Interaction nets, the Interaction Calculus, HVM4 and HVM5

A coursebook — from "what is a port?" to reading the HVM4 runtime and the HVM5 design. Claude edition, rewritten 2026-11-06. Companion sourcebook: `../compendium/bend-hvm-inet-compendium-claude.md`.

This pair is the whole set. Two documents, nothing else: the **compendium** (every source, verbatim, with section ids `A.1` … `L.n`) and this **coursebook** (one learning path with exercises and worked solutions). Whenever a lesson rests on a source, the compendium id is named — for example *(H.46)* is Taelin's "HVM: how it works", *(L.4)* is HVM4's current `interaction_calculus.md`.

Paper labels. `M.1` – `M.7` name the papers the coursebook cites — M.1 the HVM2 paper, M.2 BendRT, M.3 BendTT, M.4 Mazza's *Denotational Semantics for the Symmetric Interaction Combinators*, M.5 Laretto's seminar on the complexity of optimal reduction, M.6 Borner's thesis on effects, M.7 Taelin's 2022 account of HVM's design. They are labels only: the compendium does not reproduce those texts, and a citation such as *(Mazza, M.4, §2.3)* gives the author and the section.

Contents

- 0 · How to use this book
  - 0.1 Who it is for
  - 0.2 What changed against the first edition
  - 0.3 The legend (used in every diagram)
  - 0.4 Three toolchains
  - 0.5 The first fifteen minutes
  - 0.6 Map of the book
  - 0.7 Honest limits of this book
- 1 · Why copying is the whole game
  - 1.1 Substitution is not one step
  - 1.2 The case memoization cannot handle
  - 1.3 Copy incrementally
- 2 · The model: cells, ports, wires
  - 2.1 Definitions
  - 2.2 The two theorems (and one non-theorem)
  - 2.3 Reading a net
  - 2.4 Vicious circles: when "no active pair" is not an answer
- 3 · The six rules of the symmetric interaction combinators
  - 3.1 The three cells

    - 3.2 Annihilation: γ–γ
    - 3.3 Annihilation: δ–δ
    - 3.4 Annihilation: ε–ε
    - 3.5 Erasure: γ–ε and δ–ε
    - 3.6 Commutation: γ–δ, the copying rule
    - 3.7 The rule table
    - 3.8 A complete hand reduction
    - 3.9 Why three cells are enough, and why that is not the point
- 4 · λ-calculus on nets: β is one interaction
    - 4.1 Two shapes from one cell
    - 4.2 Variables: wires, duplicators, erasers
    - 4.3 Step gallery: `(λx. x) y`
    - 4.4 Step gallery: duplicating a function
    - 4.5 Copying stops at redexes
    - 4.6 The limit of one duplicator kind: the clone of a clone
- 5 · Labels and the Interaction Calculus
    - 5.1 From nets back to terms
    - 5.2 What a label is
    - 5.3 The four core rules, each with a real trace
    - 5.4 The rule summary
    - 5.5 Arithmetic across superpositions: the label decides the shape
    - 5.6 The clone of a clone, measured
    - 5.7 Two dialects, one calculus
- 6 · Superposition as a programming tool
    - 6.1 Two values in one place
    - 6.2 Forks: two versions of a program, sharing what they can
    - 6.3 Enumeration
    - 6.4 Search by erasure
    - 6.5 Infinite superpositions and lazy search
    - 6.6 What superposition is not
- 7 · Algorithms and their counts
    - 7.1 Booleans
    - 7.2 Church numerals and addition
    - 7.3 Peano naturals: two numeral forms that do not mix
    - 7.4 Lists
    - 7.5 The showpiece: sharing inside a lambda
    - 7.6 Where sharing does not help: naive Fibonacci
    - 7.7 Runtime fusion: applying a function 2^{256} times
- 8 · Inside HVM4 (HEAD `6defdfc`)
    - 8.1 One file, eleven sections
    - 8.2 The term word
    - 8.3 The 45 tags

    - 8.4 Book and heap
    - 8.5 The evaluator: WNF, SNF, CNF
    - 8.6 Collapse, and the number after each answer
    - 8.7 The CLI at HEAD
    - 8.8 What changed since March 2026
    - 8.9 New syntax at HEAD, tested
    - 8.10 The 61 rule sheets
- 8B · How HVM4 is built, and why
    - 8B.0 The journey, in Taelin's words
    - 8B.1 The problem: clones
    - 8B.2 Lesson 1 (HVM2): one kind of duplicator is not enough
    - 8B.3 Lesson 2 (HVM2): eager evaluation cannot be lazy
    - 8B.4 Lesson 3: graphs are slow; words are fast
    - 8B.5 A duplication is one shared slot
    - 8B.6 The book: immutable definitions, lazy instances
    - 8B.7 Memory: a bump allocator and no collector
    - 8B.8 Collapse: answers live inside terms
    - 8B.9 A full machine trace: Church 2^2 in 14 interactions
    - 8B.10 One file, one thread
- 9 · Optimality and parallelism, told honestly
    - 9.1 What "optimal" means
    - 9.2 Lamping: the abstract algorithm and the oracle
    - 9.3 Why optimal reduction was not used, and the two reasons
    - 9.4 The remaining gap: the quadratic slowdown
    - 9.4a Effects: where a confluent net meets the world
    - 9.5 Parallelism: what locality gives and what it does not
    - 9.5a Where nets come from: proof nets, linear logic, the geometry of interaction
    - 9.6 A claim-class table
- 10 · The lineage: HVM1 → HVM4, Bend 2, BendRT, HVM5
    - 10.1 The runtimes
    - 10.2 Bend 2 and BendRT: the nets are not in the released compiler
    - 10.3 HVM5 in this workspace: what it is
    - 10.4 Homoiconicity, concretely
    - 10.5 HVM5 status, honestly
- 11 · Victor Taelin's insights, verbatim
    - 11.1 Where to read Taelin first
- 12 · Drills and capstones
    - 12.1 Drill set (short, mixed)
    - 12.2 Capstones
- Appendix A · Reproduce everything in ten minutes
- Appendix B · The 61 HVM4 interaction sheets (verbatim, HEAD `6defdfc`)
- Appendix C · Verified-run ledger (HVM4 HEAD `6defdfc`, 2026-11-06)

- Appendix D · Glossary
- Appendix E · Map to the compendium
- Appendix F · Further reading (links checked 2026-11-06)

0 · How to use this book

0.1 Who it is for

You know a functional language (Haskell, OCaml, Lean, Scala) and the λ-calculus at the level of "β-reduction substitutes the argument for the variable". You do **not** need category theory, linear logic or any prior exposure to graph rewriting. Every concept is introduced before it is used; every exercise can be solved from the text that precedes it.

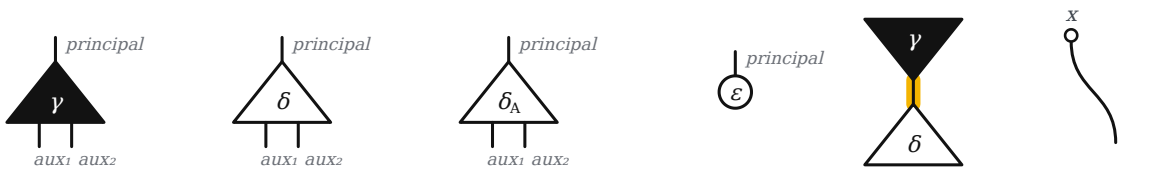
0.2 What changed against the first edition

The first edition (`bend-hvm-inet-coursebook.md`) was audited before this rewrite. The audit ledger is Appendix R of the compendium; the changes that matter to a reader are:

Area	First edition	This edition
HVM4 version	described a March 2026 clone (`2d7f28e`) with a multi-file `clang/` tree, AOT (`--to-c`), FFI and threads	describes **`HEAD 6defdfc` (2026-05-29)**: one sequential C file `src/hvm.c` , flags `-s -S -C[=N] -D -d` ; the removed features are history (§8.8)
Build command	`cd clang && clang -O2 -o main main.c -ldl`	`clang -O2 -o src/hvm src/hvm.c`
Step galleries	hand-drawn only	the six combinator rules drawn port by port, **plus real `-D` traces from HVM4** for every Interaction-Calculus rule and a full annotated 14-step trace of Church 2^2
HVM4 design	absent	Part 8B: why the machine is built this way (labels, lazy WNF stack, words not graphs, the shared DUP slot, book/ALO, memory, collapse), each decision tied to the lesson that caused it and the C code that implements it
Quotes	a handful	15 insight cards with verbatim quotes and their source files (§11); 41 quoted passages, all checked word for word against the sources
Numerals	claimed `3n = #S{#S{#S{#Z{}}}}`	two numeral forms that do **not** mix (§7.4)
Exercises	76	114, each with numbered reasoning steps
Outside sources checked	none	Mazza (symmetric vs Lafont combinators, vicious circles), Asperti–Coppola–Martini and BOHM via Laretto's seminar, Borner on effects, Taelin's 2022 HN account of HVM's design; plus HEAD measurements of runtime fusion (§7.7) and the quadratic slowdown (§9.4)

0.3 The legend (used in every diagram)

INTERACTION NET **Legend (used in every diagram)**



aux$_1$ is on the left and aux$_2$ on the right as seen from the apex; δA carries label A (Part 5). Thick amber wire: an active pair, the only place work happens. Free ports never change during reduction.

Every net is drawn from a one-line **notation**, printed under each figure, and the notation is the source of truth. `γ(a, b)` is a constructor whose aux$_1$ goes to `a` and aux$_2$ to `b`; `ε` is an eraser; `t ~ u` joins two principal ports (an active pair); `x = t` wires free port `x` to the principal of `t`; a name used once is a free port and a name used twice is a wire. So `δ(a, b) ~ γ(ε, ε)` is a duplicator facing a constructor whose two aux ports are erased. Animated players replay each reduction one interaction at a time, with the notation of every intermediate net. Term-level traces use HVM4's own notation, copied from `hvm -D` output; a line `—— APP-LAM` names the interaction that produced the next line.

0.4 Three toolchains

Everything printed in this book was produced by one of these, on **2026-11-06**:

```
# 1 · HVM4 — the machine (HEAD 6defdfc, 2026-05-29)
git clone https://github.com/HigherOrderCO/HVM4 && cd HVM4
clang -O2 -o src/hvm src/hvm.c          # ~4 s; one translation unit, no other files
./devs/test/_all_.sh                     # 217 tests; all pass at HEAD (HVM_TEST_TIMEOUT_SECS=20)
./src/hvm file.hvm -s -C                 # -s stats, -C collapse, -D step trace, -S silent

# 2 · The Interaction Calculus reference — the pure calculus (commit e9f0185)
git clone https://github.com/VictorTaelin/Interaction-Calculus && cd Interaction-Calculus
make && ./bin/ic run file.ic             # add -C to collapse

# 3 · Bend 2 — the language (local compiler 2.0.39)
bend file.bend --check-only && bend file.bend
```

Version rule. Every interaction count in this book says which tool produced it. HVM4's `Itrs` counts the interactions of *its* machine (including numeric and bookkeeping rules), the IC reference's `WORK` counts *its* rules, and the hand-reductions in Parts 2–4 count combinator rewrites. The three numbers measure the same idea at different granularities — never compare them across tools.

0.5 The first fifteen minutes

Build HVM4 (above) and type:

```
printf '@main = (&A{1,2} + 10)\n' > first.hvm
./src/hvm first.hvm -s -C
```

```
11 #3
12 #4
- Itrs: 4 interactions
```

Two answers, four interactions, no loop. `&A{1,2}` is a **superposition** — two values in one place — and `+ 10` was applied to both. (`-C` "collapses" the superposition into a list of answers. The `#3` / `#4` after each answer is the **running interaction count at the moment that answer was printed** — the first world was ready after 3 interactions, the second after 4; §8.6 shows the line of C that prints it.)

Now watch work being **shared inside a lambda**, the thing ordinary lazy languages cannot do:

```
cat > share.hvm <<'EOF'
@f = λx.λy. !z&=x; [(z₀ + z₁), y]
@main = !F& = @f(2); [F₀(10), F₁(20)]
EOF
./src/hvm share.hvm -D
```

The trace (Part 7.6 walks through all 11 lines) contains exactly one line

```
---------------------------------------- OP2-NUM-NUM
```

where `(2 + 2)` becomes `4` — **once** — although the function `@f(2)` was copied and both copies used the sum. That single line is the reason this whole lineage exists. The rest of the book explains how it works, what it costs, and where it does not help.

HVM4 TRACE · start APP-LAM ×3 DUP-LAM ×1 DUP-NOD ×5 OP2-NUM-NUM ×1 DUP-SUP ×1

step ·

```
@main
```

step ·

```
A₀(@{10})<>@{[A₁(20)]};!A&F__a=@{λa.!B&F___=@{2};[(B₀ + B₁),a]};
```

step ·

```
(λa.A₀)(@{10})<>@{[λb.A₁(20)]};!A&F__a=@{!A&F___=@{2};[(A₀ + A₁),&F__a{a,b}]};
```

step ·

```
A₀<>@{[λa.A₁(20)]};!A&F__a=@{!A&F___=@{2};[(A₀ + A₁),&F__a{@{10},a}]};
```

step ·

```
A₀<>B₀<>@{[λa.A₁<>B₁(20)]};!A&F__a=@{(C₀ +
C₁)};!B&F__a=@{[&F__a{@{10},a}]};!C&F___=@{@{2}};
```

step ·

```
A₀<>B₀<>@{[λa.A₁<>B₁(20)]};!A&F__a=(2 + @{2});!B&F__a=@{[&F__a{@{10},a}]};
```

step ·

```
A₀<>B₀<>@{[λa.A₁<>B₁(20)]};!A&F__a=4;!B&F__a=@{[&F__a{@{10},a}]};
```

step ·

```
4<>A₀<>@{[λa.4<>A₁(20)]};!A&F__a=@{[&F__a{@{10},a}]};
```

step ·

```
4<>A₀<>B₀<>@{[λa.4<>A₁<>B₁(20)]};!A&F__a=@{&F__a{@{10},a}};!B&F__a=@{[]};
```

step ·

```
4<>@{10}<>A₀<>@{[λa.4<>a<>A₁(20)]};!A&F__a=@{[]};
```

step ·

```
[4,10]<>@{[λa.4<>a<>[](20)]}
```

step ·

```
[4,10]<>4<>@{20}<>[]<>@{[]}
```

Recorded from HVM4 HEAD `6defdfc` with `-D`; rule names are HVM4's own.

