δ's side	Corrected gallery, explicit warning box, E3.3.
R-18	minor	Links were not checked. Spot check (`curl -sIL`, 12 core URLs): 11 × 200; `https://github.com/HigherOrderCO/HVM/issues/60` → **404** (cited in H.53).	command output	Listed here; the URL stays in H.53 as written.
R-19	major	HVM5 claims needed status classes: v1 reducer is a tree evaluator (not an inet), v2 kernel done sequentially, capabilities "checks, untested", TCP "not started".	G.2, G.51, G.52	Coursebook §10.5 with the status table.
R-22	minor	The first edition's appendix mapped content onto planned `/course/` page scaffolds. Both documents are now self-contained; nothing refers to a scaffold.	—	Removed; the coursebook has a generated table of contents, the compendium a numbered one.

Id	Sev.	Finding	Evidence	Fix
R-23	major	Both editions stated Lafont's universality theorem (every interaction system) for the *symmetric* combinators. Mazza: the symmetric combinators are universal only for *polarized* systems, and differ from Lafont's in that γ–γ annihilation is straight instead of crossed.	Mazza abstract, §2.4, §5	Coursebook §3 intro and §3.9 corrected; claim table row added.
R-24	major	The coursebook attributed the non-elementary cost of optimal reduction to the oracle's bookkeeping. Asperti–Mairson bound the total work of any Lévy-optimal implementation; Asperti–Coppola–Martini (2004) show duplication alone is non-elementary and that EAL-typable terms need only simple fan labels.	Laretto slides 34–36	§9.3, claim table, E9.5 corrected.
R-25	major	The quadratic slowdown was cited but never measured. HVM4 HEAD: `@eq(@X, @N)` costs 2,376 → 498,576 interactions for n = 16 → 256 (×3.9 per doubling), dominated by DUP-NOD; the control is linear.	§9.4 measurement	§9.4 rewritten with the table, the mechanism, safe nodes and unordered DUPs.
R-26	minor	Missing topics found while reading the user's notes (HOC.pdf): vicious circles, runtime fusion, effects, Taelin's 2022 design account. Each was checked against a primary source or a HEAD run before being added.	Mazza §2.3; H.48 + HEAD runs; Borner; Taelin's 2022 HN comment	Coursebook §2.4, §7.7, §9.4a, §8B.0.
R-27	minor	Second pass over all 450 pages of HOC.pdf. Added, each checked: Wadsworth's 1971 graph reduction and why it is not optimal (Laretto); superposition misconceptions (collapse does not pick a winner; SUP is not a task runner; two wrong rule shortcuts); trees vs lists as work vs span, measured on HEAD (list 5n + 1, tree 7n − 5); the roots in proof nets, the `!` comonad and the geometry of interaction (Mazza §1, Thm 4.7; Laretto). Not added: claims in the notes that are wrong (annihilation = erasure; REF needed for recursion), unsourced performance figures, and the speculative LLM/agent/consensus designs.	Mazza, Laretto, HEAD runs	Coursebook §1.2, §6.6, §9.5, §9.5a; E9.9–E9.10.
R-20	polish	The first edition counted 76 exercises with uneven coverage of the machine and HVM5.	—	114 exercises (E1.1–E11.3, E8B.1–E8B.9, D1–D20, C1–C12), every part covered.

What changed, first edition → Claude edition

- **Compendium:** Parts A–I and K unchanged byte for byte (old ids valid); new Part L (HVM4 HEAD, 70 files incl. all 61 sheets); new introduction with part map, reading paths and a Taelin reading map; Appendix R; regenerated link index; provenance with sha256 for Part L.
- **Coursebook:** rewritten. New: legend; port-by-port galleries for all six combinator rules; real `-D` traces for every IC rule; Part 8 from `src/hvm.c` at HEAD (term word, 45 tags, ITRS, ALO, SUB, collapse); Part 9 with a claim-class table; Part 10 with HVM5 v2's fork/test/merge and its dated status; 15 verbatim insight cards; Appendix B with all 61 sheets; a ledger of every run.

LINK INDEX (regenerated 2026-11-06)

421 distinct URLs, grouped by domain; each with the section where it first appears and its number of occurrences. Core links were spot-checked with `curl -sIL` on 2026-11-06 (Appendix R, R-18); the rest are listed as found, not verified.

github.com (201)

- https://github.com/bendlang/bend (`https://github.com/bendlang/bend`) · first in front · ×36
- https://github.com/HigherOrderCO/HVM4 (`https://github.com/HigherOrderCO/HVM4`) · first in front · ×22
- https://github.com/VictorTaelin/Interaction-Calculus (`https://github.com/VictorTaelin/Interaction-Calculus`) · first in front · ×24
- https://github.com/VictorTaelin (`https://github.com/VictorTaelin`) · first in front · ×13
- https://github.com/HigherOrderCO (`https://github.com/HigherOrderCO`) · first in front · ×14
- https://github.com/HigherOrderCO/HVM (`https://github.com/HigherOrderCO/HVM`) · first in front · ×1
- https://github.com/VictorTaelin/optlam (`https://github.com/VictorTaelin/optlam`) · first in front · ×2
- https://github.com/HigherOrderCO/HVM3 (`https://github.com/HigherOrderCO/HVM3`) · first in front · ×13
- https://github.com/VictorTaelin/Symmetric-Interaction-Calculus (`https://github.com/VictorTaelin/Symmetric-Interaction-Calculus`) · first in front · ×2
- https://github.com/VictorTaelin/Interaction-Calculus.git (`https://github.com/VictorTaelin/Interaction-Calculus.git`) · first in A.15 · ×2
- https://github.com/AdrielSantana/metal-bending (`https://github.com/AdrielSantana/metal-bending`) · first in A.17 · ×4
- https://github.com/aivv73/bend-voxel (`https://github.com/aivv73/bend-voxel`) · first in A.17 · ×3
- https://github.com/costamatheus97/bend-craft (`https://github.com/costamatheus97/bend-craft`) · first in A.17 · ×3
- https://github.com/eliesgalvira/bendoom (`https://github.com/eliesgalvira/bendoom`) · first in A.17 · ×4
- https://github.com/gildrb/elpis-fast (`https://github.com/gildrb/elpis-fast`) · first in A.17 · ×3
- https://github.com/Giulio2002/bend-collections (`https://github.com/Giulio2002/bend-collections`) · first in A.17 · ×3
- https://github.com/i2cjak/Backplane (`https://github.com/i2cjak/Backplane`) · first in A.17 · ×3
- https://github.com/ind-igo/bend-frontend (`https://github.com/ind-igo/bend-frontend`) · first in A.17 · ×3
- https://github.com/lilalittle/bend-packages (`https://github.com/lilalittle/bend-packages`) · first in A.17 · ×5
- https://github.com/Lulzx/bendc (`https://github.com/Lulzx/bendc`) · first in A.17 · ×3
- https://github.com/paymog/bend-kit (`https://github.com/paymog/bend-kit`) · first in A.17 · ×3
- https://github.com/nuxyel/bend-ml (`https://github.com/nuxyel/bend-ml`) · first in A.17 · ×3
- https://github.com/angerman/birc (`https://github.com/angerman/birc`) · first in A.17 · ×3
- https://github.com/FabianVegaA/mylsm (`https://github.com/FabianVegaA/mylsm`) · first in A.17 · ×3
- https://github.com/bendlib/bendlib (`https://github.com/bendlib/bendlib`) · first in A.17 · ×3
- https://github.com/777genius/awesome-bend (`https://github.com/777genius/awesome-bend`) · first in A.17 · ×4
- https://github.com/nuxyel/llama.bend (`https://github.com/nuxyel/llama.bend`) · first in A.17 · ×2
- https://github.com/rootagi/bend-json (`https://github.com/rootagi/bend-json`) · first in A.17 · ×2
- https://github.com/ind-igo/bend-evm (`https://github.com/ind-igo/bend-evm`) · first in A.17 · ×2
- https://github.com/CrocSwap/bendSVM (`https://github.com/CrocSwap/bendSVM`) · first in A.17 · ×2
- https://github.com/MatheusBBarni/bendJVM (`https://github.com/MatheusBBarni/bendJVM`) · first in A.17 · ×2
- https://github.com/nicolas-abril/bend2-libs (`https://github.com/nicolas-abril/bend2-libs`) · first in A.17 · ×2
- https://github.com/OkeyAmy/bend-voxel-bench (`https://github.com/OkeyAmy/bend-voxel-bench`) · first in A.17 · ×2
- https://github.com/jonathanperis/jonlib (`https://github.com/jonathanperis/jonlib`) · first in A.17 · ×2
- https://github.com/bendlang/bend/blob/main/guide/GUIDE.md (`https://github.com/bendlang/bend/blob/main/guide/GUIDE.md`) · first in C.2 · ×30
- https://github.com/VictorTaelin/Bend2 (`https://github.com/VictorTaelin/Bend2`) · first in C.3 · ×3
- https://github.com/VictorTaelin/Formality (`https://github.com/VictorTaelin/Formality`) · first in C.19 · ×1
- https://github.com/HigherOrderCO/Kind (`https://github.com/HigherOrderCO/Kind`) · first in C.19 · ×19
- https://github.com/VictorTaelin/articles (`https://github.com/VictorTaelin/articles`) · first in C.19 · ×1
- https://github.com/luiscleto/minebender (`https://github.com/luiscleto/minebender`) · first in C.38 · ×1
- https://github.com/bendlang/bend.git (`https://github.com/bendlang/bend.git`) · first in C.39 · ×1