0.6 Map of the book

Part	You learn	You can then
1	why copying is the whole problem	explain why memoization fails inside lambdas

Part	You learn	You can then
2	cells, ports, wires, active pairs, the two theorems	reduce a net by hand and check every step
3	the six rules of the symmetric interaction combinators	read any combinator net reduction
4	λ-calculus as a net: β is one interaction	translate terms to nets and back
5	labels and the Interaction Calculus	predict DUP-SUP, APP-SUP, DUP-LAM; explain the c2/k2 accident
6	superposition as a programming tool	search by erasure; enumerate; reason about collapse
7	algorithms and their counts	interpret `Itrs` honestly
8	the HVM4 machine at HEAD	read `src/hvm.c` and the 61 rule sheets
8B	how HVM4 is built and why: lessons from HVM1/HVM2 turned into design	explain every major data structure and its reason
9	optimality and parallelism, honestly	classify every speed claim
10	the lineage: HVM1→4, BendRT (no nets), HVM5	say what each system actually does
11	Taelin's insights, verbatim	trace each idea to its source
12	capstones	build and audit your own programs

Difficulty marks: ★ direct application · ★★ needs two ideas · ★★★ needs a construction or a proof.

0.7 Honest limits of this book

- **Model ≠ machine.** Parts 2–5 teach interaction nets as a graph model. HVM4 does not store a port graph; it stores 64-bit term words in a heap and implements the same interactions on terms (Part 8). Both views are correct; they are different levels.
- **Counts are not time.** Interaction counts measure work in a model. Wall-clock speed depends on constants, memory and scheduling (Part 9).
- **SupGen is private.** Taelin's program synthesizer built on superpositions has no public source. This book teaches the public mechanism (labels, collapse, erasure) and quotes public claims; it never implies the tool is available.
- **HVM5 is a design in progress** in this workspace (Part 10.5); released Bend 2 runs on BendRT, which is **not** an interaction-net reducer.

<hr>

1 · Why copying is the whole game

1.1 Substitution is not one step

β-reduction says `(λx. b) a → b[x := a]`. Written on paper it looks atomic. In a machine it is not: if `x` occurs *n* times in `b`, substitution must place *n* copies of `a`, and `a` may be arbitrarily large. Taelin states the problem in his manifesto *(H.42)*:

> "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"

Every evaluation strategy is a policy for this copying:

Strategy	Policy	Failure mode
call-by-value	evaluate `a` first, copy the *value*	evaluates arguments that are never used
call-by-name	copy `a` unevaluated	evaluates the same `a` once per use
call-by-need (Haskell)	copy a *pointer* to a shared thunk; evaluate once, overwrite	sharing stops at λ: a thunk *inside a function body* is re-created on every call
optimal (Lévy)	never duplicate a redex	needs a way to copy a *function* without copying the work inside it

1.2 The case memoization cannot handle

```
let f = \y -> (expensive 2) + y   -- 'expensive 2' does not depend on y
in  (f 10, f 20)
```

GHC will often float `expensive 2` out by an optimization, but the language semantics does not promise it, and it is easy to write the version where floating is impossible because the shared part is only discovered *after* the function is partially applied (`@f(2)` in §0.5 is that version). Call-by-need shares *values*; it does not share the *partially evaluated body of a lambda* between two copies of the lambda. The incremental copy of Part 4 does.

The history is older than Haskell. Wadsworth's 1971 thesis introduced **graph reduction**: a variable is a pointer to its argument's tree, so an argument is evaluated once however many times it is used, which is what call-by-need implements today. It is not optimal, and Laretto's seminar *(M.5)* gives the reason with one term, `(λf.(f a)(f b)) (λy. …)`: after one β-step the function `λy. …` is applied twice, so its body must be **unshared**, and every redex inside it is duplicated, although the two copies differ only in their argument. Lévy (1978) defined what optimal sharing would have to achieve; Lamping (1990) found the first algorithm (Part 9).

1.3 Copy incrementally

Taelin's ten-minute explanation *(H.43)* arrives at the answer by elimination: evaluating first wastes work on unused parts; copying first duplicates work. The resolution:

> "Fortunately, it turns out there IS a proper way to do it: just **copy incrementally**."

A copy advances **one layer at a time, only when someone looks**. If a copy reaches a redex before it is reduced, the redex is *not* copied — the copy waits, the redex is reduced once, and the copy continues with the result. That is the whole idea. The next three parts build the machinery that makes "one layer at a time" a precise, local rewrite rule.

Exercises — Part 1

E1.1 ★ Name the evaluation strategy that computes `expensive` zero times, one time and two times in `(λx. 7)(expensive)` and in `(λx. x + x)(expensive)`.

▾ SOLUTION

1. `(λx.7)(expensive)` : by-value evaluates it **once** (needlessly); by-name and by-need **zero** times.
2. `(λx. x + x)(expensive)` : by-value **once**; by-name **twice** (one copy per occurrence); by-need **once** (shared thunk).
3. So by-need dominates on first-order sharing — the trouble is only with sharing inside lambdas (E1.2).

E1.2 ★★ Explain, in two sentences, why `let g = (λy. (2*2) + y) in (g 1, g 2)` computes `2*2` twice under plain call-by-need but `let h = (λy. y + 1)(2*2) in (h, h)` computes it once.

▾ SOLUTION

1. In `h` the shared object is a *value* (an application whose result is a number): the thunk is evaluated once and overwritten.
2. In `g` the shared object is a *lambda*; `2*2` sits inside its body, and each call instantiates the body afresh, creating a new thunk for `2*2` per call. Call-by-need shares closures, not the redexes inside them.

E1.3 ★ Why is substitution "not atomic" while a rewrite that touches a bounded number of nodes is?

▾ SOLUTION

1. Substitution's cost is proportional to (#occurrences × size of argument): unbounded.
2. A rule that reads and writes a fixed number of nodes has a constant cost: it can be a single machine step, and two such steps on disjoint nodes can run in parallel.

E1.4 ★★ "Copy incrementally": suppose we copy the list `[f(1), f(2)]` where `f(1)` is an unevaluated call. What does the incremental copy do when the first copy's consumer asks for the head, and what does the second copy see later?

▾ SOLUTION

1. The copier peels one layer: the outer cons cell becomes two cons cells whose heads are both "still being copied" from the shared `f(1)`.
2. When the first consumer demands its head, `f(1)` is reduced **in the shared position** (to, say, 7), then the copy proceeds into the result.
3. The second copy later finds the already-reduced 7: the call ran once. The tail `[f(2)]` is copied only if someone demands it.

<hr>

2 · The model: cells, ports, wires

2.1 Definitions

An **interaction net** is a graph made of:

- **cells** (also called agents or nodes), each with a *kind* and a fixed number of ports;
- exactly one **principal port** per cell, plus $n \geq 0$ **auxiliary ports**;
- **wires**, each joining exactly two ports; every port is joined to exactly one wire;
- **free ports**: wire ends not attached to any cell — the net's interface.

An **active pair** is a wire joining two principal ports. An **interaction rule** replaces an active pair (the two cells, nothing else) by a new net with the same free ports. There is at most one rule per pair of kinds. That is Lafont's definition (1990), and it has four consequences you should keep at the front of your mind:

1. **Locality.** A rule reads exactly two cells and rewires only their auxiliary connections.
2. **Interface preservation.** The free ports before and after a rewrite are the same; the outside world cannot tell which rule fired, only that one did.
3. **No choice inside a rule.** Each active pair has exactly one rewrite.
4. **Disjointness.** A cell has one principal port, so a cell belongs to at most one active pair. Two different active pairs never share a cell.

2.2 The two theorems (and one non-theorem)

Theorem (strong confluence, "one-step diamond"). If a net N can rewrite to N_1 (firing pair P) and to N_2 (firing pair Q ≠ P), then N_1 and N_2 both rewrite in one step to the same net N_3 (fire Q in N_1, P in N_2).

Why: by consequence 4, P and Q are disjoint; each rule only touches its own two cells; so the two rewrites commute.

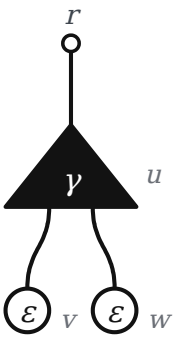
Corollary. All reduction sequences from a net to a normal form (no active pairs) have **the same length**, and the normal form is unique. The number of interactions is a property of the net, not of the scheduler.

Corollary (free parallelism). All active pairs present at one moment can fire simultaneously.

Non-theorem. Confluence does *not* say every net terminates, nor that the number of interactions equals the number of β-steps of some λ-term, nor anything about time. Those questions come back in Part 9.

2.3 Reading a net

INTERACTION NET **A net with one free port r**




```
r = γ(ε, ε)
```

cell u of kind γ, its principal toward r; $u.aux_1$ – v.principal, $u.aux_2$ – w.principal; v and w are erasers · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

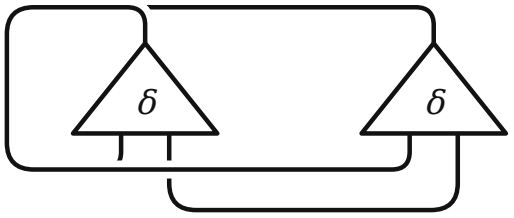
Count: 3 cells, 1 free port, 0 active pairs (u's principal touches the free port, not a principal). This net is in **normal form**.

Invariant check you will use 200 times. Before and after every rewrite, count (a) the free ports — must be unchanged; (b) the ports attached to each wire — exactly two; (c) the cells touched — exactly the two of the active pair.

2.4 Vicious circles: when "no active pair" is not an answer

"No active pair left" does not always mean "finished". Mazza's paper on the symmetric combinators *(M.4, §2.3)* names the exception. A net can contain a cycle in which every cell waits for another, so no cell can ever interact first:

INTERACTION NET **A vicious circle: two δ cells, each principal wired to an aux port of the other**




```
p = δ(δ(p, s), s)
```

no principal ever meets a principal, so nothing can fire; the notation p = δ(δ(p, s), s) shows the cycle through p · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

Mazza calls a path **straight** if it enters a cell through an auxiliary port and leaves through the principal port, or the reverse. A **vicious circle** is a cyclic straight path that never crosses two principal ports in a row. A net with no active pair **and** no vicious circle is **cut-free**: a true result. A net with a vicious circle is deadlocked; it is an error, not a value. A net that reduces to a cut-free net is called **total**.

The same paper states the efficiency consequence of strong confluence (§2.2) in one sentence *(M.4, §2.2)*:

> "All reductions leading from a net to its normal form have the same length and require the same work; in particular, if a net is normalizable, then it is strongly so."

So in interaction nets there is no "good" and "bad" evaluation order for the amount of work, as there is in λ-calculus. This is why HVM4's `Itrs` count is a property of the program (§7).

Exercises — Part 2

E2.6 ★★ A net has no active pair, but `-D` -style inspection shows two duplicator cells whose principal ports each lead into the other's auxiliary port. Is it in normal form? Is it a result?

▾ SOLUTION

1. It is in normal form in the narrow sense: no rule applies.
2. It is not cut-free: the two cells form a vicious circle (a straight cycle that never crosses two principal ports in a row).
3. So it is a deadlock, not a result. Mazza's definitions separate the two: only cut-free nets are the "true" normal forms.

E2.1 ★ A net has cells u (γ), v (δ) with `u.principal — v.principal`, `u.aux1 — v.aux1`, `u.aux2 — x`, `v.aux2 — y`. How many free ports, wires and active pairs does it have?

▾ SOLUTION

1. Free ports: x, y → **2**.
2. Wires: u.p–v.p, u.1–v.1, u.2–x, v.2–y → **4**; ports: 3+3 cell ports + 2 free = 8 = 2×4 ✓.
3. Active pairs: u.p–v.p joins two principals → **1**.

E2.2 ★ Why can a cell never be part of two active pairs at once?

▾ SOLUTION

A cell has exactly one principal port, and that port is attached to exactly one wire. An active pair needs the cell's principal port; so at most one pair.

E2.3 ★★ Use the diamond property to prove: if some reduction of N reaches a normal form in k steps, every reduction of N that reaches a normal form takes exactly k steps.

▾ SOLUTION

1. Induction on k. k = 0: N has no active pairs, so every reduction is empty.
2. Step: let two reductions start with pairs P and Q. If P = Q, apply the hypothesis to the result.
3. If P ≠ Q, the diamond gives N_3 reachable from both N_P and N_Q in one step. N_P normalizes in k−1 steps (first reduction), so by the hypothesis every normalizing reduction of N_P has k−1 steps, including the one through N_3: N_3 normalizes in k−2.
4. Hence N_Q → N_3 → … normalizes in k−1 steps, and by the hypothesis applied to N_Q every normalizing reduction of N_Q has k−1 steps. Total k. ∎

E2.4 ★★ Give a two-cell net that has an active pair but no rule, and explain why interaction systems avoid it.

▾ SOLUTION

1. In a system with kinds α and β but no α–β rule, the net `α.p — β.p` (with their aux ports free) is stuck: an active pair without a rewrite.
2. Interaction *systems* therefore require a rule for every pair of kinds that can meet (or a type discipline that prevents the meeting). The symmetric combinators of Part 3 have rules for all six pairs.

E2.5 ★ "All active pairs present at one moment can fire simultaneously." Why is that not true of β-redexes in a λ-term?

▾ SOLUTION

β-redexes can overlap: in `(λx. (λy. y) x) a` the inner redex lies inside the outer one; firing the outer one copies or moves the inner one. Active pairs never overlap (E2.2), so their rewrites are independent.

<hr>

3 · The six rules of the symmetric interaction combinators

Lafont (1997) introduced two closely related systems with three cells each, and both matter here:

	Lafont's interaction combinators	the symmetric interaction combinators
cells	γ (constructor), δ (duplicator), ε (eraser)	the same three (Mazza writes the constructor as ζ)
γ–γ annihilation	**crossed**: each γ's left port is wired to the other's right port	**straight**: left to left, right to right
δ–δ annihilation	straight	straight
all other rules	identical	identical
universality	every interaction system can be translated into them	every **polarized** interaction system can be translated into them

Mazza, who gave the symmetric combinators a denotational semantics, summarizes the difference *(M.4, abstract)*:

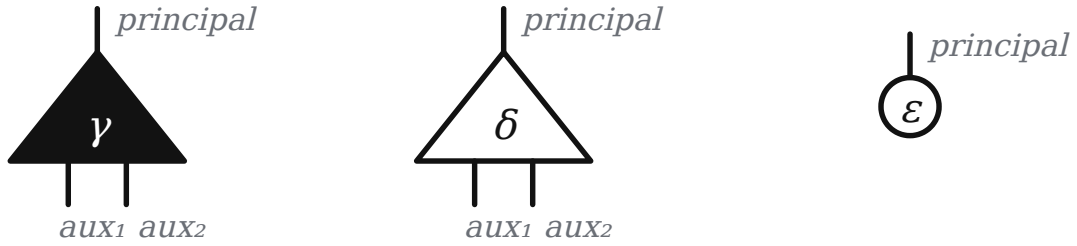
> "They enjoy a weaker universality property with respect to interaction nets, but are equally expressive."

"Polarized" is not a serious restriction in practice: Mazza lists Turing machines, cellular automata, the SK combinators, linear-logic proof nets and the λ-calculus as polarized systems *(M.4, §2.4)*. Interaction nets themselves began as a generalization of linear logic's proof nets *(M.4, §1)*. This book uses the symmetric convention throughout, because it is what the Interaction Calculus and HVM implement:

The one convention. In every annihilation, **aux1 connects to aux1 and aux2 to aux2.**

3.1 The three cells

INTERACTION NET **The three cells**



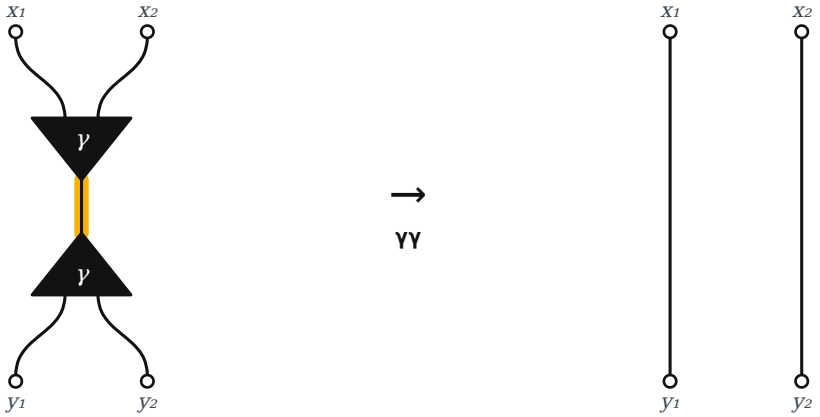
constructor · 2 aux ports | duplicator · 2 aux ports | eraser · 0 aux ports

- **γ**, the *constructor*. Two γ cells annihilate; in Part 4, γ plays both **λ** and **application**, and this annihilation is **β-reduction**.
- **δ**, the *duplicator*. Meeting a different kind, it **copies** that cell; meeting another δ, it annihilates.
- **ε**, the *eraser*. It deletes whatever it meets, one cell at a time.

Six unordered pairs of kinds can meet: γγ, δδ, εε, γδ, γε, δε. Three rules annihilate, two erase, one commutes.

3.2 Annihilation: γ–γ

INTERACTION NET **γ–γ annihilation**



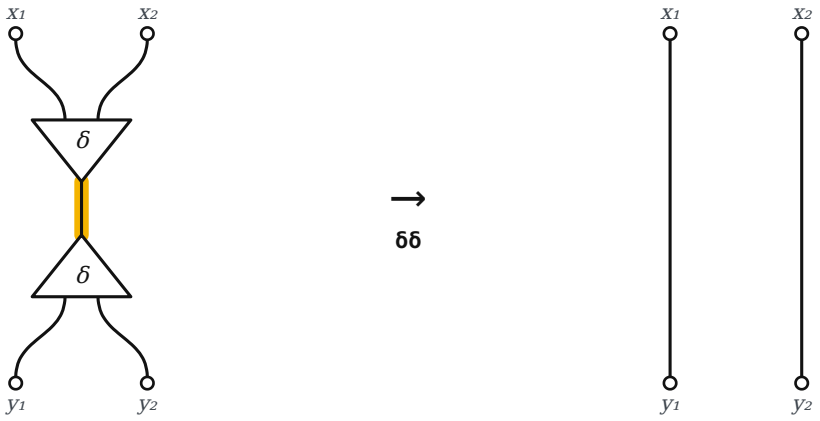

```
γ(x₁, x₂) ~ γ(y₁, y₂) → x₁ = y₁ ; x₂ = y₂
cells 2 → 0 · free ports 4 → 4
```

aux$_1$–aux$_1$, aux$_2$–aux$_2$: cells 2 → 0, free ports 4 → 4 · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

Both cells are drawn with *their own* aux1 on the left as seen from *their own* apex. The lower cell is upside down, so its aux1 appears below x_1. The rule is "aux1–aux1", not "left–left on paper".

3.3 Annihilation: δ–δ

INTERACTION NET **δ–δ annihilation**



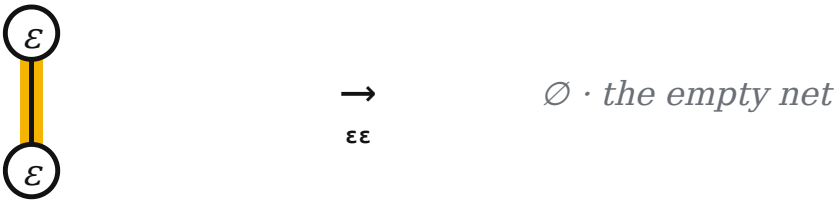
`δ(x₁, x₂) ~ δ(y₁, y₂) → x₁ = y₁ ; x₂ = y₂`
`cells 2 → 0 · free ports 4 → 4`

same shape as γ–γ; δ–δ annihilate only when their labels are equal · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

Same shape. Two duplicators facing each other cancel: a duplication meets the pair it would have built and hands the two halves through. In Part 5 this becomes **DUP-SUP with equal labels: projection**.

3.4 Annihilation: ε–ε

INTERACTION NET **ε–ε annihilation**

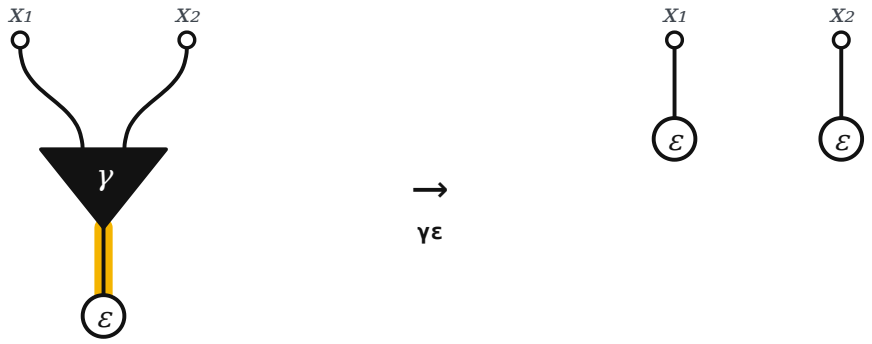


`ε ~ ε → ø (the empty net)`
`cells 2 → 0 · free ports 0 → 0`

both cells vanish: cells 2 → 0, free ports 0 → 0 · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

3.5 Erasure: γ–ε and δ–ε

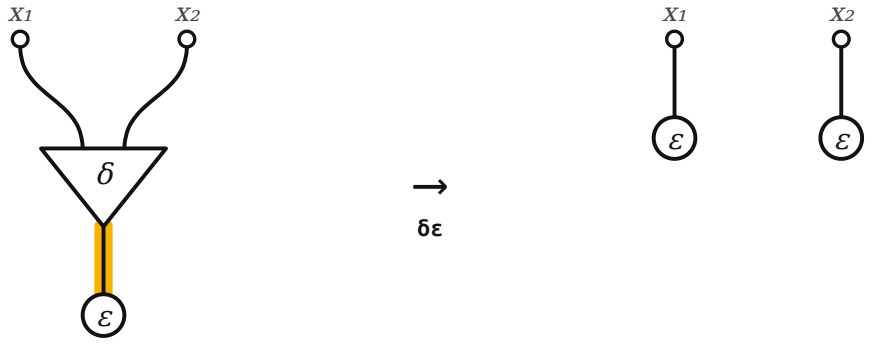
INTERACTION NET **γ–ε erasure**



`γ(x₁, x₂) ~ ε → x₁ = ε ; x₂ = ε`
`cells 2 → 2 · free ports 2 → 2`

one cell deleted, two fresh erasers sent down its aux wires: cells 2 → 2 · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

INTERACTION NET **δ–ε erasure**



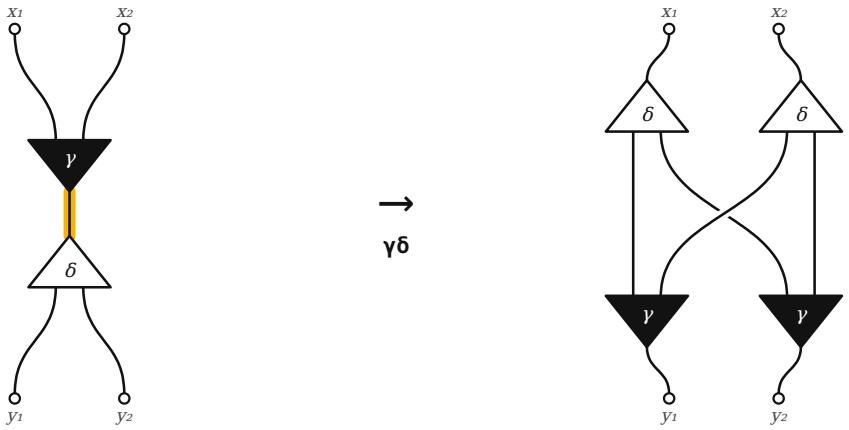
`δ(x₁, x₂) ~ ε → x₁ = ε ; x₂ = ε`
`cells 2 → 2 · free ports 2 → 2`

the same shape with δ in place of γ · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

The eraser does not delete the whole subnet at once. It deletes **one** cell and sends **two** fresh erasers down that cell's auxiliary wires. Erasing a structure of *n* cells costs *n* interactions: the "K erases an unboundedly large structure" of SKI is split into bounded steps. The δ–ε rule has the same shape with δ in place of γ.

3.6 Commutation: γ–δ, the copying rule

INTERACTION NET **γ–δ commutation, the copying rule**



`γ(x₁, x₂) ~ δ(y₁, y₂) → x₁ = δ(p, q) ; x₂ = δ(s, t) ; y₁ = γ(p, s) ; y₂ = γ(q, t)`
`cells 2 → 4 · free ports 4 → 4`

δ-copies face x_1, x_2; γ-copies face y_1, y_2; $\delta_i.\text{aux}_j$ – $\gamma_j.\text{aux}_i$ (two straight wires, two crossing): cells 2 → 4 · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

Read it as: *the δ slides through the γ, and on the way each copies the other.*

- Each wire that left **γ** (x_1, x_2) now leads into the principal of a **copy of δ**.
- Each wire that left **δ** (y_1, y_2) now leads into the principal of a **copy of γ**.
- Cross-wiring: δ-copy *i*'s aux *j* goes to γ-copy *j*'s aux *i*. In the picture δ_1.aux1–γ_1.aux1 and δ_2.aux2–γ_2.aux2 go straight down; δ_1.aux2–γ_2.aux1 and δ_2.aux1–γ_1.aux2 cross.

The new δ principals face x_1 and x_2: if those lead to further γ cells, copying continues one layer further. That is **"copy incrementally"** (§1.3) as a local rule. It is the only rule that grows a net (2 → 4).

3.7 The rule table

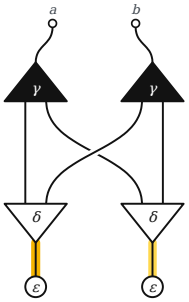
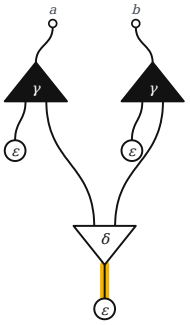
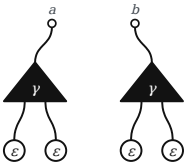
Meeting	Name	Cells	Effect
γ–γ	annihilation	2 → 0	aux1–aux1, aux2–aux2 (β in Part 4)
δ–δ	annihilation	2 → 0	aux1–aux1, aux2–aux2 (projection in Part 5)
ε–ε	annihilation	2 → 0	both vanish
γ–ε, δ–ε	erasure	2 → 2	two erasers move outward
γ–δ	commutation	2 → 4	copy each other, cross-wire

Two groups, or three? The literature sorts the six rules into exactly **two** groups by one test: same kind → **annihilation**, different kinds → **commutation** (Mazza, *M.4*, §2.1). In that convention the erasures γ–ε and δ–ε *are* commutations: ε has no auxiliary ports, so "copying" it across the other cell leaves two erasers. This book lists erasure as its own row because its effect is so different. Two shortcuts that circulate, and are wrong: annihilation is **not** erasure (γ–γ reconnects four wires; it is β), and commutation is **not** only duplication (γ–ε and δ–ε are commutations that erase). The correct functional reading: γ–γ is application, δ–δ is a copier meeting its partner (the end of a sharing), γ–δ is copying, and anything meeting ε is erasure.

3.8 A complete hand reduction

A duplicator copies a constructor whose aux ports are erased:

INTERACTION NET **δ(a, b) ~ γ(ε, ε) to normal form**

step 0 · start	step 1 · γ–δ commutation (copying)	step 2 · δ–ε erasure	step 3 · δ–ε erasure · normal form
			

`δ(a, b) ~ γ(ε, ε) ←`	`a = γ(p, q)`
	`b = γ(s, t)`
	`δ(p, s) ~ ε ←`
	`δ(q, t) ~ ε`
		`a = γ(ε, p)`
		`b = γ(ε, q)`
		`δ(p, q) ~ ε ←`
			`a = γ(ε, ε)`
			`b = γ(ε, ε)`

`3 interactions · free ports a, b unchanged at every step`

step 1 is the commutation; steps 2–3 are the two δ–ε erasures (in either order, by the diamond); free ports a, b never move · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

Step 1, γ–δ commutation. γ-copies take the δ's aux wires (toward a, b); δ-copies take the γ's aux wires (toward the two ε). The notation reads `a = γ(p, q) ; b = γ(s, t) ; δ(p, s) ~ ε ; δ(q, t) ~ ε` : two new active pairs, both δ–ε.

Steps 2–3, two δ–ε erasures (in either order, by the diamond). Each δ-copy dies and sends two erasers up into the γ-copies' aux ports, ending at `a = γ(ε, ε) ; b = γ(ε, ε)` .

Normal form: two copies of "γ with both aux ports erased". **3 interactions; free ports a, b preserved at every step.** The duplicator has finished copying and disappeared; nothing was copied that was not there.

The classic hand-drawing error. In commutation, **γ-copies go to the δ's side and δ-copies to the γ's side.** If you draw the copies on their own side, the new active pairs come out as γ–ε instead of δ–ε and every later count is wrong. Always re-count cells and active pairs after a commutation.

3.9 Why three cells are enough, and why that is not the point

Lafont's universality theorem: every interaction system can be *encoded* in his interaction combinators; for the symmetric combinators used here, every *polarized* system can (Mazza's Theorem 2.2, citing Lafont, *M.4*):

> "Any polarized interaction net system can be translated into the symmetric combinators."

It is a statement about expressiveness, like Turing completeness, and says nothing about how many interactions the encoding takes. Practical machines add cell kinds (numbers, constructors, matches, arithmetic). HVM4 has 45 tags and 61 documented interaction rules (Part 8). The six rules above are their core; the rest is engineering.

What to remember from Part 3

- One convention: **aux1 pairs with aux1.** Check free ports and touched cells at every step.
- Annihilation (γγ, δδ, εε) deletes; γγ will be β.
- Commutation (γδ) copies **one layer per interaction**, and is the only rule that grows the net.
- Erasure costs **one interaction per cell**.
- Universality is about *encodability*, not efficiency. Lafont's combinators (crossed γγ) are universal for all interaction systems; the symmetric ones (straight γγ) for all polarized systems, which includes the λ-calculus.