- https://github.com/HigherOrderCO/HVM4/blob/main/docs/primer.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/primer.md) · first in C.44 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/syntax.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/syntax.md) · first in C.44 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/main/docs/theory/interaction_calculus.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/theory/interaction_calculus.md) · first in C.46 · ×3
- https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/memory.md (https://github.com/HigherOrderCO/HVM4/blob/main/docs/hvm/memory.md) · first in C.46 · ×3
- https://github.com/bendlang/bend/tree/main/demos (https://github.com/bendlang/bend/tree/main/demos) · first in F.1 · ×15
- https://github.com/kindelia/Kindelia (https://github.com/kindelia/Kindelia) · first in F.1 · ×19
- https://github.com/HigherOrderCO/HVM1 (https://github.com/HigherOrderCO/HVM1) · first in F.4 · ×18
- https://github.com/HigherOrderCO/HVM2 (https://github.com/HigherOrderCO/HVM2) · first in F.4 · ×13
- https://github.com/HigherOrderCO/Bend1 (https://github.com/HigherOrderCO/Bend1) · first in F.4 · ×9
- https://github.com/bendlang/bend/issues (https://github.com/bendlang/bend/issues) · first in F.10 · ×15
- https://github.com/bendlang/bend/tree/main/guide (https://github.com/bendlang/bend/tree/main/guide) · first in F.11 · ×3
- https://github.com/bendlang/bend/blob/main/bend2/base.bend (https://github.com/bendlang/bend/blob/main/bend2/base.bend) · first in F.11 · ×11
- https://github.com/HigherOrderCO/HVM1/blob/master/guide/HOW.md (https://github.com/HigherOrderCO/HVM1/blob/master/guide/HOW.md) · first in F.18 · ×1
- https://github.com/HigherOrderCO/hvm/blob/main/src/hvm.rs (https://github.com/HigherOrderCO/hvm/blob/main/src/hvm.rs) · first in F.18 · ×1
- https://github.com/bendlang/bend/tree/main/paper (https://github.com/bendlang/bend/tree/main/paper) · first in F.23 · ×1
- https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md (https://github.com/kindelia/Kindelia/blob/master/WHITEBOOK.md) · first in F.29 · ×8
- https://github.com/HigherOrderCO/Bend2 (https://github.com/HigherOrderCO/Bend2) · first in F.29 · ×6
- https://github.com/don2e4/bend2-lsp (https://github.com/don2e4/bend2-lsp) · first in F.29 · ×10
- https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474 (https://github.com/HigherOrderCO/HVM/issues/167#issuecomment-1314665474) · first in F.30 · ×4
- https://github.com/HigherOrderCO/KindBook (https://github.com/HigherOrderCO/KindBook) · first in F.30 · ×2
- https://github.com/VictorTaelin/Cedille-Core (https://github.com/VictorTaelin/Cedille-Core) · first in F.30 · ×4
- https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf) · first in F.31 · ×8
- https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf) · first in F.31 · ×5
- https://github.com/bendlang/bend/blob/main/bend2/bend.lean (https://github.com/bendlang/bend/blob/main/bend2/bend.lean) · first in F.31 · ×3
- https://github.com/VictorTaelin/WebMonkeys (https://github.com/VictorTaelin/WebMonkeys) · first in F.32 · ×3
- https://github.com/VictorTaelin/UrnaCripto (https://github.com/VictorTaelin/UrnaCripto) · first in F.32 · ×3
- https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md (https://github.com/HigherOrderCO/HVM/blob/73da3bbc4b222fb8f044fcc5dad202e9752a0abc/HOW.md) · first in F.32 · ×4
- https://github.com/bendlang/bend/tree/main/bench/runtime (https://github.com/bendlang/bend/tree/main/bench/runtime) · first in F.34 · ×6
- https://github.com/bendlang/bend/tree/main/bench/checker (https://github.com/bendlang/bend/tree/main/bench/checker) · first in F.37 · ×5
- https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md (https://github.com/dart-lang/sdk/blob/main/runtime/docs/hot-reload.md) · first in F.39 · ×6
- https://github.com/HigherOrderCO/Bend1/tree/main/examples (https://github.com/HigherOrderCO/Bend1/tree/main/examples) · first in F.41 · ×7
- https://github.com/HigherOrderCO-archive/hvm-64 (https://github.com/HigherOrderCO-archive/hvm-64) · first in F.41 · ×5
- https://github.com/bendlang/bend/blob/main/guide/EFFECTS.md (https://github.com/bendlang/bend/blob/main/guide/EFFECTS.md) · first in F.55 · ×2
- https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini (https://github.com/bendlang/bend/tree/main/demos/pure_hvm5_mini) · first in F.55 · ×4
- https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md (https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md) · first in F.55 · ×2
- https://github.com/bendlang/bend/tree/main/bench (https://github.com/bendlang/bend/tree/main/bench) · first in F.55 · ×2

- https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d (https://github.com/bendlang/bend/tree/main/demos/app_slash_boss_3d) · first in F.55 · ×4
- https://github.com/bendlang/bend/tree/main/demos/app_pong_game_2d (https://github.com/bendlang/bend/tree/main/demos/app_pong_game_2d) · first in F.55 · ×2
- https://github.com/bendlang/bend/tree/main/demos/pure_par_sum (https://github.com/bendlang/bend/tree/main/demos/pure_par_sum) · first in F.55 · ×2
- https://github.com/bendlang/bend/tree/main/demos/pure_par_sort (https://github.com/bendlang/bend/tree/main/demos/pure_par_sort) · first in F.55 · ×2
- https://github.com/bendlang/bend/tree/main/evals (https://github.com/bendlang/bend/tree/main/evals) · first in F.55 · ×2
- https://github.com/bendlang/bend/tree/main/demos/proof_insertion_sort (https://github.com/bendlang/bend/tree/main/demos/proof_insertion_sort) · first in F.55 · ×2
- https://github.com/bendlang/bend/tree/main/demos/app_win_is_bug_2d (https://github.com/bendlang/bend/tree/main/demos/app_win_is_bug_2d) · first in F.55 · ×2
- https://github.com/kindelia/Kindelia/blob/master/genesis.kdl (https://github.com/kindelia/Kindelia/blob/master/genesis.kdl) · first in F.57 · ×1
- https://github.com/bendlang/bend/blob/main/demos/pure_hvm5_mini/main.bend (https://github.com/bendlang/bend/blob/main/demos/pure_hvm5_mini/main.bend) · first in G.35 · ×1
- https://github.com/golemfactory/golem (https://github.com/golemfactory/golem) · first in G.35 · ×1
- https://github.com/unisonweb/unison (https://github.com/unisonweb/unison) · first in G.35 · ×1
- https://github.com/NixOS/nix (https://github.com/NixOS/nix) · first in G.35 · ×1
- https://github.com/tezos/tezos (https://github.com/tezos/tezos) · first in G.35 · ×1
- https://github.com/starkware-libs/cairo (https://github.com/starkware-libs/cairo) · first in G.35 · ×1
- https://github.com/risc0/risc0 (https://github.com/risc0/risc0) · first in G.35 · ×1
- https://github.com/mirage/mirage (https://github.com/mirage/mirage) · first in G.35 · ×1
- https://github.com/mirage/irmin (https://github.com/mirage/irmin) · first in G.35 · ×1
- https://github.com/bendlang/bend/pull/866 (https://github.com/bendlang/bend/pull/866) · first in G.35 · ×1
- https://github.com/rchain/rchain (https://github.com/rchain/rchain) · first in G.35 · ×1
- https://github.com/Lorenzobattistela (https://github.com/Lorenzobattistela) · first in H.1 · ×1
- https://github.com/pjcavalcanti (https://github.com/pjcavalcanti) · first in H.1 · ×1
- https://github.com/nicolas-abril (https://github.com/nicolas-abril) · first in H.1 · ×1
- https://github.com/Ostrowskii (https://github.com/Ostrowskii) · first in H.1 · ×1
- https://github.com/Sipher (https://github.com/Sipher) · first in H.1 · ×1
- https://github.com/HigherOrderCO/Bend (https://github.com/HigherOrderCO/Bend) · first in H.10 · ×12
- https://github.com/leanprover/lean4 (https://github.com/leanprover/lean4) · first in H.11 · ×1
- https://github.com/leanprover-community/mathlib4 (https://github.com/leanprover-community/mathlib4) · first in H.11 · ×1
- https://github.com/modular/modular (https://github.com/modular/modular) · first in H.12 · ×1
- https://github.com/HigherOrderCO/HVM4/tree/main/docs/hvm/interactions (https://github.com/HigherOrderCO/HVM4/tree/main/docs/hvm/interactions) · first in H.17 · ×1
- https://github.com/HigherOrderCO/Bend#running-bend-programs (https://github.com/HigherOrderCO/Bend#running-bend-programs) · first in H.22 · ×1
- https://github.com/higherorderco/hvm (https://github.com/higherorderco/hvm) · first in H.26 · ×4
- https://github.com/HigherOrderCO/Bend/issues/341 (https://github.com/HigherOrderCO/Bend/issues/341) · first in H.26 · ×1
- https://github.com/HigherOrderCO/Bend/tree/main/examples (https://github.com/HigherOrderCO/Bend/tree/main/examples) · first in H.26 · ×1
- https://github.com/HigherOrderCO/Bend/blob/main/GUIDE.md (https://github.com/HigherOrderCO/Bend/blob/main/GUIDE.md) · first in H.26 · ×1
- https://github.com/HigherOrderCO/Bend/blob/main/FEATURES.md (https://github.com/HigherOrderCO/Bend/blob/main/FEATURES.md) · first in H.26 · ×1
- https://github.com/HigherOrderCO/HVM-Core (https://github.com/HigherOrderCO/HVM-Core) · first in H.27 · ×1
- https://github.com/HigherOrderCO/Kindex/blob/master/Apps/IC/_.kind2 (https://github.com/HigherOrderCO/Kindex/blob/master/Apps/IC/_.kind2) · first in H.27 · ×1
- https://github.com/HigherOrderCO/HVM/blob/master/guide/HOW.md (https://github.com/HigherOrderCO/HVM/blob/master/guide/HOW.md) · first in H.27 · ×1
- https://github.com/HigherOrderCO/HVM/issues/60 (https://github.com/HigherOrderCO/HVM/issues/60) · first in H.27 · ×2
- https://github.com/HigherOrderCO/HVM/issues/167 (https://github.com/HigherOrderCO/HVM/issues/167) · first in H.27 · ×1

- https://github.com/inpla/inpla (https://github.com/inpla/inpla) · first in H.27 · ×1
- https://github.com/VictorTaelin/abstract-algorithm (https://github.com/VictorTaelin/abstract-algorithm) · first in H.41 · ×2
- https://github.com/victortaelin/interaction-calculus-of-constructions (https://github.com/victortaelin/interaction-calculus-of-constructions) · first in H.41 · ×2
- https://github.com/VictorTaelin/Formality-Net-legacy (https://github.com/VictorTaelin/Formality-Net-legacy) · first in H.41 · ×1
- https://github.com/HigherOrderCO/HVM/blob/main/paper/PAPER.pdf (https://github.com/HigherOrderCO/HVM/blob/main/paper/PAPER.pdf) · first in H.41 · ×1
- https://github.com/higherorderco/kind (https://github.com/higherorderco/kind) · first in H.41 · ×1
- https://github.com/Kindelia/HVM/issues/64#issuecomment-1030688993 (https://github.com/Kindelia/HVM/issues/64#issuecomment-1030688993) · first in H.46 · ×1
- https://github.com/Kindelia/HVM/blob/master/src/runtime.c (https://github.com/Kindelia/HVM/blob/master/src/runtime.c) · first in H.46 · ×1
- https://github.com/HigherOrderCO/kindbook (https://github.com/HigherOrderCO/kindbook) · first in H.47 · ×1
- https://github.com/VictorTaelin/articles/tree/master/0004-supercompilation-for-free (https://github.com/VictorTaelin/articles/tree/master/0004-supercompilation-for-free) · first in H.48 · ×1
- https://github.com/HigherOrderCO/HVM3/blob/main/book/lambda_enumerator_optimal.hvml (https://github.com/HigherOrderCO/HVM3/blob/main/book/lambda_enumerator_optimal.hvml) · first in H.51 · ×1
- https://github.com/higherOrderCO/HVM-lang (https://github.com/higherOrderCO/HVM-lang) · first in H.52 · ×1
- https://github.com/HigherOrderCO/kind2/blob/c8fd9e54d500a783dc9e428b178647cbcb27235f/docs/equality.md (https://github.com/HigherOrderCO/kind2/blob/c8fd9e54d500a783dc9e428b178647cbcb27235f/docs/equality.md) · first in H.54 · ×1
- https://github.com/HigherOrderCO/HVM4` (https://github.com/HigherOrderCO/HVM4%60) · first in K.5 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/README.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/README.md) · first in K.5 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/AGENTS.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/AGENTS.md) · first in L.1 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/primer.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/primer.md) · first in L.2 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/theory/interaction_calculus.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/theory/interaction_calculus.md) · first in L.3 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/core.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/core.md) · first in L.4 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/memory.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/memory.md) · first in L.5 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/syntax.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/syntax.md) · first in L.6 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/collapser.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/collapser.md) · first in L.7 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_cop.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_cop.md) · first in L.8 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_dry.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_dry.md) · first in L.9 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_dup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_dup.md) · first in L.10 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_lam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_lam.md) · first in L.11 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_nam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_nam.md) · first in L.12 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_nod.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_nod.md) · first in L.13 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_var.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/alo_var.md) · first in L.14 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_era.md) · first in L.15 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_inc.md) · first in L.16 · ×1

- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_num.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_num.md) · first in L.17 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/and_sup.md) · first in L.18 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_ctr.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_ctr.md) · first in L.19 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_dry.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_dry.md) · first in L.20 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_era.md) · first in L.21 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_inc.md) · first in L.22 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_lam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_lam.md) · first in L.23 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_mat_ctr.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_mat_ctr.md) · first in L.24 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_mat_num.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_mat_num.md) · first in L.25 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_mat_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_mat_sup.md) · first in L.26 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_nam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_nam.md) · first in L.27 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/app_sup.md) · first in L.28 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_era.md) · first in L.29 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_inc.md) · first in L.30 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_num.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_num.md) · first in L.31 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/ddu_sup.md) · first in L.32 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_era.md) · first in L.33 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_inc.md) · first in L.34 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_num.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_num.md) · first in L.35 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dsu_sup.md) · first in L.36 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_dry.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_dry.md) · first in L.37 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_lam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_lam.md) · first in L.38 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_nam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_nam.md) · first in L.39 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_nod.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_nod.md) · first in L.40 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/dup_sup.md) · first in L.41 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_any.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_any.md) · first in L.42 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_ctr.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_ctr.md) · first in L.43 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_dry.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_dry.md) · first in L.44 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_era.md) · first in L.45 · ×1

- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_inc.md) · first in L.46 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_lam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_lam.md) · first in L.47 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_mat.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_mat.md) · first in L.48 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_nam.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_nam.md) · first in L.49 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_num.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_num.md) · first in L.50 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_sup.md) · first in L.51 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_use.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/eql_use.md) · first in L.52 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/mat_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/mat_inc.md) · first in L.53 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_era.md) · first in L.54 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_inc.md) · first in L.55 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_num_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_num_era.md) · first in L.56 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_num_num.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_num_num.md) · first in L.57 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_num_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_num_sup.md) · first in L.58 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/op2_sup.md) · first in L.59 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_era.md) · first in L.60 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_inc.md) · first in L.61 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_num.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_num.md) · first in L.62 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/or_sup.md) · first in L.63 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/uns.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/uns.md) · first in L.64 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_era.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_era.md) · first in L.65 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_inc.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_inc.md) · first in L.66 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_sup.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_sup.md) · first in L.67 · ×1
- https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_val.md (https://github.com/HigherOrderCO/HVM4/blob/6defdfc/docs/hvm/interactions/use_val.md) · first in L.68 · ×1
- https://github.com/HigherOrderCO/HVM/issues/60` (https://github.com/HigherOrderCO/HVM/issues/60%60) · first in L.70 · ×1

bend2.dev (54)

- https://bend2.dev/ (https://bend2.dev/) · first in front · ×8
- https://bend2.dev/contents/ (https://bend2.dev/contents/) · first in C.2 · ×2
- https://bend2.dev/llms.txt (https://bend2.dev/llms.txt) · first in C.2 · ×2
- https://bend2.dev/learn/examples.json (https://bend2.dev/learn/examples.json) · first in C.2 · ×1
- https://bend2.dev/errors/diagnostics.json (https://bend2.dev/errors/diagnostics.json) · first in C.2 · ×2
- https://bend2.dev/notes/what-is-bend2/ (https://bend2.dev/notes/what-is-bend2/) · first in C.3 · ×7
- https://bend2.dev/learn/hello/ (https://bend2.dev/learn/hello/) · first in C.4 · ×6

- https://bend2.dev/notes/bend2-release-date/ (https://bend2.dev/notes/bend2-release-date/) · first in C.5 · ×5
- https://bend2.dev/learn/ (https://bend2.dev/learn/) · first in C.6 · ×12
- https://bend2.dev/notes/bend2-vs-lean/ (https://bend2.dev/notes/bend2-vs-lean/) · first in C.6 · ×7
- https://bend2.dev/notes/interaction-nets-explained/ (https://bend2.dev/notes/interaction-nets-explained/) · first in C.6 · ×8
- https://bend2.dev/learn/functions/ (https://bend2.dev/learn/functions/) · first in C.8 · ×1
- https://bend2.dev/learn/numbers/ (https://bend2.dev/learn/numbers/) · first in C.8 · ×1
- https://bend2.dev/notes/bend2-syntax-primer/ (https://bend2.dev/notes/bend2-syntax-primer/) · first in C.8 · ×1
- https://bend2.dev/learn/adts/ (https://bend2.dev/learn/adts/) · first in C.9 · ×1
- https://bend2.dev/learn/records/ (https://bend2.dev/learn/records/) · first in C.9 · ×1
- https://bend2.dev/learn/pattern-matching/ (https://bend2.dev/learn/pattern-matching/) · first in C.9 · ×1
- https://bend2.dev/learn/recursion/ (https://bend2.dev/learn/recursion/) · first in C.9 · ×1
- https://bend2.dev/learn/totality/ (https://bend2.dev/learn/totality/) · first in C.9 · ×2
- https://bend2.dev/learn/imports/ (https://bend2.dev/learn/imports/) · first in C.9 · ×1
- https://bend2.dev/learn/definition-order/ (https://bend2.dev/learn/definition-order/) · first in C.9 · ×1
- https://bend2.dev/errors/ (https://bend2.dev/errors/) · first in C.11 · ×1
- https://bend2.dev/notes/ai-agents-bend/ (https://bend2.dev/notes/ai-agents-bend/) · first in C.11 · ×4
- https://bend2.dev/learn/proofs/ (https://bend2.dev/learn/proofs/) · first in C.15 · ×6
- https://bend2.dev/learn/induction/ (https://bend2.dev/learn/induction/) · first in C.15 · ×1
- https://bend2.dev/learn/equality/ (https://bend2.dev/learn/equality/) · first in C.15 · ×1
- https://bend2.dev/learn/holes/ (https://bend2.dev/learn/holes/) · first in C.15 · ×1
- https://bend2.dev/learn/types-as-specifications/ (https://bend2.dev/learn/types-as-specifications/) · first in C.15 · ×2
- https://bend2.dev/learn/dependent-types/ (https://bend2.dev/learn/dependent-types/) · first in C.16 · ×1
- https://bend2.dev/learn/sized-lists/ (https://bend2.dev/learn/sized-lists/) · first in C.16 · ×1
- https://bend2.dev/errors/unsafe-dependents/ (https://bend2.dev/errors/unsafe-dependents/) · first in C.16 · ×1
- https://bend2.dev/learn/duplication/ (https://bend2.dev/learn/duplication/) · first in C.22 · ×3
- https://bend2.dev/learn/kinds/ (https://bend2.dev/learn/kinds/) · first in C.22 · ×1
- https://bend2.dev/learn/arrays/ (https://bend2.dev/learn/arrays/) · first in C.22 · ×1
- https://bend2.dev/learn/maps/ (https://bend2.dev/learn/maps/) · first in C.22 · ×1
- https://bend2.dev/learn/templates/ (https://bend2.dev/learn/templates/) · first in C.22 · ×1
- https://bend2.dev/learn/closures/ (https://bend2.dev/learn/closures/) · first in C.22 · ×1
- https://bend2.dev/learn/folds/ (https://bend2.dev/learn/folds/) · first in C.22 · ×1
- https://bend2.dev/notes/hvm4-explained/ (https://bend2.dev/notes/hvm4-explained/) · first in C.25 · ×8
- https://bend2.dev/notes/search-by-superposition/ (https://bend2.dev/notes/search-by-superposition/) · first in C.25 · ×1
- https://bend2.dev/notes/release-status/ (https://bend2.dev/notes/release-status/) · first in C.25 · ×2
- https://bend2.dev/notes/bend2-vs-bend1/ (https://bend2.dev/notes/bend2-vs-bend1/) · first in C.25 · ×5
- https://bend2.dev/learn/io/ (https://bend2.dev/learn/io/) · first in C.31 · ×1
- https://bend2.dev/learn/concurrency/ (https://bend2.dev/learn/concurrency/) · first in C.31 · ×1
- https://bend2.dev/learn/files/ (https://bend2.dev/learn/files/) · first in C.31 · ×1
- https://bend2.dev/learn/compile-to-c/ (https://bend2.dev/learn/compile-to-c/) · first in C.31 · ×2
- https://bend2.dev/learn/compile-to-javascript/ (https://bend2.dev/learn/compile-to-javascript/) · first in C.31 · ×1
- https://bend2.dev/notes/install-bend2-from-source/ (https://bend2.dev/notes/install-bend2-from-source/) · first in C.39 · ×1
- https://bend2.dev/notes/bend-runtime-options/ (https://bend2.dev/notes/bend-runtime-options/) · first in C.39 · ×1
- https://bend2.dev/notes/bendrt-explained/ (https://bend2.dev/notes/bendrt-explained/) · first in C.39 · ×3
- https://bend2.dev/learn/parallelism/ (https://bend2.dev/learn/parallelism/) · first in C.41 · ×6
- https://bend2.dev/learn/measuring-speedup/ (https://bend2.dev/learn/measuring-speedup/) · first in C.41 · ×1
- https://bend2.dev/learn/gpu/ (https://bend2.dev/learn/gpu/) · first in C.41 · ×6
- https://bend2.dev/notes/bend2-vs-mojo/ (https://bend2.dev/notes/bend2-vs-mojo/) · first in C.49 · ×5

gist.github.com (26)