Exercises — Part 3

E3.8 ★★ Draw the γ–γ annihilation once in Lafont's convention and once in the symmetric convention, for two γ cells with auxiliary ports a, b (top cell) and c, d (bottom cell). Which wires differ?

▾ SOLUTION

1. Symmetric: a–c and b–d (left to left, right to right).
2. Lafont's interaction combinators: a–d and b–c (the auxiliary ports are exchanged).
3. Only γ–γ differs; δ–δ is straight in both, and every other rule is identical. Mixing the two conventions in one hand reduction gives wrong wiring after the first β.

E3.1 ★ Draw the result of a δ–δ annihilation where the upper δ has aux ports p (aux1), q (aux2) and the lower one r (aux1), s (aux2).

▾ SOLUTION

1. Rule: aux1–aux1, aux2–aux2.
2. Result: two wires **p–r** and **q–s**, no cells. Free ports p, q, r, s unchanged.

E3.2 ★ How many interactions does it take to erase a balanced binary tree of 7 γ cells (depth 3) whose 8 leaf wires end in ε cells, starting from an ε touching the root's principal?

▾ **SOLUTION**

1. Each γ–ε deletes one γ and sends two erasers to its children: 7 interactions.
2. The 8 erasers sent to the leaves meet the 8 leaf erasers: 8 ε–ε annihilations.
3. Total **15**, linear in the size of the tree.

E3.3 ★★ Redo step 1 of §3.8 from the rule of §3.6 alone, naming the new cells δ_1, δ_2, γ_1, γ_2 and listing all wires.

▾ **SOLUTION**

1. Old: γ.aux1–ε_1, γ.aux2–ε_2, δ.aux1–a, δ.aux2–b.
2. δ-copies take γ's aux wires: δ_1.p–ε_1.p, δ_2.p–ε_2.p.
3. γ-copies take δ's aux wires: γ_1.p–a, γ_2.p–b.
4. Cross-wiring δ_i.aux$_j$–γ_j.aux$_i$: δ_1.1–γ_1.1, δ_1.2–γ_2.1, δ_2.1–γ_1.2, δ_2.2–γ_2.2.
5. 8 wires; cells γ_1, γ_2, δ_1, δ_2, ε_1, ε_2 = **6**; ports 4·3 + 2·1 + 2 free = 16 = 2·8 ✓. Active pairs: δ_1–ε_1, δ_2–ε_2.

E3.4 ★ Which rules make a net larger, which smaller, which keep the size?

▾ **SOLUTION**

1. Larger: γδ commutation (2 → 4).
2. Smaller: the three annihilations (2 → 0).
3. Same: the two erasures (2 → 2), which push erasers toward cells that will be deleted.

E3.5 ★★ A δ meets a γ whose two aux ports are joined to *each other* by one wire (a loop). What does commutation produce, and what happens next?

▾ **SOLUTION**

1. The δ-copies take γ's aux wires; since those wires are one loop, δ_1.p is joined to **δ_2.p**: a new active pair.
2. The γ-copies face the δ's old aux wires; cross-wiring gives γ_1.1–δ_1.1, γ_1.2–δ_2.1, γ_2.1–δ_1.2, γ_2.2–δ_2.2.
3. δ_1–δ_2 annihilate (aux1–aux1, aux2–aux2): γ_1.1–γ_1.2 and γ_2.1–γ_2.2.
4. Result: **two copies of the loop γ**, in 2 interactions. In Part 4 this is "duplicate the identity function λx.x", and HVM4's trace agrees: DUP-LAM then DUP-SUP (§4.4).

E3.6 ★★★ Prove that erasure terminates on a finite region that is a tree of γ/δ cells with ε leaves (no commutation possible).

▾ SOLUTION

1. Measure: the number of γ/δ cells in the region.
2. Each γε/δε interaction removes one γ/δ cell and creates only ε cells: the measure strictly decreases.
3. εε annihilations remove cells and create none.
4. A strictly decreasing natural number cannot decrease forever. ∎ Commutation is excluded because it is the only rule that creates γ/δ cells.

E3.7 ★ Is γ–δ commutation the same rule as δ–γ commutation?

▾ SOLUTION

Yes. An active pair is an unordered wire between two principal ports; the rule is defined on the unordered pair of kinds. One rule per unordered pair is what makes the system deterministic.

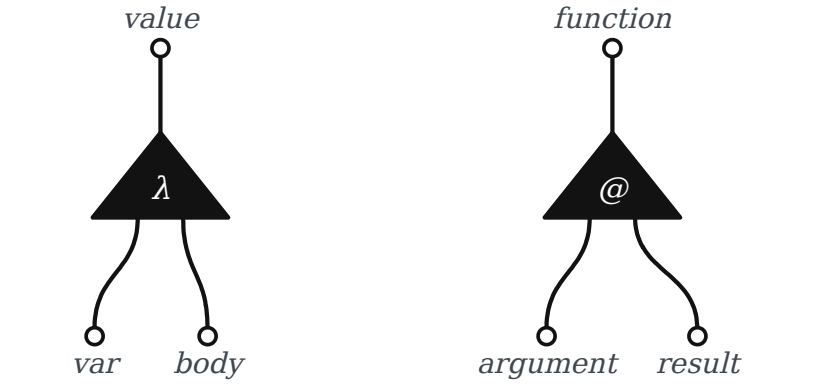
<hr>

4 · λ-calculus on nets: β is one interaction

4.1 Two shapes from one cell

Both λ-abstraction and application are **γ cells**, used with different port roles:

INTERACTION NET **λ and @ are both γ cells**



λ cell · principal = the value

@ cell · principal faces the function

```
value = λ(var, body)
function = @(argument, result)
```

λ: aux_1 = the variable, aux_2 = the body. @: aux_1 = the argument, aux_2 = the result · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

When a λ is applied, the @ cell's principal (facing the function) touches the λ's principal: **an active γ–γ pair**. Annihilation connects aux1–aux1 (**variable ↔ argument**) and aux2–aux2 (**body ↔ result**). That is β-reduction: substitution by wiring, **one interaction, regardless of the size of the argument**.

4.2 Variables: wires, duplicators, erasers

A variable is a wire from the λ's aux1 to its occurrence. The number of occurrences decides the plumbing:

Uses of x	Plumbing at the λ's aux1
1	a single wire (linear)
0	an **ε** cell (the argument will be erased, one cell at a time)
n > 1	**n − 1 δ cells** in a fan, principal toward the λ, aux ports toward the occurrences

The δ's principal faces the binder. Nothing happens until β connects that binder to an argument whose principal faces back; then the δ meets the argument's root cell and copying begins (§3.6).

4.3 Step gallery: `(λx. x) y`

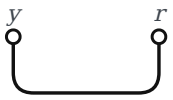
INTERACTION NET **(λx. x) y: one interaction**

step 0 · start



`(λx. x) y`
`@(y, r) ~ λ(x, x) ←`

step 1 · λ–@ annihilation = β · normal form



`y`
`y = r`

`1 interaction · free ports y, r unchanged at every step`

the λ's body is its variable, one loop wire x; β splices var–arg and body–result, leaving r–y · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

Annihilation: var–arg gives `λ.aux1` — `y` ; body–result gives `λ.aux2` — `r` ; λ.aux1 and λ.aux2 were the two ends of one loop wire, so splicing yields **r–y**. One interaction. HVM4 prints the same single step for `(λx.x)(λy.y)` :

```
@main
---------------------------------------- APP-LAM
@{@{λa.a}}
```

(`@{…}` marks terms still in book form, expanded lazily; §8.4.)

4.4 Step gallery: duplicating a function

`!f&A = λx.x; [f₀, f₁]` , "make two copies of the identity function and put them in a list". As a net: a δ (label A; labels are Part 5) whose principal faces the λ of `λx.x` .

1. **γ–δ commutation (DUP-LAM).** Two λ-copies appear at f_0, f_1; two δ-copies appear on the λ's aux wires. Those wires are one loop, so the two δ-copies face each other principal to principal (E3.5).
2. **δ–δ annihilation (DUP-SUP).** The δ-copies cancel; each λ-copy's variable is wired to its own body.

Two identity functions, **2 interactions**:

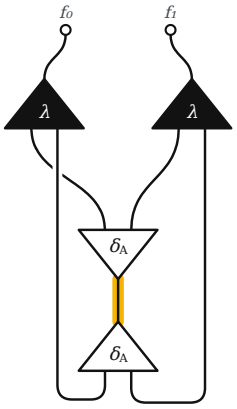
INTERACTION NET **Duplicating λx. x**

step 0 · start



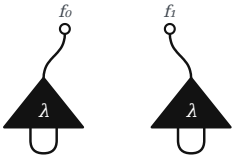

```
!f&A = λx.x; [f₀, f₁]
δA(f₀, f₁) ~ λ(x, x) ←
```

step 1 · λ–δA commutation (copying)




```
(DUP-LAM)
f₀ = λ(p, q)
f₁ = λ(s, t)
δA(p, s) ~ δA(q, t) ←
```

step 2 · δA–δA annihilation · normal form




```
[λa.a, λb.b]
f₀ = λ(p, p)
f₁ = λ(q, q)
```

2 interactions · free ports f₀, f₁ unchanged at every step

step 1 is DUP-LAM (commutation), step 2 DUP-SUP (equal labels annihilate) · γ constructor · δ duplicator · ε eraser · ○ free port · `amber band` = active pair

HVM4 at HEAD:

HVM4 TRACE · start DUP-LAM ×1 DUP-SUP ×1

step ·

```
@main
```

step ·

```
λa.A₀<>@{[λb.A₁]};!A&A=@{&A{a,b}};
```

step ·

```
λa.a<>@{[λb.b]}
```

Recorded from HVM4 HEAD `6defdfc` with `-D` ; rule names are HVM4's own.

```
@main
---------------------------------------- DUP-LAM
λa.A₀<>@{[λb.A₁]};!A&A=@{&A{a,b}};
---------------------------------------- DUP-SUP
λa.a<>@{[λb.b]}
```

Read the middle line slowly. After DUP-LAM the copies are `λa.A₀` and `λb.A₁`: two lambdas whose bodies are the two *sides* of a duplication `!A&A = &A{a,b}`, a duplication of **a superposition of the two copies' variables**. That superposition is the δ-copy seen from the other side (Part 5). The next step is DUP-SUP with equal labels, projection, giving `λa.a` and `λb.b`.

4.5 Copying stops at redexes

When a duplicator walks into a function body and reaches an active pair, it cannot pass: commutation needs a principal facing it, and the redex's two cells face each other. The copy *waits*. The redex reduces **once**, and the duplicator continues into the result. This is §1.3's mechanism, and it is why in §0.5 `(2 + 2)` was computed once for two copies of `@f(2)`.

4.6 The limit of one duplicator kind: the clone of a clone

With a single δ kind, nets for λ-terms go wrong when a duplicator copies a term that *itself contains* a duplicator: an outer δ-copy meets an inner δ and **annihilates** (δδ) where it should have **commuted**. The copies pair up wrong, and the net no longer reads back as the λ-term's normal form. This is the *clone of a clone* problem. Lamping solved it in 1990 with extra control cells, the **oracle**, at a large constant cost (Part 9). The Interaction Calculus solves the common case with **labels**: duplicators with different labels commute instead of annihilating (Part 5).

What to remember from Part 4

- λ and @ are both γ; **β is one γγ annihilation**: var↔arg, body↔result.
- n uses → n−1 δ; zero uses → ε.
- Copying is **incremental and lazy**; a copy waits at a redex, so the redex is reduced once.
- One unlabeled δ cannot handle nested duplication; labels fix the common case.

Exercises — Part 4

E4.1 ★ Translate `λx. λy. x` to a net: list cells and wires.

▾ SOLUTION

1. Cells: λ_1 (binds x), λ_2 (binds y), ε (y unused).
2. Wires: root–λ_1.p; λ_1.aux2–λ_2.p; λ_2.aux2–λ_1.aux1 (the body of λ_2 is x); λ_2.aux1–ε.p.
3. 3 cells, 1 free port, 0 active pairs: a normal form.

E4.2 ★ How many interactions does `(λx. λy. x) a b` take, with a and b free ports? What happens to b?

▾ SOLUTION

1. β_1: x ⇔ a; the result is λ_2, i.e. `λy. a`.
2. β_2: `(λy.a) b` : y ⇔ b. y's wire led to an ε, so now that ε faces b. b is a free port, not a cell: nothing more happens.
3. **2 interactions**; the result carries a; b is attached to an eraser (if b were a term, one erasure per cell would follow).

E4.3 ★★ Draw the net of `λf. λx. f (f x)` (Church 2). How many δ cells, and where?

▾ SOLUTION

1. f occurs twice → **one δ**, principal at λf.aux1, aux ports to the two occurrences.
2. Two @ cells: $@_2$ = `f (…)` outer, $@_1$ = `f x` inner. δ.aux1–$@_2$.p, δ.aux2–$@_1$.p.
3. Wires: root–λf.p; λf.aux2–λx.p; λx.aux2–$@_2$.aux2; $@_2$.aux1–$@_1$.aux2; $@_1$.aux1–λx.aux1.
4. **5 cells** (4 γ, 1 δ), 0 active pairs.

E4.4 ★★ Count the interactions of `(λx. x x)(λy. y)` on the net and check against HVM4.

▾ SOLUTION

1. β: x ⇔ (λy.y); x's δ now faces the λy cell.
2. Copying the identity: DUP-LAM + δδ on the loop = 2 (§4.4).
3. `id id` : one more β.
4. Total **4**. HVM4: `@main = (λ&x. x(x))(λy.y)` reports `λa.a` with `Itrs: 4` (E7.1 asks you to run it).

E4.5 ★★★ Construct a λ-term whose reduction makes a duplicator copy another duplicator, and say what goes wrong with a single unlabeled δ kind.

▾ SOLUTION

1. Church 2 applied to Church 2: `(λf.λx. f (f x)) (λg.λy. g (g y))`. β puts the second numeral (with its own δ for g) where f was; the first numeral's δ starts copying it.
2. The outer δ reaches the inner δ: **δ–δ**. With one δ kind this annihilates, connecting halves pairwise as if the inner fan were the pair being built.
3. The result is not 4 = `λs.λz. s(s(s(s z)))`. With labels (Part 5) the two fans commute when their labels differ; §7.3 measures both cases on HVM4.

<hr>

5 · Labels and the Interaction Calculus

5.1 From nets back to terms

Drawing nets is the best way to *understand* reduction and the worst way to *write programs*. The **Interaction Calculus (IC)** is a term language whose terms are exactly these nets, written as text. Taelin's reference README *(H.33, L.4)* defines it by three departures from the λ-calculus:

> "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*."

and gives the payoff in one sentence:

> "Superpositions and duplications allow the IC to be optimally evaluated, making some computations exponentially faster."