- https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615 (https://gist.github.com/VictorTaelin/9061306220929f04e7e6980f23ade615) · first in C.26 · ×9

- https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34 (https://gist.github.com/VictorTaelin/d5c318348aaee7033eb3d18b0b0ace34) · first in C.26 · ×5
- https://gist.github.com/VictorTaelin/5d42e97684ff81f7e507aa5c149b2a64 (https://gist.github.com/VictorTaelin/5d42e97684ff81f7e507aa5c149b2a64) · first in C.26 · ×1
- https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e (https://gist.github.com/VictorTaelin/7c4c69a1f07b5c668be613f1032e7d4e) · first in C.26 · ×5
- https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6 (https://gist.github.com/VictorTaelin/311f6a58a7756945196c15733e61d0c6) · first in C.45 · ×11
- https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b (https://gist.github.com/VictorTaelin/46936b9fdfc3f982f07963c11756e36b) · first in C.48 · ×12
- https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1 (https://gist.github.com/VictorTaelin/77fd5a2a8a4a07e1da6157ebca3c7cf1) · first in C.48 · ×5
- https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8 (https://gist.github.com/VictorTaelin/a5571afaf5ee565689d2b9a981bd9df8) · first in F.30 · ×4
- https://gist.github.com/VictorTaelin/5776ede998d0039ad1cc9b12fd96811c (https://gist.github.com/VictorTaelin/5776ede998d0039ad1cc9b12fd96811c) · first in F.30 · ×3
- https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b (https://gist.github.com/VictorTaelin/2aba162f2b04478dc53e5615f482db7b) · first in F.30 · ×4
- https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a (https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a) · first in F.30 · ×4
- https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5 (https://gist.github.com/VictorTaelin/3f748a46e95071e29462b1ac93c294c5) · first in F.30 · ×6
- https://gist.github.com/VictorTaelin/85b94b5ba5b8b5440ded64bba8c89ac2 (https://gist.github.com/VictorTaelin/85b94b5ba5b8b5440ded64bba8c89ac2) · first in F.32 · ×3
- https://gist.github.com/VictorTaelin/676241eb4e290a006e5618f97b8a1c25 (https://gist.github.com/VictorTaelin/676241eb4e290a006e5618f97b8a1c25) · first in F.32 · ×3
- https://gist.github.com/VictorTaelin/7fe49a99ebca42e5721aa1a3bb32e278 (https://gist.github.com/VictorTaelin/7fe49a99ebca42e5721aa1a3bb32e278) · first in F.32 · ×3
- https://gist.github.com/VictorTaelin/60d3bc72fb4edefecd42095e44138b41 (https://gist.github.com/VictorTaelin/60d3bc72fb4edefecd42095e44138b41) · first in F.32 · ×3
- https://gist.github.com/VictorTaelin/fb798a5bd182f8c57dd302380f69777a (https://gist.github.com/VictorTaelin/fb798a5bd182f8c57dd302380f69777a) · first in F.32 · ×3
- https://gist.github.com/VictorTaelin/c672b5bd9e734bae63ac35c010be38ab (https://gist.github.com/VictorTaelin/c672b5bd9e734bae63ac35c010be38ab) · first in F.32 · ×2
- https://gist.github.com/VictorTaelin/1af22e2c87f176da0e5ff8cd3430b04f (https://gist.github.com/VictorTaelin/1af22e2c87f176da0e5ff8cd3430b04f) · first in F.32 · ×2
- https://gist.github.com/VictorTaelin/d293328f75291b23e203e9d9db9bd136 (https://gist.github.com/VictorTaelin/d293328f75291b23e203e9d9db9bd136) · first in F.32 · ×2
- https://gist.github.com/VictorTaelin/997260bc68dc35acb82960051176daf4 (https://gist.github.com/VictorTaelin/997260bc68dc35acb82960051176daf4) · first in H.47 · ×1
- https://gist.github.com/VictorTaelin/27e1bce6f5c8d2f89aafef75025b4765 (https://gist.github.com/VictorTaelin/27e1bce6f5c8d2f89aafef75025b4765) · first in H.47 · ×1
- https://gist.github.com/VictorTaelin/2a1914bf827ae4c20b9e26d0c708ed99 (https://gist.github.com/VictorTaelin/2a1914bf827ae4c20b9e26d0c708ed99) · first in H.48 · ×1
- https://gist.github.com/VictorTaelin/bc0e8b38f93130a083b8de293d9cc7f5 (https://gist.github.com/VictorTaelin/bc0e8b38f93130a083b8de293d9cc7f5) · first in H.48 · ×1
- https://gist.github.com/VictorTaelin/053b3a608e633b826cfd79bc749d9cdf (https://gist.github.com/VictorTaelin/053b3a608e633b826cfd79bc749d9cdf) · first in H.48 · ×1
- https://gist.github.com/VictorTaelin/25da5d76f4be0c9ccf7e9c9c3e5693d7 (https://gist.github.com/VictorTaelin/25da5d76f4be0c9ccf7e9c9c3e5693d7) · first in H.48 · ×1

x.com (10)

- https://x.com/bendlang (https://x.com/bendlang) · first in F.10 · ×11
- https://x.com/VictorTaelin/status/1806690584670679387 (https://x.com/VictorTaelin/status/1806690584670679387) · first in F.32 · ×2
- https://x.com/VictorTaelin/status/1791213162525524076 (https://x.com/VictorTaelin/status/1791213162525524076) · first in F.32 · ×5

- https://x.com/avsa (https://x.com/avsa) · first in H.1 · ×1
- https://x.com/VictorTaelin/status/2093045510101676483 (https://x.com/VictorTaelin/status/2093045510101676483) · first in H.9 · ×2
- https://x.com/VictorTaelin/status/2078471338755232193 (https://x.com/VictorTaelin/status/2078471338755232193) · first in H.10 · ×5
- https://x.com/VictorTaelin/status/2029567059881857081 (https://x.com/VictorTaelin/status/2029567059881857081) · first in H.10 · ×5
- https://x.com/VictorTaelin/status/2078553927268589989 (https://x.com/VictorTaelin/status/2078553927268589989) · first in H.11 · ×3
- https://x.com/VictorTaelin/status/1779203515098845626 (https://x.com/VictorTaelin/status/1779203515098845626) · first in H.18 · ×1
- https://x.com/VictorTaelin/status/1819774880130158663 (https://x.com/VictorTaelin/status/1819774880130158663) · first in H.50 · ×1

raw.githubusercontent.com (8)

- https://raw.githubusercontent.com/HigherOrderCO/HVM4/main/src/hvm.c (https://raw.githubusercontent.com/HigherOrderCO/HVM4/main/src/hvm.c) · first in B.6 · ×2
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/main.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/main.bend) · first in F.25 · ×1
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/main.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/main.bend) · first in F.27 · ×1
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/LAWS.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/LAWS.bend) · first in F.27 · ×2
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/PROOF.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_hvm5_mini/PROOF.bend) · first in F.27 · ×1
- https://raw.githubusercontent.com/bendlang/bend/main/demos/proof_insertion_sort/LAWS.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/proof_insertion_sort/LAWS.bend) · first in F.28 · ×1
- https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/LAWS.bend (https://raw.githubusercontent.com/bendlang/bend/main/demos/pure_par_sort/LAWS.bend) · first in F.28 · ×1
- https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl (https://raw.githubusercontent.com/kindelia/Kindelia/master/genesis.kdl) · first in F.38 · ×6

dl.acm.org (7)

- https://dl.acm.org/doi/10.1145/96709.96718 (https://dl.acm.org/doi/10.1145/96709.96718) · first in C.40 · ×5
- https://dl.acm.org/doi/pdf/10.1145/96709.96718 (https://dl.acm.org/doi/pdf/10.1145/96709.96718) · first in F.31 · ×8
- https://dl.acm.org/doi/pdf/10.1145/96709.96711 (https://dl.acm.org/doi/pdf/10.1145/96709.96711) · first in F.31 · ×8
- https://dl.acm.org/doi/10.1145/96709.96711 (https://dl.acm.org/doi/10.1145/96709.96711) · first in H.18 · ×3
- https://dl.acm.org/doi/10.1145/268946.268971 (https://dl.acm.org/doi/10.1145/268946.268971) · first in H.18 · ×2
- https://dl.acm.org/doi/10.1145/232627.232639 (https://dl.acm.org/doi/10.1145/232627.232639) · first in H.41 · ×1
- https://dl.acm.org/doi/abs/10.1145/1131313.1131315 (https://dl.acm.org/doi/abs/10.1145/1131313.1131315) · first in H.41 · ×1

en.wikipedia.org (7)

- https://en.wikipedia.org/wiki/Interaction_nets (https://en.wikipedia.org/wiki/Interaction_nets) · first in F.1 · ×12
- https://en.wikipedia.org/wiki/Bitonic_sorter (https://en.wikipedia.org/wiki/Bitonic_sorter) · first in H.26 · ×1
- https://en.wikipedia.org/wiki/Bubble_sort (https://en.wikipedia.org/wiki/Bubble_sort) · first in H.27 · ×1
- https://en.wikipedia.org/wiki/Radix_sort (https://en.wikipedia.org/wiki/Radix_sort) · first in H.27 · ×1
- https://en.wikipedia.org/wiki/Church_encoding (https://en.wikipedia.org/wiki/Church_encoding) · first in H.27 · ×1
- https://en.wikipedia.org/wiki/Amdahl%27s_law (https://en.wikipedia.org/wiki/Amdahl%27s_law) · first in H.41 · ×1
- https://en.wikipedia.org/wiki/Exponentiation_by_squaring (https://en.wikipedia.org/wiki/Exponentiation_by_squaring) · first in H.48 · ×1

rholang.org (6)

- https://rholang.org/docs/rholang/ (https://rholang.org/docs/rholang/) · first in G.35 · ×1

- https://rholang.org/docs/deploy-contract/ (https://rholang.org/docs/deploy-contract/) · first in G.35 · ×1
- https://rholang.org/docs/propose-block/ (https://rholang.org/docs/propose-block/) · first in G.35 · ×1
- https://rholang.org/docs/names-processes/ (https://rholang.org/docs/names-processes/) · first in G.35 · ×1
- https://rholang.org/docs/send-receive/ (https://rholang.org/docs/send-receive/) · first in G.35 · ×1
- https://rholang.org/docs/rholang-cost-accounting (https://rholang.org/docs/rholang-cost-accounting) · first in G.35 · ×1

bend-lang.com (5)

- https://bend-lang.com/install.sh (https://bend-lang.com/install.sh) · first in A.2 · ×16
- https://bend-lang.com (https://bend-lang.com) · first in F.1 · ×17
- https://bend-lang.com/bender/terms#s18 (https://bend-lang.com/bender/terms#s18) · first in F.5 · ×12
- https://bend-lang.com/#lab (https://bend-lang.com/#lab) · first in F.29 · ×7
- https://bend-lang.com/bendai/terms#s18 (https://bend-lang.com/bendai/terms#s18) · first in H.2 · ×1

lean-lang.org (5)

- https://lean-lang.org/papers/lean4.pdf (https://lean-lang.org/papers/lean4.pdf) · first in H.11 · ×1
- https://lean-lang.org/use-cases/mathlib/ (https://lean-lang.org/use-cases/mathlib/) · first in H.11 · ×1
- https://lean-lang.org/doc/reference/latest/Run-Time-Code/Reference-Counting/ (https://lean-lang.org/doc/reference/latest/Run-Time-Code/Reference-Counting/) · first in H.22 · ×1
- https://lean-lang.org/doc/reference/latest/IO/Tasks-and-Threads/ (https://lean-lang.org/doc/reference/latest/IO/Tasks-and-Threads/) · first in H.22 · ×1
- https://lean-lang.org/doc/reference/latest/ (https://lean-lang.org/doc/reference/latest/) · first in H.22 · ×1

www.reddit.com (http://www.reddit.com) (5)

- https://www.reddit.com/r/bendlang/ (https://www.reddit.com/r/bendlang/) · first in F.10 · ×11
- https://www.reddit.com/user/SrPeixinho/ (https://www.reddit.com/user/SrPeixinho/) · first in F.32 · ×4
- https://www.reddit.com/r/haskell/comments/2zqtfk/why_isnt_anyone_talking_about_optimal_lambda/ (https://www.reddit.com/r/haskell/comments/2zqtfk/why_isnt_anyone_talking_about_optimal_lambda/) · first in H.41 · ×1
- https://www.reddit.com/r/haskell/comments/568gtk/why_the_overall_lack_of_interest_in_interaction/ (https://www.reddit.com/r/haskell/comments/568gtk/why_the_overall_lack_of_interest_in_interaction/) · first in H.41 · ×1
- https://www.reddit.com/user/matt-noonan (https://www.reddit.com/user/matt-noonan) · first in H.48 · ×1

docs.modular.com (4)

- https://docs.modular.com/mojo/manual/gpu/intro-tutorial/ (https://docs.modular.com/mojo/manual/gpu/intro-tutorial/) · first in H.12 · ×1
- https://docs.modular.com/mojo/manual/values/ownership/ (https://docs.modular.com/mojo/manual/values/ownership/) · first in H.22 · ×1
- https://docs.modular.com/mojo/std/algorithm/backend/cpu/parallelize/parallelize/ (https://docs.modular.com/mojo/std/algorithm/backend/cpu/parallelize/parallelize/) · first in H.22 · ×1
- https://docs.modular.com/mojo/manual/gpu/fundamentals/ (https://docs.modular.com/mojo/manual/gpu/fundamentals/) · first in H.22 · ×1

miro.medium.com (4)

- https://miro.medium.com/max/1400/1*o12r581ZcQ1k2CoaVX9G5Q.png (https://miro.medium.com/max/1400/1*o12r581ZcQ1k2CoaVX9G5Q.png) · first in H.48 · ×1
- https://miro.medium.com/max/1400/1*opcLp8r3HQDgz804Ypzv1g.png (https://miro.medium.com/max/1400/1*opcLp8r3HQDgz804Ypzv1g.png) · first in H.48 · ×1
- https://miro.medium.com/max/830/1*96aAfAWW-Q7GdJpY2P281Q.png (https://miro.medium.com/max/830/1*96aAfAWW-Q7GdJpY2P281Q.png) · first in H.48 · ×1
- https://miro.medium.com/max/1400/1*Vdf-gN8Wq1Kj80AdZiqo_g.png (https://miro.medium.com/max/1400/1*Vdf-gN8Wq1Kj80AdZiqo_g.png) · first in H.48 · ×1

twitter.com (4)

- https://twitter.com/HigherOrderComp (https://twitter.com/HigherOrderComp) · first in F.30 · ×1
- https://twitter.com/VictorTaelin (https://twitter.com/VictorTaelin) · first in F.32 · ×1
- https://twitter.com/victortaelin (https://twitter.com/victortaelin) · first in F.55 · ×3
- https://twitter.com/VictorTaelin/status/1744788091833917696 (https://twitter.com/VictorTaelin/status/1744788091833917696) · first in H.49 · ×1

hub.bend-lang.com (3)

- https://hub.bend-lang.com (https://hub.bend-lang.com) · first in A.17 · ×23
- https://hub.bend-lang.com/packages (https://hub.bend-lang.com/packages) · first in C.1 · ×2
- https://hub.bend-lang.com/0x729eecea86ea5a2cdba3a2856a313bca?f=bolt%2Fconfig.bend (https://hub.bend-lang.com/0x729eecea86ea5a2cdba3a2856a313bca?f=bolt%2Fconfig.bend) · first in F.55 · ×2

i.imgur.com (3)

- https://i.imgur.com/tMhe73c.png (https://i.imgur.com/tMhe73c.png) · first in H.41 · ×1
- https://i.imgur.com/Dxf9qRK.png (https://i.imgur.com/Dxf9qRK.png) · first in H.41 · ×1
- https://i.imgur.com/HjsjqlE.png (https://i.imgur.com/HjsjqlE.png) · first in H.41 · ×1

www.youtube.com (http://www.youtube.com) (3)

- https://www.youtube.com/watch?v=sDPuQ-UjhVQ (https://www.youtube.com/watch?v=sDPuQ-UjhVQ) · first in F.32 · ×4
- https://www.youtube.com/watch?v=HCOQmKTFzYY (https://www.youtube.com/watch?v=HCOQmKTFzYY) · first in F.32 · ×4
- https://www.youtube.com/watch?v=NaytZOiX3fs (https://www.youtube.com/watch?v=NaytZOiX3fs) · first in F.32 · ×4

builtwithbend.com (2)

- https://builtwithbend.com/ (https://builtwithbend.com/) · first in C.1 · ×2
- https://builtwithbend.com/?q=http&category=libraries` (https://builtwithbend.com/?q=http&category=libraries%60) · first in C.31 · ×1

doc.rust-lang.org (2)

- https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html) · first in H.22 · ×1
- https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html (https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html) · first in H.22 · ×1

docs.python.org (2)

- https://docs.python.org/3/howto/free-threading-python.html (https://docs.python.org/3/howto/free-threading-python.html) · first in H.22 · ×1
- https://docs.python.org/3/library/concurrent.futures.html (https://docs.python.org/3/library/concurrent.futures.html) · first in H.22 · ×1

docs.tezos.com (2)

- https://docs.tezos.com/smart-contracts/languages/michelson (https://docs.tezos.com/smart-contracts/languages/michelson) · first in G.35 · ×1
- https://docs.tezos.com/smart-contracts (https://docs.tezos.com/smart-contracts) · first in G.35 · ×1

downloads.haskell.org (2)

- https://downloads.haskell.org/ghc/latest/docs/users_guide/using-concurrent.html (https://downloads.haskell.org/ghc/latest/docs/users_guide/using-concurrent.html) · first in H.22 · ×1
- https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/types.html (https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/types.html) · first in H.22 · ×1

higherorderco.com (2)

- https://higherorderco.com (https://higherorderco.com) · first in F.32 · ×7
- https://higherorderco.com/ (https://higherorderco.com/) · first in H.26 · ×1

news.ycombinator.com (2)

- https://news.ycombinator.com/item?id=40390287 (https://news.ycombinator.com/item?id=40390287) · first in F.32 · ×9
- https://news.ycombinator.com/edit?id=35342297 (https://news.ycombinator.com/edit?id=35342297) · first in H.27 · ×1

ocaml.org (2)

- https://ocaml.org/manual/5.5/parallelism.html#parallel-garbage-collection (https://ocaml.org/manual/5.5/parallelism.html#parallel-garbage-collection) · first in H.22 · ×1
- https://ocaml.org/manual/5.5/parallelism.html (https://ocaml.org/manual/5.5/parallelism.html) · first in H.22 · ×1

paper.higherorderco.com (2)

- https://paper.higherorderco.com (https://paper.higherorderco.com) · first in F.31 · ×8
- https://paper.higherorderco.com/ (https://paper.higherorderco.com/) · first in H.26 · ×1

127.0.0.1:7070 (1)

- http://127.0.0.1:7070 (http://127.0.0.1:7070) · first in F.58 · ×1

HigherOrderCO.com (1)

- https://HigherOrderCO.com/ (https://HigherOrderCO.com/) · first in H.28 · ×1

adrielsantana.github.io (1)

- https://adrielsantana.github.io/metal-bending/ (https://adrielsantana.github.io/metal-bending/) · first in A.17 · ×3

api.marketdata.app (1)

- https://api.marketdata.app/v1/options/expirations/GME/?date=2026-10-29 (https://api.marketdata.app/v1/options/expirations/GME/?date=2026-10-29) · first in C.31 · ×1

arxiv.org (1)

- https://arxiv.org/pdf/1701.04240 (https://arxiv.org/pdf/1701.04240) · first in H.41 · ×1

ayushsomani.me (1)

- https://ayushsomani.me/ (https://ayushsomani.me/) · first in H.1 · ×1

c.tenor.com (1)

- https://c.tenor.com/md3foOULKGIAAAAC/magic.gif (https://c.tenor.com/md3foOULKGIAAAAC/magic.gif) · first in H.46 · ×1

chart-studio.plotly.com (1)

- https://chart-studio.plotly.com/ (https://chart-studio.plotly.com/) · first in H.27 · ×1

cleto.dev (1)

- https://cleto.dev/minebender/ (https://cleto.dev/minebender/) · first in C.34 · ×1

dev.risczero.com (1)

- https://dev.risczero.com/terminology (https://dev.risczero.com/terminology) · first in G.35 · ×1

developer.nvidia.com (1)

- https://developer.nvidia.com/cuda-downloads?target_os=Linux (https://developer.nvidia.com/cuda-downloads?target_os=Linux) · first in H.26 · ×1

discord.HigherOrderCO.com (1)