The six forms (HVM4 surface syntax; the IC reference uses the same ideas with slightly different spelling, §5.7):

Form	Syntax	As a net
variable	`x`	a wire
lambda	`λx. t`	γ cell, principal = the lambda
application	`f(a)`	γ cell, principal faces `f`
superposition	`&L{a, b}`	δ_L cell, principal = the pair, aux = a, b
duplication	`!x&L = v; t`	δ_L cell, principal faces `v`, aux = the two sides x_0, x_1
erasure	`&{}`	ε cell

The key line is the pair **superposition / duplication**: they are the *same* cell kind δ_L, read from its two sides. A duplication has its principal facing the value being copied; a superposition has its principal facing whoever consumes the pair. Taelin's README: *"Superpositions work like pairs. Duplications eliminate superpositions through projections."*

5.2 What a label is

A **label** is a name carried by every δ cell. The rule for two δ cells meeting principal-to-principal now depends on it:

- **same label** → annihilate (projection): the duplication *was* the superposition's other half;

- **different labels** → commute (copy): they are two unrelated duplications, and each must copy the other.

This is the local, decidable approximation of Lamping's oracle (§4.6, Part 9): instead of tracking at run time which fan belongs to which, the *program* says so by naming them.

5.3 The four core rules, each with a real trace

The IC has four computational rules plus erasure. Two of them **eliminate** (a constructor meets its matching destructor) and two **propagate** (a constructor passes through an unrelated destructor). All traces below are `hvm -D` at HEAD.

APP-LAM — application meets lambda (β). γ–γ annihilation.

```
(λx.f)(a)                    @main = (λx.x)(λy.y)
--------- APP-LAM             ---------------------------------------- APP-LAM
x ← a                        @{@{λa.a}}
f                            Itrs: 1
```

DUP-SUP — duplication meets superposition. δ–δ. *Same label: projection.*

```
!x&L = &L{a,b}; t            @main = !x&A = &A{1,2}; [x₀, x₁]
----------------- DUP-SUP    ---------------------------------------- DUP-SUP
x₀ ← a                       @{1}<>@{[@{2}]}            result [1,2] · Itrs: 1
x₁ ← b
t
```

Different labels: commutation — each copies the other.

```
!x&L = &R{a,b}; t            @main = !x&A = &B{1,2}; [x₀, x₁]
----------------- DUP-SUP    ---------------------------------------- DUP-SUP
x₀ ← &R{a₀,b₀}               &B{A₀,B₀}<>@{[&B{A₁,B₁}]};!A&A=@{1};!B&A=@{2};
x₁ ← &R{a₁,b₁}               ---------------------------------------- DUP-NOD
!a&L = a                     &B{1,A₀}<>@{[&B{1,A₁}]};!A&A=@{2};
!b&L = b                     ---------------------------------------- DUP-NOD
t                            &B{1,2}<>@{[&B{1,2}]}
                             collapsed: [1,1] and [2,2] · Itrs: 5 (with collapse)
```

Read the result: both copies `x₀` and `x₁` became the *same* superposition `&B{1,2}`. Collapsing gives two worlds, `[1,1]` and `[2,2]`: in each world, both copies agree. Copying a superposition copies the *choice*, not one side.

APP-SUP — application meets superposition. γ–δ commutation: applying a pair applies both halves, and the argument is duplicated.

```
(&L{f,g})(a)                 @main = (&A{λx.x, λy.10})(7)
------------ APP-SUP          ---------------------------------------- APP-SUP
!x&L = a                     &A{@{λa.a}(A₀),@{λa.10}(A₁)};!A&A=@{7};
&L{f(x₀), g(x₁)}             ---------------------------------------- APP-LAM
                             &A{@{A₀},@{λa.10}(A₁)};!A&A=@{7};
                             ---------------------------------------- DUP-NOD
                             &A{7,@{λa.10}(7)}
                             ---------------------------------------- APP-LAM
                             &A{7,@{10}}                collapsed: 7 and 10 · Itrs: 4
```

`DUP-NOD` is HVM4's rule for duplicating a data node `T{a,b,…}` one layer at a time: both sides get a copy of the node, and each child is put behind a new duplication with the same label (sheet `dup_nod`, Appendix B). A number has no children, so its copy finishes in one interaction.

DUP-LAM — duplication meets lambda. γ–δ commutation the other way: copying a function.

```
!f&L = λx.g; t                @main = !f&A = λx.x; [f₀, f₁]
-------------- DUP-LAM        ------------------------------------- DUP-LAM
f₀ ← λx0.G₀                   λa.A₀<>@{[λb.A₁]};!A&A=@{&A{a,b}};
f₁ ← λx1.G₁                   ------------------------------------- DUP-SUP
x  ← &L{x0,x1}                λa.a<>@{[λb.b]}            Itrs: 2
!G&L = g
t
```

This is the deep rule. Each copy gets a fresh variable (`x0` , `x1`), and the *original* variable `x` , still sitting inside the body `g` , is replaced by the superposition `&L{x0,x1}` of the two new ones. The body is shared behind a new duplication `!G&L = g` . When that duplication later meets the superposition — same label L — it projects: copy 0 sees `x0` , copy 1 sees `x1` . **The variable temporarily lives outside its binder.** That is why IC variables must be *global* (departure 2): the λ-calculus' lexical scoping cannot express this intermediate state; the IC can.

Erasure. Erasing a value applies the ε rules: `&{}` applied to anything is `&{}` ; an erased lambda erases its body; an erased superposition erases both halves. Two observations at HEAD:

```
@main = (λx.7)(&A{1,2})       → 7      Itrs: 1   (the argument is dropped by β; nothing forces it)
@main = !x&A = &{}; x₀        → (no result printed)  Itrs: 1   (the only branch was erased)
```

5.4 The rule summary

Pair	Rule	Kind	Effect
@ meets λ	APP-LAM	γγ annihilation	β: substitute by wiring
DUP_L meets SUP_L	DUP-SUP	δδ, same label	projection
DUP_L meets SUP_R (L≠R)	DUP-SUP	δδ, different labels	commute: copy the choice
@ meets SUP	APP-SUP	γδ commutation	apply both halves; duplicate the argument
DUP meets λ	DUP-LAM	γδ commutation	copy the function; superpose its variable
anything meets ε	ERA-…	erasure	delete one cell, propagate

Counts measured at HEAD for these one-liners: β 1; same-label projection 1; different-label copy of a pair of numbers 5 (with collapse); APP-SUP with numeric argument 4; copying the identity 2; copying a two-argument function and calling both copies 11 (`!f&A = λx.λy.[x,y]; [f₀(1,2), f₁(3,4)]` → `[[1,2],[3,4]]`).

5.5 Arithmetic across superpositions: the label decides the shape

```
@main = &A{1,2} + &A{10,20}     → 11, 22                Itrs: 4
@main = &A{1,2} + &B{10,20}     → 11, 21, 12, 22        Itrs: 10
```

Same label: the two superpositions are *the same choice*, so `+` pairs them up — 2 worlds. Different labels: independent choices — the **cross product**, 4 worlds, and more than twice the work. Labels are how a program says which alternatives are correlated.

5.6 The clone of a clone, measured

Church numeral 2 with label C, and an identical one with label K:

```
@c2 = λs. !S&C=s; λx. S₀(S₁(x))
@k2 = λs. !S&K=s; λx. S₀(S₁(x))
```

```
@main = @c2(@c2)     →  λa.λb.a                Itrs: 9
@main = @c2(@k2)     →  λa.λb.a(a(a(a(b))))    Itrs: 20    (= 4, i.e. 2²)
```

`2 2` should be $2^2 = 4$. With **different** labels the inner and outer duplications commute and the answer is 4. With the **same** label, the outer numeral's copier meets the inner numeral's fan and *annihilates* — exactly the §4.6 failure — and the result `λa.λb.a` is not a Church numeral at all. Nothing crashed: the program is a valid IC term with a different meaning than the λ-term you intended. This is the single most important thing to understand about labels:

> **A label is semantics, not decoration.** Equal labels mean "these fans are two halves of one duplication." If that is false, the program computes something else — silently.

HVM4's auto-dup sugar `λx&. …` generates a **fresh** label per use site, which is usually what you want. You can verify that fresh labels are distinct from user labels:

```
@main = (λx&. [x₀, x₁])(&A{1,2})     →  [1,1], [2,2]     Itrs: 6
```

The fresh label differs from A, so the duplication *copies the choice* (both components agree in each world) instead of projecting (which would have given `[1,2]`).

5.7 Two dialects, one calculus

The IC reference (`VictorTaelin/Interaction-Calculus`, binary `ic`) and HVM4 implement the same calculus with different spelling. The safe rule: **read each dialect's own syntax document before writing** (*L.4*, *L.7* for HVM4; *H.33* for the IC reference). One verified IC-reference run:

```
$ echo '(λx.x λy.y)' > p.ic && ic run p.ic
λ$a.$a
WORK: 1 interactions
```

(The IC reference writes application as `(f a)` with juxtaposition; HVM4 writes `f(a)`, and HVM4's README examples in the old style `((@add 1) 2)` **do not parse at HEAD**: a stale upstream README, logged as finding R-07 in the compendium.)

> **Trap: the asterisk.** In the IC reference, `*` means erasure. In HVM4, `*` is the **wildcard `ANY`** (tag 40: "a wildcard that duplicates itself and equals anything"); erasure is `&{}`. `@main = [*, 1]` at HEAD prints `[*,1]`, it does not erase anything.

What to remember from Part 5

- IC = λ + **labeled** duplications and superpositions; variables affine and **global**.
- Four rules: **APP-LAM** (β), **DUP-SUP** (same label: project; different: copy the choice), **APP-SUP** (distribute), **DUP-LAM** (copy a function, superpose its variable), plus erasure.
- A SUP is a DUP seen from the other side: one cell kind, two readings.
- Same label = correlated alternatives (pairwise); different labels = independent (cross product).
- Labels are semantics: `@c2(@c2)` is not `4`. Fresh labels (`λx&.`) are the safe default.

Exercises — Part 5

E5.1 ★ Reduce `!x&A = &A{3, 4}; x₀ + x₁` by hand. Which rule fires first, and what is the result?

▾ SOLUTION

1. The duplication meets a superposition with the same label: **DUP-SUP, projection**: $x_0 \leftarrow 3$, $x_1 \leftarrow 4$.
2. Then `3 + 4` → 7 (an arithmetic rule, OP2-NUM-NUM).
3. Result **7**, 2 interactions on HVM4.

E5.2 ★ Same with `!x&A = &B{3, 4}; x₀ + x₁`. How many worlds after collapse, and which values?

▾ SOLUTION

1. Different labels → commute: $x_0 \leftarrow$ &B{3_0,4_0}, $x_1 \leftarrow$ &B{3_1,4_1}, and the numbers are duplicated (DUP-NOD).
2. `&B{3,4} + &B{3,4}`: both operands carry label B, so they pair up: `&B{3+3, 4+4}`.
3. **2 worlds: 6 and 8.** (Not 4 worlds: both copies make the same B-choice.)

E5.3 ★★ Explain why `&A{1,2} + &A{10,20}` gives 2 results but `&A{1,2} + &B{10,20}` gives 4, using the APP-SUP/DUP-SUP picture of `+`.

▾ SOLUTION

1. `+` is strict in both arguments; meeting a superposition it distributes (an OP2-SUP rule, the arithmetic analogue of APP-SUP), duplicating the other operand under that superposition's label.
2. Same label: the duplication of `&A{10,20}` under label A meets a superposition with label A → projection: branch 0 gets 10, branch 1 gets 20 → `&A{11, 22}`.
3. Different labels: the duplication under A meets `&B{…}` → commute: each A-branch gets a full `&B{10,20}` → 4 combinations.

E5.4 ★★ In DUP-LAM, why must the copied variable `x` be replaced by a *superposition* `&L{x0,x1}`, and why must its label be the *same* L as the duplication?

▾ SOLUTION

1. The body g is shared by both copies (behind `!G&L = g`); inside it, `x` must mean "x0 in copy 0, x1 in copy 1". One position holding two values that are told apart by which copy reads it: a superposition.
2. When `!G&L = g` later copies the body and reaches that superposition, it must **project** (copy 0 takes x0), not commute. Projection happens only for equal labels: hence label L.

E5.5 ★★ Predict the collapsed results and the number of worlds of `@main = (λx&. x₀ + x₁)(&A{1, 2})`. Then explain what changes if you write `!x&A = &A{1,2}; x₀ + x₁`.

▾ SOLUTION

1. `λx&.` uses a fresh label F ≠ A. Duplicating `&A{1,2}` under F **commutes**: $x_0 = x_1 =$ &A{1,2} (same choice), so `x₀ + x₁ = &A{1+1, 2+2}` → **2 worlds: 2 and 4**.
2. With `!x&A`, the labels match: **projection** $x_0 = 1$, $x_1 = 2$ → **one result: 3**.
3. Lesson: the same-looking program means "use the value twice" (fresh label) or "split the pair" (matching label).

E5.6 ★★★ Using §5.6, explain why `@c2(@c2)` yields `λa.λb.a` (no applications of `a` to `b` survive) rather than an error.

▾ SOLUTION

1. After β, the outer numeral duplicates its argument `s` (the inner numeral) with a C-labelled fan; the inner numeral's body contains its own C-labelled fan.
2. When the outer copier reaches the inner fan, labels match → **projection** instead of commutation: the outer copier's two outputs are wired to the inner fan's two sides, i.e. the "copies" are the inner numeral's two occurrence wires of *its* s.
3. The resulting graph is a valid IC net, but its readback is not the λ-term `4`; the structure that would apply `b` collapses into the degenerate `λa.λb.a`. No rule is missing, so no error is raised: the program means something different.

E5.7 ★ What does `@main = [*, 1]` print on HVM4 HEAD, and why is that a trap for readers of the IC reference?

▾ SOLUTION

1. It prints `[*,1]` with 0 interactions: `*` is the wildcard `ANY`, a value.
2. In the IC reference `*` is erasure, so a reader moving between dialects would expect the first element to vanish. HVM4 erases with `&{}`.

E5.8 ★★ HVM4 rejects `!c&C = &A{1,2}; (c₁ * c₁)` with `variable 'c₁' used 2 times`. Explain why, and give two correct fixes.

▾ SOLUTION

1. IC variables are affine; each side `c₀`, `c₁` of a duplication may be used at most once. Using `c₁` twice would need another duplicator.
2. Fix A: duplicate again, `!c&C = …; !d&D = c₁; (d₀ * d₁)`.
3. Fix B: let the parser insert duplicators: `!&c = …; (c * c)` (the hint in the error message: "declare variable as '&c' to allow multiple uses").

<hr>

6 · Superposition as a programming tool

6.1 Two values in one place

A superposition `&L{a,b}` is a value that is *both* a and b until something observes it. Computation proceeds on both alternatives **sharing the common work**; `-C` (collapse) at the end turns the superposed result into a list of plain answers. HVM4 writes them one per line, each followed by `#N`, the running interaction count when that world was printed (§8.6).

```
@main = (&A{1,2} + 10)          →  11, 12        Itrs: 4
@main = λx. (x + &A{1,2})       →  λa.(a + 1), λa.(a + 2)       Itrs: 1
```

The second line shows that superpositions may sit inside lambdas: collapse lifts the choice to the top of the term.