- https://discord.HigherOrderCO.com/ (https://discord.HigherOrderCO.com/) · first in H.52 · ×1

discord.bend-lang.com (1)

- https://discord.bend-lang.com (https://discord.bend-lang.com) · first in F.10 · ×11

discord.com (1)

- https://discord.com/channels/912426566838013994/915345481675186197/1311434500911403109 (https://discord.com/channels/912426566838013994/915345481675186197/1311434500911403109) · first in H.51 · ×1

discord.higherorderco.com (1)

- https://discord.higherorderco.com (https://discord.higherorderco.com) · first in F.32 · ×5

docs.golem.network (1)

- https://docs.golem.network/docs/creators/javascript/task-executor/task-model (https://docs.golem.network/docs/creators/javascript/task-executor/task-model) · first in G.35 · ×1

docs.google.com (1)

- https://docs.google.com/viewer?url=https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf (https://docs.google.com/viewer?url=https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf) · first in F.32 · ×3

docs.openzeppelin.com (1)

- https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts (https://docs.openzeppelin.com/contracts/5.x/learn/upgrading-smart-contracts) · first in F.39 · ×6

docs.oracle.com (1)

- https://docs.oracle.com/en/java/javase/26/core/virtual-threads.html (https://docs.oracle.com/en/java/javase/26/core/virtual-threads.html) · first in H.22 · ×1

docs.ton.org (1)

- https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades (https://docs.ton.org/blockchain-basics/contract-dev/techniques/upgrades) · first in F.39 · ×6

ethereum.org (1)

- https://ethereum.org/developers/docs/smart-contracts/upgrading (https://ethereum.org/developers/docs/smart-contracts/upgrading) · first in F.39 · ×6

homepage.divms.uiowa.edu (1)

- https://homepage.divms.uiowa.edu/~astump/papers/fu-stump-rta-tlca-14.pdf (https://homepage.divms.uiowa.edu/~astump/papers/fu-stump-rta-tlca-14.pdf) · first in H.41 · ×1

imgur.com (1)

- https://imgur.com/a/Zb0W7FQ (https://imgur.com/a/Zb0W7FQ) · first in H.41 · ×1

instagram.com (1)

- https://instagram.com/victortaelin (https://instagram.com/victortaelin) · first in F.32 · ×4

kindelia.org (1)

- https://kindelia.org (https://kindelia.org) · first in F.38 · ×6

learn.microsoft.com (1)

- https://learn.microsoft.com/en-us/windows/wsl/install (https://learn.microsoft.com/en-us/windows/wsl/install) · first in H.26 · ×2

mathlib-initiative.org (1)

- https://mathlib-initiative.org/roadmap/ (https://mathlib-initiative.org/roadmap/) · first in H.11 · ×1

medium.com (1)

- https://medium.com/@maiavictor/some-functions-may-have-negative-complexity-im-worried-for-my-crypto-c53f6e7343d3 (https://medium.com/@maiavictor/some-functions-may-have-negative-complexity-im-worried-for-my-crypto-c53f6e7343d3) · first in H.48 · ×2

nix.dev (1)

- https://nix.dev/concepts/flakes.html (https://nix.dev/concepts/flakes.html) · first in G.35 · ×1

np.github.io (1)

- https://np.github.io/bend-workshop/ (https://np.github.io/bend-workshop/) · first in H.1 · ×1

pauillac.inria.fr (1)

- http://pauillac.inria.fr/~levy/pubs/80curry.pdf (http://pauillac.inria.fr/~levy/pubs/80curry.pdf) · first in F.31 · ×8

pbs.twimg.com (1)

- https://pbs.twimg.com/media/GArVWfTW4AASNfQ?format=png (https://pbs.twimg.com/media/GArVWfTW4AASNfQ?format=png) · first in H.49 · ×1

pdf.sciencedirectassets.com (1)

- https://pdf.sciencedirectassets.com/272575/1-s2.0-S0890540100X00600/1-s2.0-S0890540197926432/main.pdf?X-Amz-Security-Token=IQoJb3JpZ2luX2VjEEkaCXVzLWVhc3QtMSJGMEQCIDQQ1rA8jV9UQEKlaFzLfqlNP%2B%2FrcTj4NCbI40n%2FrGbyAiBLG%2FpnaXmp6BGaG1YpIr3YYsHzxet6aQXc1qnkbb3W0irMBAhSEAUaDDA1OTAwMzU0Njg2NSIM1FOMCDHcoyvFFAU6KqkEgJPcH6B8%2BRYsdLtUERtVlwtcXZBW38xnvb%2FPRSkmxcNaK%2BmQTa7L3ZFuZt9rpNjrB3sJHg%2Bxc%2FqdAF%2FsthEb1NreHNze7LmbStuRufZCGEcxax%2FsyjnSb9bnrHuDEpnck1Dhk2YPqR8%2Bim%2BdQisDUp%2F4torZsCK1%2BPAEQkQAmGqinioexAr8dEE0B0lHgxBz5YRlkV9pjLoq%2FjWFqiUSO2bPdVi2AfpDbXI48ek6gQs%2F6VTIFRShfezfAr1HoDlQEoyyVYnVy6wI%2Fu1WVB%2FA0JJHK1B7rZFEYiIPSAdUpVSOvjhNHN9eIxIxIFX6h0Zz3YJ4QDeLCPztfMCIYYxAex6hoBBVzTkRzszs18hK1K%2FMUMwF4o%2FDy1i3WLeUmC36CL7WXDik%2BTZ7WjJNYGVRILH6cDsHrg17A0MVI5njvw7iM%2FrYKoOgBD2ESct4nO3mpRkKVq%2F9UyKScwVT5VrNpuLWLnrg29BDvE%2BDoFI6c71cisENjhIhGPNrBCQvZLNe1k%2BD54NyfqOe4a1DguuzxBnsNj6BBD2IM6TyDvCz9w36u194aN8oks9hLuTuKp7Rk05dTt6rj4pThkHA%2FQQymmx74MlQtTXTnD5v%2F%2BmGSUz6vHzqaV2Ft5xjWf9w9NJHfTkFkpxNEv8fTUUSMBEhL4nF8wj0wiNbSwp9NvPOj3YMlG2icNxdAZyNsJYJUowOCXi4JTwCkqb2WdN0i88pOSaAautZrBg7nzCKyuCbBjqqAT0zXItndBn%2Be6oyH2l8sD%2B5v%2FjlqCz8%2Bx%2Bz%2FZA3dntddFac64iWFGPbJeRGw05BiPX5TKBnrR%2BmaqfO%2F7SxoYfTV4hI5Z2ImJcoiEd%2BWUmNK2wntMlGtFn%2FmFeeljKBeMxnfh8DN0qRz10NZAfxhvqxAEBu67G0ZXpECGxr8fAiBrdvnEac6rWfv8%2FT0VA%2Fu6xjlMlrrwU65xAuVuIG%2BXpsdC073VLm1%

2BEW&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20221119T011901Z&X-Amz-SignedHeaders=host&X-Amz-Expires=300&X-Amz-Credential=ASIAQ3PHCVTYYRK5XVMW%2F20221119%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Signature=74892553e56ba432350974a6f4dbebfd97418e2187a5c4e183da61dd0e951609&hash=bc1de316d0b6ee58191106c1cdbc34d1eaeab536a9bbc02dfae09818a8cc2510&host=68042c943591013ac2b2430a89b270f6af2c76d8dfd086a07176afe7c76c2c61&pii=S0890540197926432&tid=spdf-00500b38-a41c-4d5b-98bb-4a2754da3953&sid=17532fa99b4522476f2b00d636dc838e7e36gxrqa&type=client&ua=515904515402570a0401&rr=76c51d7eea7b4d36 (https://pdf.sciencedirectassets.com/272575/1-s2.0-S0890540100X00600/1-s2.0-S0890540197926432/main.pdf?X-Amz-Security-Token=IQoJb3JpZ2luX2VjEEkaCXVzLWVhc3QtMSJGMEQCIDQQ1rA8jV9UQEKlaFzLfqINP%2B%2FrcTj4NCbI40n%2FrGbyAiBLG%2FpnaXmp6BGaG1Yplr3YYsHzxet6aQXc1qnkbb3W0irMBAhSEAUaDDA1OTAwMzU0Njg2NSIM1FOMCDHcoyvFFAU6KqkEgJPcH6B8%2BRYsdLtUERtVlwtcXZBW38xnvb%2FPRSkmxcNaK%2BmQTa7L3ZFuZt9rpNjrB3sJHg%2Bxc%2FqdAF%2FsthEb1NreHNze7LmbStuRufZCGEcxax%2FsyjnSb9bnrHuDEpnck1Dhk2YPqR8%2Bim%2BdQisDUp%2F4torZsCK1%2BPAEQkQAmGqinioexAr8dEE0BOlHgxBz5YRIkV9pjLoq%2FjWFqiUSO2bPdVi2AfpDbXI48ek6gQs%2F6VTIFRShfezfAr1HoDlQEoyyVYnVy6wI%2Fu1WVB%2FA0JJHK1B7rZFEYilPSAdUpVSOvjhNHN9elxIxlFX6hOZz3YJ4QDeLCPztfMClYYxAex6hoBBVzTkRzszs18hK1K%2FMUMwF4o%2FDy1i3WLeUmC36CL7WXDik%2BTZ7WjJNYGVRILH6cDsHrg17A0MVI5njvw7iM%2FrYKoOgBD2ESct4nO3mpRkKVq%2F9UyKScwVT5VrNpuLWLnrg29BDvE%2BDoFI6c71cisENjhIhGPNrBCQvZLNe1k%2BD54NyfqOe4a1DguuzxBnsNj6BBD2lM6TyDvCz9w36u194aN8oks9hLuTuKp7Rk05dTt6rj4pThkHA%2FQQymmx74MlQtTXTnD5v%2F%2BmGSUz6vHzqaV2Ft5xjWf9w9NJHfTkFkpxNEv8fTUUSMBEhL4nF8wj0wiNbSwp9NvPOj3YMIG2icNxdAZyNsJYJUowOCXi4JTwCkqb2WdNOi88pOSaAautZrBg7nzCKyuCbBjqqATOzXItndBn%2Be6oyH2l8sD%2B5v%2FjIqCz8%2Bx%2Bz%2FZA3dntddFac64iWFGPbJeRGw05BiPX5TKBnrR%2BmaqfO%2F7SxoYfTV4hl5Z2lmJcoiEd%2BWUmNK2wntMlGtFn%2FmFeeljKBeMxnfh8DN0qRz10NZAfxhvqxAEBu67G0ZXpECGxr8fAiBrdvnEac6rWfv8%2FT0VA%2Fu6xjIMIrrwU65xAuVuIG%2BXpsdC073VLm1%2BEW&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20221119T011901Z&X-Amz-SignedHeaders=host&X-Amz-Expires=300&X-Amz-Credential=ASIAQ3PHCVTYYRK5XVMW%2F20221119%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Signature=74892553e56ba432350974a6f4dbebfd97418e2187a5c4e183da61dd0e951609&hash=bc1de316d0b6ee58191106c1cdbc34d1eaeab536a9bbc02dfae09818a8cc2510&host=68042c943591013ac2b2430a89b270f6af2c76d8dfd086a07176afe7c76c2c61&pii=S0890540197926432&tid=spdf-00500b38-a41c-4d5b-98bb-4a2754da3953&sid=17532fa99b4522476f2b00d636dc838e7e36gxrqa&type=client&ua=515904515402570a0401&rr=76c51d7eea7b4d36) · first in H.27 · ×1

pedrovioliv.github.io (1)

- https://pedrovioliv.github.io/bend-crater/ (https://pedrovioliv.github.io/bend-crater/) · first in C.1 · ×2

s3 (1)

- https://s3 (https://s3) · first in F.58 · ×1

sh.rustup.rs (1)

- https://sh.rustup.rs (https://sh.rustup.rs) · first in H.26 · ×3

stackoverflow.com (1)

- https://stackoverflow.com/questions/31707614/why-are-%CE%BB-calculus-optimal-evaluators-able-to-compute-big-modular-exponentiation (https://stackoverflow.com/questions/31707614/why-are-%CE%BB-calculus-optimal-evaluators-able-to-compute-big-modular-exponentiation) · first in H.41 · ×3

tromp.github.io (1)

- https://tromp.github.io/cl/Binary_lambda_calculus.html (https://tromp.github.io/cl/Binary_lambda_calculus.html) · first in H.52 · ×1

wiki.haskell.org (1)

- https://wiki.haskell.org/Introduction_to_IO (https://wiki.haskell.org/Introduction_to_IO) · first in F.16 · ×3

www.amazon.com (http://www.amazon.com) (1)

- https://www.amazon.com/Implementation-Functional-Programming-Languages-Theoretical/dp/0521621127 (https://www.amazon.com/Implementation-Functional-Programming-Languages-Theoretical/dp/0521621127) · first in H.43 · ×3

www.cambridge.org (http://www.cambridge.org) (1)

- https://www.cambridge.org/core/services/aop-cambridge-core/content/view/1F2763B0F931680F9B15BDC750BEB343/S0956796800001994a.pdf/bologna_optimal_higherorder_machine.pdf (https://www.cambridge.org/core/services/aop-cambridge-core/content/view/1F2763B0F931680F9B15BDC750BEB343/S09

56796800001994a.pdf/bologna_optimal_higherorder_machine.pdf) · first in H.41 · ×1

www.cas.mcmaster.ca (http://www.cas.mcmaster.ca) (1)

- http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/ (http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/) · first in H.27 · ×1

www.cs.umd.edu (http://www.cs.umd.edu) (1)

- https://www.cs.umd.edu/class/fall2017/cmsc330/lectures/02-lambda-calc-encodings.pdf (https://www.cs.umd.edu/class/fall2017/cmsc330/lectures/02-lambda-calc-encodings.pdf) · first in H.48 · ×1

www.modular.com (http://www.modular.com) (1)

- https://www.modular.com/blog/the-next-big-step-in-mojo-open-source (https://www.modular.com/blog/the-next-big-step-in-mojo-open-source) · first in H.12 · ×1

www.researchgate.net (http://www.researchgate.net) (1)

- https://www.researchgate.net/publication/235778993_The_optimal_implementation_of_functional_programming_languages (https://www.researchgate.net/publication/235778993_The_optimal_implementation_of_functional_programming_languages) · first in H.27 · ×2

www.sciencedirect.com (http://www.sciencedirect.com) (1)

- https://www.sciencedirect.com/science/article/pii/S0890540197926432 (https://www.sciencedirect.com/science/article/pii/S0890540197926432) · first in C.40 · ×12

www.semanticscholar.org (http://www.semanticscholar.org) (1)

- https://www.semanticscholar.org/paper/Interaction-Combinators-Lafont/6cfe09aa6e5da6ce98077b7a048cb1badd78cc76 (https://www.semanticscholar.org/paper/Interaction-Combinators-Lafont/6cfe09aa6e5da6ce98077b7a048cb1badd78cc76) · first in H.28 · ×1

www.unison-lang.org (http://www.unison-lang.org) (1)

- https://www.unison-lang.org/docs/the-big-idea/ (https://www.unison-lang.org/docs/the-big-idea/) · first in G.35 · ×1

PROVENANCE

Claude edition (2026-11-06)

- Parts **A–I** and **K**: copied byte for byte from `bend-hvm-inet-compendium.md` (v1, assembled 2026-11-06 from the workspace; its own provenance follows below, unchanged).
- Part **L**: HVM4 `6defdfc7dae2a3cca5dd6e74ed0612385b5646a8 Fri May 29 23:42:54 2026 -0300`, cloned 2026-11-06; sha256 of every file:
  - `README.md` `7b5189dbd32c241613de515bb60a1e8d3265a6b78d1075dcf7460f12a4a5a542`
  - `AGENTS.md` `5b25af5425ae2f26fb6ebb92a7b2243268ffd5db36460a84a610216f73186f74`
  - `docs/primer.md` `f5bb698e0e28bd6f2d6b74b170e39542e660d28d20edc74917a6ae05defcfde1`
  - `docs/theory/interaction_calculus.md` `6f201c05cb0ff4fe6e6816b2f93131270fdaec1f7791569395a61c3ef89d4647`
  - `docs/hvm/core.md` `87827527f60a2819a91fefdb78d0aef91c9ff6780c4aa5d5231a1f13f66410ca`
  - `docs/hvm/memory.md` `b3625f5ad17763d658f25a37d347cf79c5301dcc07e34ec6edec05e7754a50bb`
  - `docs/hvm/syntax.md` `5fbbf6be67cd02ca7ca70c524a751c1b8ae0ca61fc96e1c465e549320161a059`
  - `docs/hvm/collapser.md` `9c0cc95831409d1880075ad122c880f8f93fa6057a83855f19c2ed7cb4151da0`
  - `docs/hvm/interactions/alo_cop.md` `a5adcafb5bc1b5cea5eb77fb369c44ed4ef776cc2a796c03f90119bd17f05de7`
  - `docs/hvm/interactions/alo_dry.md` `36551727a31e64c6bdb8586bd8fe333bf22b4bbb84d3600935dbf6a096b008e7`
  - `docs/hvm/interactions/alo_dup.md` `c834965ca5fcce4362425bd04161d941072e46bb6e5f231d13b3e97237e14f5b`
  - `docs/hvm/interactions/alo_lam.md` `cbb635eff68d2117673f20786d0bfc011d9e39f37aa43fd6022d39a03f5f4fc0`
  - `docs/hvm/interactions/alo_nam.md` `579f8297074b0a89e4794e5099e057164dcdfa61886f62968f9eb42b36e97864`
  - `docs/hvm/interactions/alo_nod.md` `5842601e1843d88c945ec972cddf02b3bc7a74713960317b36db6a37a6e49be0`
  - `docs/hvm/interactions/alo_var.md` `c16cd5e2e1a3904e3706467522e4fe77936b76a43a2fb3d2f4d3601e5fa9be04`
  - `docs/hvm/interactions/and_era.md` `0d8d5470af3852cd399c8742c13447276ca028a8bc13163d4b6bfa604860f58a`
  - `docs/hvm/interactions/and_inc.md` `f1a0562049c1e539cbe2cbb7c70594519e2693d3984e1e986b095e9fdc816bff`
  - `docs/hvm/interactions/and_num.md` `8b21b758e83ed64c2f7c7bafaec2cf005ec97656f18a1cec3a1ce375a04f19ca`
  - `docs/hvm/interactions/and_sup.md` `ae3de6e21b6af6da50add3c0f18640d9cab656dd68d60a79db7349a395bdd2ab`
  - `docs/hvm/interactions/app_ctr.md` `22874f02a60411118c1ae9b1593fdbe76429a5e7becd2d882034ce5498332ed5`
  - `docs/hvm/interactions/app_dry.md` `18a4c2cf81e9255c7ecaae43fb118fdba271d8efa1ca7a1aa1113107a63f65d9`
  - `docs/hvm/interactions/app_era.md` `ae44bfff69b93587ebd1ccf1e19c3c743a2cd175559c76a634fd7384b7ffd5db`
  - `docs/hvm/interactions/app_inc.md` `c02e736535cc35603593067223dd799b26c49aedb0f008c3f727e337a07bd614`
  - `docs/hvm/interactions/app_lam.md` `2cb8df6bacd1dd0464d4f12cc1eb082af87f16563248733a6af5c84527cabcd6`
  - `docs/hvm/interactions/app_mat_ctr.md` `cba24df652c5d40f8b23d847f452f4a978e0b6525a5078b38fff4b887764e483`
  - `docs/hvm/interactions/app_mat_num.md` `948fc7644bbf899541303f5acf0dd97251261376d32b8304566253bbeb8562ad`
  - `docs/hvm/interactions/app_mat_sup.md` `aa4a8ea6b6b7483ae3f7e595e94a94798b987042aaed2c7df003e8184fdde4d6`
  - `docs/hvm/interactions/app_nam.md` `bef26d18c4c38ce3e5aa34095b4b9e4b14d698750c7e63f59e84c9b859bfddb3`
  - `docs/hvm/interactions/app_sup.md` `1e77a7da29fcea9f56f49ff517f0d7867713e7e3323b98ad233c6280e3a786cc`
  - `docs/hvm/interactions/ddu_era.md` `d8465000cc188b097e02da32c7d45a7f0cf28eaa2069cedda4990a15d177fa67`
  - `docs/hvm/interactions/ddu_inc.md` `a6bf4a6a2e0911b1781cc339ff508e03679158a985d3ee6e0825904ed09c8498`
  - `docs/hvm/interactions/ddu_num.md` `2151115d31e4256fa05acff68e0e984a30aa47717757ff64a930708e94b0c140`
  - `docs/hvm/interactions/ddu_sup.md` `d3c0e49dc36fcbfb90cb468ee7260313fc1da0037fe6d338bc9117d804d76948`
  - `docs/hvm/interactions/dsu_era.md` `6884e66f98f4599a61a27f92d9112df1ed2980ffc183e05e7b12003ca1de2e43`
  - `docs/hvm/interactions/dsu_inc.md` `fd2d02bf501778bbee88de83192b6b7966cdd57d3fcde2ce3143bd92e13312c5`
  - `docs/hvm/interactions/dsu_num.md` `e23e1764a53627f1e5766b05332fb9ce5a572f473a20d5f2f6cb37da8af96361`
  - `docs/hvm/interactions/dsu_sup.md` `25e3e363bebdcce4a4f91270e2472610834168ef80c5fa7dc54c468f929630ea`
  - `docs/hvm/interactions/dup_dry.md` `345369581ebcb0df4c8395e6f6f08800b4fc6b08250b39bb35f76a55632e2c43`
  - `docs/hvm/interactions/dup_lam.md` `e54e8d3183fb1ab04f639742eebcc3aa6a411182280f4cdcdbb57d885ff98502`
  - `docs/hvm/interactions/dup_nam.md` `1cc75f0c62d2ddb39ccae304dcd5ffba795d4d452387765e024fff6dc8999b87`
  - `docs/hvm/interactions/dup_nod.md` `7cf7e906813b52521ff6bf48a7837d7cb050b070ba942ae5011c6caf92c6634d`
  - `docs/hvm/interactions/dup_sup.md` `843f20cf1512cda50780751285d813234a579463f5613aedc08b0eb6f4b9bc18`
  - `docs/hvm/interactions/eql_any.md` `5be7ce1e03af9f37175a48759688816ced225bf3d5be37f2427a77dce8514b6d`