6.2 Forks: two versions of a program, sharing what they can

HEAD adds **auto-fork** sugar, `&L!{A; B}`: both branches see every variable in scope, duplicated automatically:

```
@main = (λa. &L!{(a + 1); (a * 10)})(5)     →  6, 50      Itrs: 4
```

One argument, two programs, one evaluation of everything they share. (The explicit form is the sup-fork `&L[x,y,&z]{A; B}`, which names the captured variables; §8.9.)

6.3 Enumeration

Superpositions can enumerate data as easily as numbers. All 3-bit strings, one world per string (from the HVM4 test suite):

```
@bits = λ{ #Z: #E{} ; #S: λp.!P&L=p ; &L{ #O{@bits(P₀)}, #I{@bits(P₁)} } }
@main = @bits(#S{#S{#S{#Z{}}}})
```

```
#O{#O{#O{#E{}}}}   #O{#O{#I{#E{}}}}   #O{#I{#O{#E{}}}}   #O{#I{#I{#E{}}}}
#I{#O{#O{#E{}}}}   #I{#O{#I{#E{}}}}   #I{#I{#O{#E{}}}}   #I{#I{#I{#E{}}}}
Itrs: 40
```

Notice that all three levels use the **same** label L. That is correct here because each level's superposition is consumed by a *different* constructor position; the strings come out as a full binary tree of choices.

6.4 Search by erasure

The pattern Taelin describes for program search *(H.50)*:

> "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."

The mechanical version: superpose the candidates, apply a test, and let the failing branches **erase themselves**; collapse prints the survivors.

```
@keep  = λx. λ{0: &{}; λn. x}                          // keep x when the test is non-zero
@cands = &A{&B{&C{0,1},&C{2,3}},&B{&C{4,5},&C{6,7}}}    // 0..7 as three independent bits
@main  = !&c = @cands; @keep(c, (c * c) == (c + c))
```

```
0 #74
2 #103
Itrs: 167
```

The solutions of $x^2 = 2x$ in 0..7. Two remarks keep this honest:

1. **The work is not free.** Eight candidates are still evaluated where they differ; sharing helps only with the work they have in common. 167 interactions here; Part 9 discusses when this beats a loop.
2. **Erasure shapes the result.** A branch that never reaches the output (because `&{}` replaced it) produces no line at all — count *results that reach the output*, not candidates.

6.5 Infinite superpositions and lazy search

Because superpositions are only unfolded on demand, one may be infinite. "Every natural number" is a two-line definition:

```
@if   = λ{ #T: λt.λf.t ; #F: λt.λf.f }
@add  = λ{ #Z: λb.b ; #S: λp.λb.#S{@add(p, b)}; }
@eq   = λ{ #Z: λ{ #Z: #T{}; #S: λbp. #F{} }; #S: λap.λ{ #Z: #F{}; #S: λbp. @eq(ap, bp) } }
@X    = &Z{#Z{}, #S{@X}}          // the superposition of all naturals
@two  = #S{#S{#Z{}}}
@four = #S{#S{#S{#S{#Z{}}}}}
@main = @if(@eq(@add(@X, @two), @four), @X, &{})
```

```
#S{#S{#Z{}}} #306
Itrs: 536
```

The equation `X + 2 = 4` solved by laziness: the runtime unfolded `@X` only as far as the comparison demanded, erased every failing branch, and stopped. Collapse explores worlds in priority order (§8.6), which is why an infinite space can still yield its first answer.

6.6 What superposition is not

- It is **not** a process fork: there is one term in one heap; alternatives are cells inside it.
- It is **not** free parallelism: alternatives share work where the computation is the same and pay separately where it differs.
- It does **not** make search polynomial: a superposition of 2^n candidates can still need 2^n distinct pieces of work. The win is sharing, and sharing depends on the problem.

- It is **not** in released Bend 2: Bend 2 runs on BendRT (Part 10); `sup` is an HVM-lineage feature.
- It does **not** pick the best candidate. Collapse *enumerates* every surviving world (§6.4, §8.6); "selection" only happens where a test turns failing branches into `&{}`.
- It is **not** a way to run independent tasks side by side. Two unrelated computations are already independent subterms; wrapping them in `&L{a, b}` adds APP-SUP and DUP-SUP work and creates nothing to share.

Two rule shortcuts that circulate in notes and generated explanations, and are wrong:

Shortcut	What the rule actually does
"DUP–DUP with different labels reorders work for parallelism"	it **copies** each duplicator through the other (DUP-SUP commute, §5.3): 2 cells become 4
"SUP meeting a constructor collapses to the winner"	an eliminator meeting a SUP **distributes** over it (APP-SUP, APP-MAT-SUP): both branches continue

What to remember from Part 6

- `&L{a,b}` = two values in one place; `-C` lists the worlds.
- Same label: correlated (pairwise); different labels: independent (cross product).
- **Search by erasure**: superpose candidates, apply a test whose failing branches become `&{}`, collapse.
- Infinite superpositions work because unfolding is lazy; collapse follows priorities.
- Counts are honest only if you count the work on *distinct* parts.

Exercises — Part 6

E6.1 ★ Predict the output of `@main = &A{1,2} * &B{3,4}` (values and number of worlds).

▾ SOLUTION

1. Different labels → cross product.
2. Worlds: 1·3, 1·4, 2·3, 2·4 → **3, 4, 6, 8** (4 worlds; HVM4 prints them in its queue order).

E6.2 ★ Rewrite §6.4's test to find the x in 0..7 with `x*x + 1 == 26`. What does it print?

▾ SOLUTION

1. `@main = !&c = @cands; @keep(c, ((c * c) + 1) == 26)`.
2. Only x = 5 satisfies $5^2 + 1 = 26$: **one line, `5`** (HEAD prints `5 #117`, ltrs 134).

E6.3 ★★ Why does §6.4 use three different labels A, B, C for the candidate bits, whereas §6.3 uses one label L at every level?

▾ SOLUTION

1. In §6.4, the candidate `c` is a single number built from three choices that must be **independent** (all 8 combinations). If they shared a label, a duplication of `c` would project them pairwise and the candidate space would collapse.
2. In §6.3, each level's superposition is placed into a different constructor field and never duplicated against another level's; the recursion duplicates the predecessor `p` under L, and those duplications project only against their own superposition. The labels never meet each other wrongly.
3. Rule of thumb: if two choices can meet in one duplication and must stay independent, they need different labels.

E6.4 ★★ In §6.5, what would go wrong if `@eq` returned `#F{}` for failing branches and the program printed `@eq(...)` directly instead of using `@if(…, @X, &{})`?

▾ SOLUTION

1. Nothing would be erased: every candidate world would produce a line, `#F{}` for failures.
2. Because `@X` is infinite, collapse would keep producing `#F{}` lines forever (or until the `-C=N` limit).
3. The erasure `&{}` is what turns a predicate into a **search**: failures disappear instead of being printed.

E6.5 ★★ "Superposition is a fork of the program." Give two precise reasons the sentence is misleading.

▾ SOLUTION

1. There is no second process or copied heap; the alternatives are cells inside one term, and everything not yet distinguished is shared.
2. Costs follow differences, not branches: two branches that differ only in a constant pay only for the parts that depend on it. A real fork pays for both copies of everything.

E6.6 ★★★ Design a superposition-based program that returns all pairs (x, y) with x, y ∈ {0,1,2,3} and x + y == 3, using two independent 2-bit candidates. Write it and state the expected worlds.

▾ SOLUTION

1. Candidates: `@cx = &A{&B{0,1},&B{2,3}}`, `@cy = &C{&D{0,1},&D{2,3}}` (four distinct labels: x and y independent, bits independent).
2. Test and keep: `@main = !&x = @cx; !&y = @cy; @keep([x, y], (x + y) == 3)` with `@keep` from §6.4.
3. Expected survivors: **[0,3], [1,2], [2,1], [3,0]** — four lines (in the collapser's priority order).
4. Run it; if a pair appears twice or is missing, look for a shared label.

<hr>

7 · Algorithms and their counts

Every count in this part was produced by HVM4 HEAD on 2026-11-06 (`-s -C`). Counts are properties of the program's graph (§2.2), so they reproduce exactly; heap sizes and times do not, and are never quoted.

7.1 Booleans

```
@ctru = λt. λf. t              // True  = take the first
@cfal = λt. λf. f              // False = take the second
@cnot = λb. λt. λf. b(f, t)    // NOT   = swap the arguments
@main = @cnot(@cnot(@ctru))
```

```
λa.λb.a      Itrs: 6
```

Two NOTs, each one β to receive `b` and the applications inside: 6 interactions, no duplication (every variable is used once).

7.2 Church numerals and addition

```
@c1   = 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))
@cadd = λa. λb. λs. λz. !S&B=s; a(S₀, b(S₁, z))
@cmul = λa. λb. λs. λz. a(b(s), z)
```

A Church numeral *n* is a function that applies `s` *n* times; as a net it is a fan of *n* − *1* duplicators on `s` (§4.2). `c4` is built as "duplicate the doubling function", so it has two duplications, both labelled C.

Program	Result	Itrs
`@cadd(@c1, @c2)`	`λa.λb.a(a(a(b)))` = 3	8
`@cadd(@c4, @c4)`	`λa.λb.a(a(a(a(a(a(a(a(b)))))))))` = 8	25
`@cmul(@c2, @k2)`	`λa.λb.a(a(a(a(b))))` = 4	14
`@cmul(@c2, @c2)`	`λa.λb.a(a(a(a(b))))` = 4	14
`@c2(@k2)` (exponentiation 2^2)	`λa.λb.a(a(a(a(b))))` = 4	20
`@c2(@c2)`	`λa.λb.a` (degenerate)	9

The last four rows are the most instructive in the book:

- `cmul(c2, c2)` is correct **even with equal labels.** Multiplication *composes* `b(s)` into `a` without making `a`'s fan copy `b`'s fan: no C-fan ever meets another C-fan.

- `c2(c2)` (exponentiation) is wrong with equal labels because the outer numeral's fan **copies** the inner numeral, and its copier meets the inner fan (§5.6).

So the rule is not "never reuse a label". It is: **a fan must not meet a same-labelled fan unless they are two halves of one duplication.**

7.3 Peano naturals: two numeral forms that do not mix

HVM4 offers two ways to write naturals, matched by two different pattern families:

Form	Example	Matched by	Prints as
n-literals	`3n`	`0n: / 1n+:`	`3n`
constructor chains	`#S{#S{#S{#Z{}}}}`	`#Z: / #S:`	`#S{#S{#S{#Z{}}}}`

Feeding a `3n` literal to a function written with `#Z: / #S:` patterns stops the run with `RUNTIME_ERROR: cannot apply a number`. The classics, written with constructor chains:

```
@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)); }
@fib = λ{ #Z:#Z{}; λ{#S:λ{#Z:#S{#Z{}}; λ{#S:λn.!F&=@fib; !N&=n; @add(F₀(N₀), F₁(#S{N₁})); &{}}}; &{}}}
@sum = λ{ #Z:#Z{}; λ{#S:λp.!P&S=p;#S{@add(P₀, @sum(P₁))}; &{}}}
```

Program	Result	Itrs
`@fac(#S{#S{#S{#Z{}}}})`	`#S` × 6 → 6	109
`@fib(#S{#S{#S{#S{#S{#Z{}}}}}})`	`#S` × 5 → 5	274
`@sum(#S{#S{#S{#S{#Z{}}}}})`	10	55

(The test suite's own factorial builds its 3 as `@add(@two, @one)` first and reports 119 = 109 + the 10 interactions that build the argument. Counts measure the graph you hand the machine, including how its inputs were built.)

7.4 Lists

```
@map = λ&f. λ{ []: []; <>: λh.λt. (f(h) <> @map(f, t)) }
@main = @map(λx.(x*2), [1,2,3])
```

```
[2,4,6]      Itrs: 35
```

`λ&f.` asks the parser to insert duplicators for every use of `f` (two here: `f(h)` and the recursive `@map(f, t)`). Each recursion step copies the function **one layer at a time** (DUP-LAM), applies one copy, and passes the other on.

7.5 The showpiece: sharing inside a lambda

```
@f = λx.λy. !z&=x; [(z₀ + z₁), y]
@main = !F& = @f(2); [F₀(10), F₁(20)]
```

```
[[4,10],[4,20]]      Itrs: 11
```

The full `-D` trace at HEAD (abridged to rule names, every line is one interaction):

```
 1  APP-LAM       @f receives 2; the partially applied function is now λy. … (2 + 2) …
 2  DUP-LAM       the copier F reaches λy: two lambda copies, y becomes &F{a,b}
 3  APP-LAM       copy 0 receives 10
 4  DUP-NOD       the shared body is duplicated one node (the list) at a time …
 5  DUP-NOD       … the copy reaches the sum and must wait: (2 + 2) is a redex
 6  OP2-NUM-NUM   (2 + 2) → 4   ← computed ONCE, in the shared position
 7  DUP-NOD       the number 4 is duplicated (one interaction for a number)
 8  DUP-NOD       the copier continues into the list
 9  DUP-SUP       &F{10, a} projected: copy 0 gets 10
10  DUP-NOD       the tail [] is duplicated
11  APP-LAM       copy 1 receives 20
```

Line 6 is the point of the whole book: the copier stopped at a redex (§4.5), the redex fired once, and both copies received its result. Call-by-need would have evaluated `2 + 2` once **per call** (§1.2).

> **Honesty about this example.** Calling the function twice instead of sharing one partial application (`@main = [@f(2,10), @f(2,20)]`) takes **8** interactions, fewer than 11, and its `-D` trace contains `OP2-NUM-NUM` **twice**. On a tiny example, the bookkeeping of sharing costs more than the work it saves. The win is *asymptotic*: replace `2 + 2` with an expensive computation and the shared version does it once while the unshared one does it twice. Optimal evaluation is a claim about families of programs, not about every program.

7.6 Where sharing does not help: naive Fibonacci

The machine-integer Fibonacci in the test suite (`fib_small`, a `λ{0:…;1:…;λ&n.…}` switch) computes `fib(34) = 5702887` in **79,522,600 interactions**, about 5 seconds on this machine. That is exponential, exactly as in any language: the two recursive calls `fib(n-1)` and `fib(n-2)` are *distinct* computations, not two uses of one shared value. There is nothing for a duplicator to share. Sharing helps when a program *creates* a shared value; it never turns independent work into shared work.

7.7 Runtime fusion: applying a function 2^{256} times

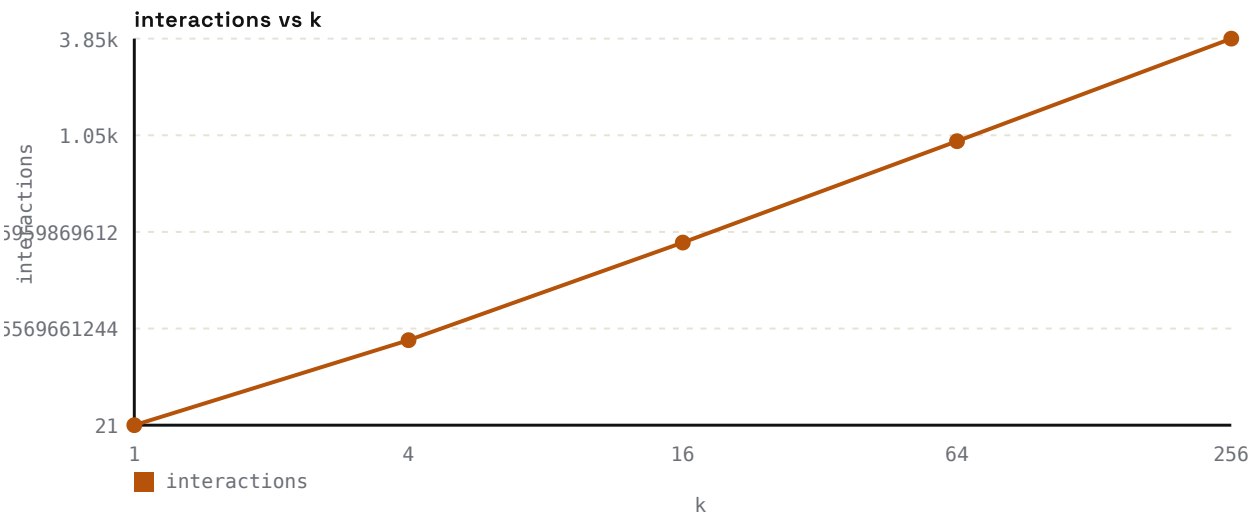
The showpiece (§7.5) shares a redex between two copies. The same mechanism does something more surprising when a function is **composed with itself**: the composition is reduced once, under the λ, and both later uses get the reduced result. Taelin's essay on the abstract algorithm's speed *(H.48)* calls this runtime fusion and uses `not` as the example: `not ∘ not` is the identity, so composing by repeated squaring reaches `not^(2^k)` in k compositions, provided the evaluator actually normalizes each composition before copying it.

The program (Church-style booleans, so `not` is a λ-term the runtime can fuse):

```
@not  = λb.λt.λf.b(f,t)
@T    = λt.λf.t
@pow  = λ{0: λf.f; λ&p. λf. !F&A=f; @pow(p - 1, λx.F₀(F₁(x)))}
@main = @pow(16, @not)(@T)
```

`@pow(k, f)` returns `f` composed with itself 2^k times: each level duplicates the current function and composes the two copies. (A numeric switch passes the number itself to the default branch, hence `p - 1`.) Measured on HEAD `6defdfc` with `-s -C`; every run returns `λa.λb.a`, which is `@T`, correct because 2^k is even:

k	applications of `not`	interactions
1	2	21
4	16	66
16	65,536	246
64	1.8×10^{19}	966
256	1.2×10^{77}	3,846



Plotted from the table above.

The count is exactly **15k + 6**: linear in k, so logarithmic in the number of applications. For comparison, applying `not` one call at a time (`@rep = λ{0: λx.x; λ&n. λx. @rep(n - 1, @not(x))}`) costs exactly **7n + 2**: 458,754 interactions for n = 65,536, against 246 for the fused version of the same 65,536 applications.

Why it works, in the terms of this book:

1. `!F&A=f; λx.F₀(F₁(x))` duplicates `f` once and composes the copies. When the composition is later duplicated at the next level, the copier meets a redex inside the λ and **evaluates it first** (§4.5, §8B.5).
2. That redex is `not(not(x))`, which reduces to a λ-term of constant size (`λt.λf.x(t,f)`). So every level copies a small normal form, not a growing chain of `not`s.

3. A strict or call-by-need evaluator cannot do step 1: the redex is under a λ that has not been applied yet (§1.2).

Two honest limits. First, fusion depends on the composed function having a **small normal form**; `inc` on unary naturals does not shrink when composed with itself, and repeated squaring gains nothing there. Second, this program uses one static label `A` at every level and gives the right answer for every k tested (1–256); the principled form uses one label per level, a dynamic label `!F&(p)=f` (§8.9), and costs exactly 19k + 6 (4,870 interactions at k = 256). Labels decide correctness (§5.6); they must be argued, not assumed.

What to remember from Part 7

- Counts are structural and reproducible; heap sizes and times are not evidence.
- Church numerals are fans; a fan's label decides copy vs projection. `cmul(c2,c2)` is safe, `c2(c2)` is not.
- Peano naturals have two surface forms with separate pattern families: do not mix them.
- The showpiece shows the mechanism (one `OP2-NUM-NUM` for two copies) and its overhead (11 vs 8).
- Sharing helps only where the program shares; naive Fibonacci is exponential everywhere.
- **Runtime fusion:** self-composition by squaring reaches `not^(2^256)` in 3,846 interactions, because each composition is normalized under its λ before it is copied. It works only when the composition has a small normal form.

Exercises — Part 7

E7.1 ★ Run `@main = (λ&x. x(x))(λy.y)` and explain each of its interactions (compare E4.4).

▾ **SOLUTION**

1. Output `λa.a`, `Itrs: 4`.
2. APP-LAM (the argument enters), DUP-LAM (the identity is copied), DUP-SUP (the copied variable is projected), APP-LAM (`id id`).
3. Same count as the hand net of E4.4: the machine and the model agree.

E7.2 ★ Predict `@cadd(@c2, @c2)`: result and a rough count; then run it.

▾ **SOLUTION**

1. 2 + 2 = 4 applications: `λa.λb.a(a(a(a(b))))`.
2. More than `cadd(c1,c2)` (8), less than `cadd(c4,c4)` (25). HEAD reports **9** interactions.

E7.3 ★★ Why does `@cmul(@c2, @c2)` give 4 while `@c2(@c2)` does not? Point to the exact place where fans meet.

▾ SOLUTION

1. `cmul a b s z = a(b(s), z)` : `b(s)` builds "apply s twice" (the inner C-fan duplicates `s`), and `a` receives that function and duplicates it with *its* C-fan.
2. The outer C-fan copies `b(s) = λx0. S0₀(S0₁(x0))`, whose free variables `S0₀`, `S0₁` are the *sides* of the inner fan, not the fan itself. Copying passes over variables, so the two fans never meet principal to principal.
3. In `c2(c2)` the outer fan duplicates the inner *numeral* `λs. !S0&C=s; …`, and copying the λ brings the outer copier to the inner fan's principal: same label → projection → wrong result.

E7.4 ★★ (Debug) A student writes `@add = λ{ #Z: λb.b ; #S: λp.λb.#S{@add(p, b)}; }` and `@main = @add(1n, 2n)`. What happens and how do you fix it?

▾ SOLUTION

1. `RUNTIME_ERROR: cannot apply a number` : `1n` is an n-literal, but the function matches constructor tags `#Z` / `#S`.
2. Fix A: write the inputs as chains, `@add(#S{#Z{}}, #S{#S{#Z{}}})`.
3. Fix B: rewrite the patterns with the n-literal family, `λ{ 0n: λb.b ; 1n+: λp.λb. … }`.

E7.5 ★★ Why does the test suite's factorial of 3 report 119 interactions when §7.3 reports 109?

▾ SOLUTION

1. The suite computes its argument as `@add(@two, @one)` inside `@main`; those 10 interactions are part of the run.
2. Interaction counts measure everything the machine does from `@main`, including building inputs. Compare programs only when their inputs are built the same way.

E7.6 ★★ In §7.5, which line of the trace would be missing if `@f` were written `λx.λy. [(x + x), y]` with `λ&x`? Would the sum still be shared?

▾ SOLUTION

1. With `λ&x`, the parser inserts the duplication of `x` inside the body, so the structure is the same: `(x₀ + x₁)` sits inside the λy body, behind the copier.
2. The copier still meets the `+` redex after `x` is known and waits; `OP2-NUM-NUM` fires once.
3. Yes, still shared: auto-dup inserts the same duplicator a manual `!z&=x` would. The count may differ by the shape of the inserted fan.

E7.7 ★★★ Construct a variant of §7.5 where the unshared version costs **more** interactions than the shared one, and state your prediction before running.

▾ SOLUTION

1. Make the shared work expensive: put a call that takes many interactions in the shared position, e.g. `@f = λx.λy. !z&=x; [@sum(z₀ … ), y]` with `@sum` over a chain of length 20, versus writing the call twice.
2. Prediction: shared ≈ cost(call) + overhead (≈ 10), unshared ≈ 2 × cost(call). For a call of ~250 interactions: ~260 vs ~500.
3. Run both; the crossover point (where overhead equals the saved call) is the honest summary of "sharing pays".

E7.9 ★★ In §7.7, why must `not` be a λ-term (Church boolean) rather than a pattern match on constructors `#T{}` / `#F{}` for fusion to happen?

▾ SOLUTION

1. Fusion needs `not ∘ not` to reduce **before** it receives its argument, under the λ of the composition.
2. With Church booleans, `not(not(x))` is β-reducible with `x` unknown: it normalizes to `λt.λf.x(t,f)`, a constant-size term.
3. A constructor match on `x` cannot fire until `x` is a constructor; under the λ, `x` is a variable, so the match is stuck and each level would copy a growing chain of pending matches.

E7.10 ★★★ Predict the interaction count of `@pow(32, @not)(@T)` from the table, and say what result it prints.

▾ SOLUTION

1. The measured counts fit 15k + 6 exactly (21, 66, 246, 966, 3,846 for k = 1, 4, 16, 64, 256).
2. k = 32 → 15 · 32 + 6 = **486** interactions.
3. 2^{32} is even, so the result is `@T = λa.λb.a`. (Measured: 486.)

E7.8 ★ Why is `fib(34)` exponential on HVM4 too?

▾ SOLUTION

The two recursive calls are different computations (different arguments); no value is used twice, so no duplicator ever reaches a redex that another copy also needs. Optimal reduction never shares work that is not the same work.

<hr>

8 · Inside HVM4 (HEAD `6defdfc`)

Parts 2–7 used nets as the *model*. This part opens the *machine*. Everything below refers to `src/hvm.c` at commit `6defdfc` (2026-05-29), the only source file of the runtime; line numbers are for that commit.

8.1 One file, eleven sections

At HEAD, HVM4 is a single translation unit: `clang -O2 -o src/hvm src/hvm.c`. The repository's `AGENTS.md` *(L.2)* lists its sections:

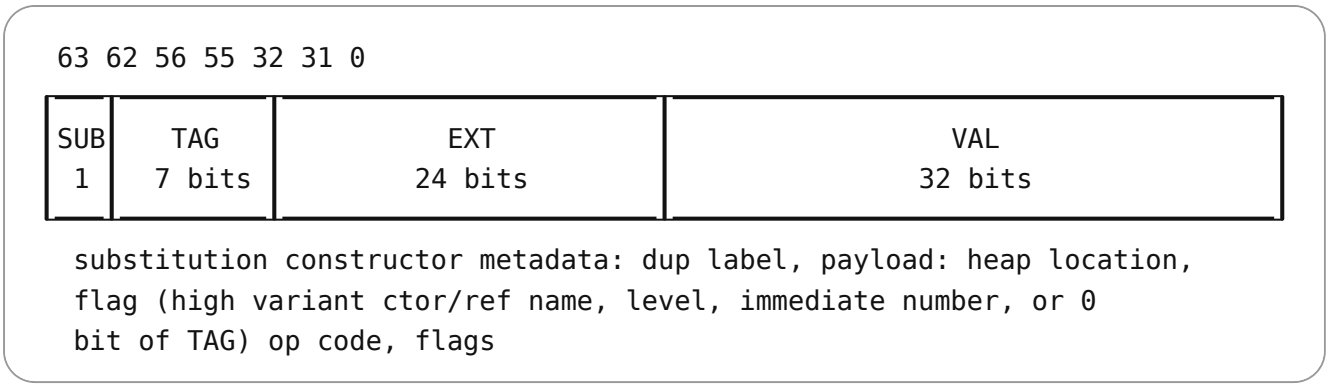
> "`src/hvm.c` is the only C runtime source. It is organized as broad sections, inspired by Bend's `Core.ts`"

Section	Contents
Types	scalar aliases, term tags, parser/runtime structs, globals
Term	term packing, tag/ext/val helpers, constructors, clone helpers, OP2
Heap	allocation, heap access, substitutions
Nick, System, Table, Print	names, file helpers, intern table, pretty-printing
Runtime	process setup, program preparation, entry lookup, `@main` evaluation
Parse	lexer, bindings, syntax sugar, includes, definitions
WNF	the stack evaluator and the interaction rules
Data	small 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 the entry point

The `--help` text calls it *"A sequential Interaction Calculus runtime."* There is no thread pool, no AOT compiler and no FFI at HEAD (§8.8).

8.2 The term word

Every term, static or dynamic, is one 64-bit word *(L.6, `src/hvm.c:123–133`)*:




```
#define TAG_BITS 8
#define EXT_BITS 24
#define VAL_BITS 32
#define TAG_SHIFT 56
#define EXT_SHIFT 32
#define TAG_SUB_MASK (1u << (TAG_BITS - 1))   // the high bit of the tag byte
#define TAG_MASK     (TAG_SUB_MASK - 1u)
```

The label of a duplication or superposition lives in **EXT** (24 bits): labels are not names at run time, they are numbers in the word.

8.3 The 45 tags

```
 0 APP   1 VAR   2 LAM   3 DP0   4 DP1   5 SUP   6 DUP   7 ALO      ← "hot tags first (0-7)"
 8 REF   9 NAM  10 DRY  11 ERA  12 MAT  13–29 C00…C16 (constructors by arity)
30 NUM  31 SWI  32 USE  33 OP2  34 DSU  35 DDU  36 EQL  37 AND  38 OR
39 UNS  40 ANY  41 INC  42 BJV  43 BJ0  44 BJ1
```

Grouped by role:

Family	Tags	Role
λ-calculus	APP, VAR, LAM	application, linked variable, lambda
sharing	SUP, DUP, DP0, DP1	superposition; duplication; the two *sides* of a duplication (x_0 , x_1)
dynamic labels	DSU, DDU	superposition/duplication whose label is computed at run time (strict on the label)
data	C00–C16, NUM, NAM, ERA, ANY	constructors by arity, unboxed u32, names, erasure, wildcard
elimination	MAT, SWI, USE	constructor match, numeric switch, unboxing
operators	OP2, EQL, AND, OR	arithmetic, structural equality, short-circuit logic
plumbing	ALO, REF, DRY, UNS, INC	lazy book allocation, book reference, "dry" (stuck) application, unscoped binders, collapse priority
quoted variables	BJV, BJ0, BJ1	de Bruijn-level variables in static (book) terms

8.4 Book and heap

HVM4 keeps two kinds of terms *(L.6)*:

- **Static (book) terms** — the definitions `@name = …`, parsed once and stored immutably, with variables as **de Bruijn levels** (BJV/BJ0/BJ1).
- **Dynamic (heap) terms** — the mutable graph being reduced, with **linked** variables: a VAR points to its lambda's body slot; DP0/DP1 point to their DUP node.