- docs/hvm/interactions/eql_ctr.md 2266c01b83d3a0df22b1a4020e5cafecb9de321fc9d6788a5eb8cda403f8c00d
- docs/hvm/interactions/eql_dry.md 1f3fa9bae92550ae2494628d38fa97c31b1f9da765a878609ddb9c496ed00fcc
- docs/hvm/interactions/eql_era.md 95af364283558ba35c981ee8811eb021dc8c77ad58ff6cb2a4e7f2876e21808f
- docs/hvm/interactions/eql_inc.md 667d6e16a11f4f4b83f8d748135d223b4d9edc3f68b31e77c84a3195830f402b
- docs/hvm/interactions/eql_lam.md 6fda3622754d9f8ffa97f93af797bd74d73d45a40e7c3f32cd502fa1536fb902
- docs/hvm/interactions/eql_mat.md feef42ba481eb12d72c850b78da88d2c4cdcde54cecbb99f0a86ef911c620c79
- docs/hvm/interactions/eql_nam.md 16f42a2060bd5ec373bd708d5a578d397712149e30a3e873bd4223ae69c5e74d
- docs/hvm/interactions/eql_num.md 0524c59d7d6a8e595a9783453f6ed7904cefaa53006d829681c168d2059a5b8c
- docs/hvm/interactions/eql_sup.md 229616b9d058f92bd6f2bdbd5efb173ad09d6f0c51bf22589d127e1daaa36aa0
- docs/hvm/interactions/eql_use.md 4462dd0ed2020c0624e708da9428d226bce595d961e7bac548d402375673799c
- docs/hvm/interactions/mat_inc.md 4434cf7c19f715cc3f5f55813c2f651894c4be1a2f2ff361043f7f4327e74edb
- docs/hvm/interactions/op2_era.md afd8f400bd4cab805737804cf3ab9df0eac88163874cc673fe8bbb1146177cc6
- docs/hvm/interactions/op2_inc.md 2c3921de50ccc8aaa9851fca87beb2b4f643ea316205c69dffd89899311ef371
- docs/hvm/interactions/op2_num_era.md 47dde3fd6bd6904cb1b016919580ed80b7aeb79537a43577e4e05a2d49bf879d
- docs/hvm/interactions/op2_num_num.md 895d0ecbb68c2330c554fb837fbc4f5b1e724fb1de686c891b94623156180de1
- docs/hvm/interactions/op2_num_sup.md b463a50eb815d49f3235583b5081a3e3dc6e2c8529719cbcfff41bb3de5aff57
- docs/hvm/interactions/op2_sup.md 4648987fb2dc3c88e568dcb5e498a56c166696fa9b36c48d510ae4e72cedfba7
- docs/hvm/interactions/or_era.md e55a41b3161779144a67c6262ee90ef6266f96dcc3fd5d6414ac14aa607e8aa3
- docs/hvm/interactions/or_inc.md db4e80c3b4a32baa75e9adf21efe71c9e0c4315b4a44bd0e8cb855f22145bba1
- docs/hvm/interactions/or_num.md d92946d8c1b20ec09d8acd5fa0a0853dfee55051d30524979fe0e126c491f51c
- docs/hvm/interactions/or_sup.md 75d398e065347e7686eb186310fac282ca675fab591ac7938aadae3f592f7ced
- docs/hvm/interactions/uns.md b2db1050595412dd9a27d2eb0306755ff6ce366ba893b2e7722c33a647f79b31
- docs/hvm/interactions/use_era.md 8aef6f01f7aea73a5e85ae3e26be4fe071a8646f0a2ab4ef112c7c5cae9d6f33
- docs/hvm/interactions/use_inc.md 96e4df6c8cc442f8d80ea4d5ca047519e5ca0bbaf3ed49d8281202e08ff27353
- docs/hvm/interactions/use_sup.md ccb36a4ea884b0330370ec9e3eb8735a3097d4e8bc9eb9f42f3943357c5c207f
- docs/hvm/interactions/use_val.md 4cb6984d6478e96a5aab06f755b4a88c026ad887155cf5401afd69b4bd6bd737

- Part **M**: sources fetched 2026-11-06 from the URLs in each banner; sha256 `HVM2.pdf` `589ac12512472ecc…`, `BendRT.pdf` `ef9f9fbc1a2e3435…`, `BendTT.pdf` `954be63fcd63bffe…`, `Mazza-CombSem-MSCS.pdf` `f637cabc8fa49f0b…`, `Laretto-optimal-reduction.pdf` `e96c6f02861bd284…`, `Borner-optimal-effects.pdf` `bcc4170bb6ce7fd9…`, `HN-30227823.txt` `4b783e2279b9feb3…`; PDF text extracted with pypdf.
- Interaction-Calculus reference: `VictorTaelin/Interaction-Calculus` `e9f0185` (2025-11-13); its README is H.33.
- Licences: Bend is Apache-2.0. The HVM4 and Interaction-Calculus repositories ship no licence text: their files are reproduced here for study, which is not a grant of rights.

v1 provenance (unchanged)

Parts A–I and K hold **376 files / 2.65 MB** of verbatim source. Counts per part:

Part	What	Items	Size	Source root
A	Bend 2 tutorial (complete)	17	201.6 KB	`bend-docs/src/content/tutorial`
B	Bend 2 site pages: learn + notes	21	69.1 KB	`bend-docs/src/content/learn` · `bend-docs/src/content/notes`
C	Bend 2 skills (.opencode → bend2*, verbatim)	49	384.8 KB	`bend-gme/version-1/.opencode/skills`
D	HVM5 Interaction Calculus skill (.opencode → hvm5-ic, verbatim)	16	94.7 KB	`bend-gme/version-1/.opencode/skills/hvm5-ic`
E	Runtime-project skills (.opencode → research-to-release, verbatim)	16	81.8 KB	`bend-gme/version-1/.opencode/skills/research-to-release`
F	Bend2 / HVM5 / INET knowledge base (bend-runtime/knowledge, verbatim)	60	820.6 KB	`bend-runtime/knowledge`

Part	What	Items	Size	Source root
G	HVM5 runtime specifications (bend-runtime/hvm5, verbatim)	66	423.6 KB	`bend-runtime/hvm5`
H	Primary sources: Victor Taelin, HigherOrderCO, bend2.dev, upstream bendlang/bend (verbatim)	55	480.7 KB	`bend-docs/openspec/changes/bend-docs-site/research + fresh fetches`
I	HVM4 runtime documentation (HigherOrderCO/HVM4 clone, verbatim)	71	138.4 KB	`forgecode_clone/hvm`
K	Workspace context: Bend experiments and docs-site design notes	5	15.7 KB	`bend-runtime · bend-branch-graph · bend-branch-lab · bend-docs/openspec`

What was found but deliberately excluded (and why)

Found	Where	Why not included
`bend-runtime/docs/**`	bend-runtime	An older copy of the same tutorial/learn/notes content that is in Parts A/B (only 6 files differ, all version drift; the bend-docs copies are newer)
`bend-runtime/hvm5/version1/legacy/lattice-v0/{slices,checkpoints}/*`, `legacy/ts-oracle`	bend-runtime	Slice/checkpoint execution planning and a TS oracle — no HVM/INET knowledge; the design-level files of that folder are included in Part G
`bend-runtime/hvm5/**` non-doc files (src, bench code, tests, contracts, compiler)	bend-runtime	Code and fixtures, not documents; the READMEs that explain them are included
`bend-video/**`	bend-video	A Remotion video project about the Bend story (the `lattice/` there is the video's editorial plan) — not a knowledge source
`weather/.opencode/skills/bend-hvm`, `lambench-pro/.opencode/skills/bend-hvm`, `effect-hvm-1/.opencode/skills/bend-hvm`	three other projects	Found while scanning; all are 2-line stubs (front matter + title), no content
`effect-hvm-1/**` except `packages/hvm4/docs`	effect-hvm-1	An Effect-TS fork; its `packages/hvm4` is the same HVM4 C runtime clone already included as Part I (docs verified byte-identical to the forgecode_clone copy)
`forgecode_clone/hvm/**` except README/CLAUDE/docs	forgecode_clone	The C sources, clang/, bench, bindings — code; the docs are Part I, and the whole test suite is aggregated at the end of Part I
`bend-game/**`, `bend-docs` site infra, `openspec/changes/bend-docs-site/{tasks,specs}/*`	workspace	Scaffolding and site mechanics; only the docs-site proposal/design are included (Part K)
`bend-docs/openspec/changes/bend-docs-site/research/new-style.txt`	bend-docs	A CSS stylesheet fragment for the docs site, not prose

Known duplicates (kept on purpose)

- The `hvm5-ic` skill (Part D) and the HVM5 v1/v2 specs (Part G) are different edits of overlapping material; same for the knowledge-base rollups `ALL-COMBINED*` (end of Part F) versus the reorganized files that precede them.
- Interactive duplicates: the tutorial (Part A) and the bend2-* skills (Part C) share chapters; the bend2.dev pages (Part H `sources/`) back the tutorial; the HVM4 docs exist twice in the workspace (only one copy was included).
- When two copies disagree, prefer the one with the newer date in its banner; the tutorial and knowledge base were checked against `bend 2.0.25`, the .opencode skills against 2.0.38/2.0.39, and the fresh fetches are `main` on 2026-11-06.