A reference `@f` is not copied eagerly. **ALO** ("allocate") is a lazy wrapper around a book term: it is expanded into heap nodes one layer at a time, only when the evaluator looks at it — the same incremental principle as copying. That is the `@{…}` notation in `-D` traces: `@{λa.a}` is a book term that has not yet been allocated. The `ALO-*` interaction sheets (`alo_lam`, `alo_var`, `alo_dup`, …) describe each expansion step.

Substitution. When β connects a variable to its argument, HVM4 writes the argument into the variable's slot and sets the **SUB** bit, so the next reader of that slot knows it holds a substitution, not a binder. That is the term-machine encoding of "wire the variable to the argument" (§4.1).

8.5 The evaluator: WNF, SNF, CNF

- **WNF** (weak normal form) — a stack machine walks to the head of the term. When the head is an eliminator (APP, DUP side, MAT, OP2, …) whose scrutinee is in WNF, the pair of their tags selects an interaction function: `wnf_app_lam`, `wnf_dup_sup`, `wnf_app_sup`, `wnf_dup_lam`, `wnf_op2_num_num`, …. HEAD has **69 `wnf_*` functions** implementing the 61 documented sheets plus helpers.
- **SNF** (strong normal form) — reduce to WNF, then recurse into sub-terms (the full readback of a normal form).
- **CNF** (collapsed normal form) — reduce to WNF, then lift the first SUP to the top, so the term becomes `&L{left, right}` and the two worlds can be enumerated *(L.8)*.

Every interaction increments one global counter through a macro:

```
static u64 ITRS = 0;                          // src/hvm.c:171
#define ITRS_INC(name) ...  ITRS++; ...       // src/hvm.c:175–181
```

`ITRS_INC("APP-LAM")` and its 59 siblings are the 60 call sites; the `name` is what `-D` prints above each step.

8.6 Collapse, and the number after each answer

With `-C`, `eval_collapse` *(L.8)* enumerates worlds with a sequential priority queue:

> "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)`."

The number printed after each answer is **not** that key. It is the running interaction counter:

```
printf(" \033[2m#%llu\033[0m", (unsigned long long)wnf_itrs_total());   // src/hvm.c:6193
```

So `11 #3 / 12 #4` means: the first world was complete after 3 interactions, the second after 4. In §6.5, the answer appeared at `#306` and the run ended at `Itrs: 536` — the collapser kept exploring (and erasing) the remaining branches after printing it. `INC` is the programmer's lever on the order: wrapping a branch in `↑` lowers its priority (the `INC-*` sheets). The newest commit (`6defdfc`, "general filter-credit scheduler for -C search ordering") refines that ordering for search programs.

8.7 The CLI at HEAD

```
-s, --stats             Show statistics after evaluation
-S, --silent            Suppress result output
-C, --collapse[=N]      Collapse superpositions (limit N)
-D, --step-by-step      Trace each reduction step
-d, --debug             Enable debug mode
-v, --version           Print version
```

`-C10` limits collapse to 10 lines (the README's example). `-D` prints the whole term after each interaction, with the rule name as a separator line (every trace in this book).

8.8 What changed since March 2026

The first edition described commit `2d7f28e` (2026-03-20). Seventeen commits later, HEAD differs substantially *(L.1, git log)*:

Commit	Change
`1369e3b` … `e2d1f59`	a CUDA runtime experiment (`cuda/`), reported 4 → 23.6 G interactions/s on the authors' benchmarks
`224670d`	a semi-space copying GC for bounded memory on tail recursion
`14a3eea`, `4b4676d`, `2e57823`	new syntax: sup-fork `&L[x,y,&z]{A;B}`, auto-fork `&L!{A;B}`, dup destructuring `! &L{x,y} = v; body`
`4bc2336`	**"Refactor HVM runtime into pure single source"**: the `clang/` tree (AOT emitter, FFI loader, threads, work-stealing queues) is deleted; `src/hvm.c` replaces it
`be02279`	"Remove GC from pure runtime"
`6defdfc`	collapse: general filter-credit scheduler

The CUDA directory does not exist at HEAD. When you read older material that mentions `--to-c`, `-T` threads or FFI, it describes the March runtime, which survives only in git history.

8.9 New syntax at HEAD, tested

```
@main = ! &L{x,y} = 7; [x, y]              → [7,7]          Itrs: 1
@main = ! &L{x,y} = &L{1,2}; [x, y]        → [1,2]          Itrs: 1
@main = ! &A{x,y} = &B{1,2}; [x, y]        → [1,1], [2,2]   Itrs: 5
@main = (λa. &L!{(a + 1); (a * 10)})(5)    → 6, 50          Itrs: 4
```

Dup destructuring names the two sides as plain variables (`x`, `y` instead of v_0, v_1); the rules are unchanged.

8.10 The 61 rule sheets

`docs/hvm/interactions/` has one file per interaction, each a sequent-style rule (*L.9–L.69*). The full list with one line each is Appendix B. Two findings at HEAD:

- `eql_nam.md` is **empty** (20 bytes, a title and an empty code block), although `wnf_eql_nam` exists in the C file. Read the code for that rule.
- The sheets describe the term machine. The four from Part 5 are `app_lam`, `dup_sup`, `app_sup`, `dup_lam`; `dup_nod` copies one data node and re-duplicates its children under the same label: the data analogue of DUP-LAM, one layer per interaction.

What to remember from Part 8

- HEAD = one file, sequential; build with `clang -O2 -o src/hvm src/hvm.c`.
- Term word = SUB(1) | TAG(7) | EXT(24) | VAL(32); labels live in EXT.
- 45 tags; 61 sheets; 69 `wnf_*` functions; one counter `ITRS`, incremented by `ITRS_INC(name)`.
- Book terms are allocated lazily (ALO); variables are linked; substitution sets the SUB bit.
- `#N` after a collapsed answer = interactions so far, not a queue key.

Exercises — Part 8

E8.1 ★ Decode the term word `0x05000003_00000010`: which tag, label and value?

▾ **SOLUTION**

1. Top byte `0x05`: SUB bit 0, TAG = 5 = **SUP**.
2. Bytes 6–4: `0x000003` = EXT = **label 3**.
3. Low 32 bits `0x00000010` = VAL = heap location **16** (a SUP node: `[tm0, tm1]` stored at 16 and 17).

E8.2 ★ Where does HVM4 store the label of `&A{1,2}` at run time, and what limits the number of distinct labels?

▾ **SOLUTION**

In the EXT field of the SUP word: 24 bits, so at most $2^{24} \approx 16.7$ million distinct static labels. Dynamic labels (DSU/DDU) are computed values stored the same way.

E8.3 ★★ In `src/hvm.c`, find the function implementing DUP-LAM and the line that names it for `-D`.

▾ **SOLUTION**

1. `grep -n "fn .*wnf_dup_lam" src/hvm.c` finds the function.
2. Inside it, `ITRS_INC("DUP-LAM")` both counts and names the step; `-D` prints that name.

E8.4 ★★ A program prints `5 #117` and then `- Itrs: 134 interactions`. What happened in the last 17 interactions?

▾ SOLUTION

The answer was complete after 117 interactions. The collapser then continued with the remaining branches; each was reduced until its `@keep` test erased it (`&{}`) or it produced no further answer. Those interactions count, but print nothing.

E8.5 ★ Why does the `-D` trace show `@{λa.a}` instead of `λa.a` ?

▾ SOLUTION

`@{…}` marks a book term not yet allocated on the heap (ALO). The evaluator expands it lazily, one layer at a time, when it is needed; the result is the same term.

E8.6 ★★ Predict the rule names in `-D` for `@main = !x&A = 5; [x₀, x₁]` , then run it.

▾ SOLUTION

1. The duplicated value is a number: one **DUP-NOD** gives both sides 5.
2. Result `[5,5]` , 1 interaction.

E8.7 ★★ Old notes say ”run with `-T4` to use four threads” and ” `--to-c` compiles to C”. What happens at HEAD, and why?

▾ SOLUTION

Both flags are gone: HEAD’s CLI accepts only `-s -S -C[=N] -D -d -v -h` . The threaded runtime and the AOT emitter lived in the `clang/` tree that commit `4bc2336` removed in favour of a pure sequential single file.

E8.8 ★★★ Explain how a term machine with linked variables and SUB-flagged slots can implement ”β is one interaction, by wiring” without a port graph.

▾ SOLUTION

1. In a net, β connects the variable’s wire to the argument. In the term machine, the λ’s variable has exactly one occurrence (affinity), so ”the variable’s wire” is one heap slot.
2. APP-LAM writes the argument into that slot and marks it with SUB. Any reader arriving at the variable sees SUB and reads the argument: the wire is spliced.
3. The body becomes the application’s result in place. Constant work, no traversal of the body: the same one-interaction β as the net, in a different representation.

<hr>

8B · How HVM4 is built, and why

Part 8 described *what* is in `src/hvm.c`. This part explains *why* each piece is shaped the way it is. Most of the reasons are lessons learned the hard way in earlier runtimes, and the authors wrote them down: in Taelin's "HVM: how it works" *(H.46)*, in the HVM2 paper *(M.1)*, in HVM4's own memory and theory docs *(L.4, L.6)*, and in the git history. Every design decision below is paired with the source that motivates it and the code that implements it (`src/hvm.c` at `6defdfc`).

8B.0 The journey, in Taelin's words

When HVM was first announced (February 2022), a Hacker News reader asked why the authors of the optimal-reduction book never made it fast, and what led to HVM's memory layout. Taelin (account `LightMachine`) answered with the shortest account of how HVM was designed that exists *(M.7)*:

> "A naive implementation just stores graphs/edges on memory, but most of these edges are redundant. For example, a lambda doesn't need to point to its parent."

> "By trying to turn the graph in a tree, I've ended up with SIC"

> "Doing so was tricky, specially because 1. the missing edges complicated the transversal; 2. some upward edges can't disappear (variables); 3. DUP nodes aren't part of expressions, they just "float", which was counter intuitive to get right. Finally, I learned to appreciate the fact that global rewrite rules are better than case-trees for recursive functions, since they greatly reduce the total rewrite count."

> "1. Learn the abstract algorithm"
>
> "2. Implement it naively as a graph"
>
> "3. Optimize it by making trees whenever possible"
>
> "4. Favor rewrite equations over case-trees"
>
> "These steps lead to the design of HVM, which does fairly well in practice."

Each of the three "tricky" points is still visible in HVM4 at HEAD, and the rest of this part explains them:

Taelin's 2022 point	HVM4 at HEAD
remove redundant edges; a λ needs no parent pointer	nodes are headerless words, pointers go one way (§8B.4)
variables are the upward edges that cannot disappear	a VAR points at its binder's slot; β writes there with the SUB bit (§8B.4)

Taelin's 2022 point	HVM4 at HEAD
DUP nodes "float", outside the expression tree	a DUP node is a shared slot with no parent, read by DP0/DP1 (§8B.5)
global rewrite equations over case trees	definitions live in an immutable book and are instantiated lazily (§8B.6)

8B.1 The problem: clones

Taelin frames the whole design around one word *(H.46)*:

> "By clones, I mean when a value is copied, shared, replicated, or whatever else you call it."

Strict evaluation wastes work on unused arguments; lazy evaluation wastes work by cloning unevaluated expressions. Every mainstream language answers with **references**: Haskell's thunks, Rust's borrows, mutexes in parallel code. His diagnosis:

> "References. **They** ruin everything."

> "They're the reason Haskell isn't optimal, since thunks can't share computations that have free variables (i.e., any expression inside lambdas)."

and his answer, the design principle of HVM1 through HVM4:

> "Clones aren't bad. They just need to relax."

> "**HVM's runtime has no references. Instead, it features a `.clone()` primitive that has zero cost, until the cloned value needs to be read."

That primitive is the DUP of Part 5. Everything else in the machine exists to make it cheap.

8B.2 Lesson 1 (HVM2): one kind of duplicator is not enough

HVM2 was a faithful, massively parallel implementation of interaction combinators with **one** duplicator kind. Its paper *(M.1, §13.1)* records what that cost:

> "A higher-order lambda that clones its variable can not be cloned."

Church 2^2 (`(λf.λx.(f (f x)) λf.λx.(f (f x)))`) "can not be soundly reduced by HVM2": exactly the clone-of-a-clone failure of §4.6. The paper lists two fixes. More duplicator kinds, or Lamping's bookkeeping nodes, and explains why the second was rejected:

> "Adding bookkeeping to HVM should be easy. Sadly, this has the consequence of bringing a constant-time overhead, decreasing performance by about 10x."

Design consequence in HVM4: labels. Every DUP and SUP carries a label in the EXT field of its word (24 bits, §8.2), so there are 2^{24} duplicator kinds instead of one, and DUP-SUP compares two integers to decide between projection and commutation:

```
fn Term wnf_dup_sup(u32 lab, u64 loc, u8 side, Term sup) {   // src/hvm.c:3923
  ITRS_INC("DUP-SUP");
  u64 sup_loc = term_val(sup);
  u32 sup_lab = term_ext(sup);
  if (lab == sup_lab) {                          // same label: projection
    Term tm0 = heap_read(sup_loc + 0);
    Term tm1 = heap_read(sup_loc + 1);
    return heap_subst_cop(side, loc, tm0, tm1);
  } else {                                       // different labels: commute
    u64 base = heap_alloc(4);
    u64 at   = base;
    Copy A  = term_clone_at(sup_loc + 0, lab);
    Copy B  = term_clone_at(sup_loc + 1, lab);
    Term s0 = term_new_sup_at(at + 0, sup_lab, A.k0, B.k0);
    Term s1 = term_new_sup_at(at + 2, sup_lab, A.k1, B.k1);
    return heap_subst_cop(side, loc, s0, s1);
  }
}
```

That `if` is the whole "oracle" of HVM4: a constant-time comparison instead of Lamping's run-time bookkeeping. Its price is the incompleteness of §5.6: labels are only correct if the program (or the compiler, or the auto-dup sugar with fresh labels) assigns them correctly. The **dynamic labels** of HEAD (tags DSU/DDU, `&(lab){…}` , `!x&(lab) = …`) go one step further: the label is computed at run time, so a program can generate as many independent duplicator kinds as it needs, e.g. one per recursion depth.

8B.3 Lesson 2 (HVM2): eager evaluation cannot be lazy

HVM2 reduced every active pair it could find, in parallel. Its paper *(M.1, §13.2)*:

> "In our first implementation, HVM1, we used a lazy evaluation model. This not only ensured that no unnecessary work was done, but also allowed one to compute with infinite structures, like lists."

> "And if you do have an infinite structure, HVM2 will never halt (because the redex bag will never be empty)."

Laziness is not optional for this lineage. Collapse (§6.5's infinite `@X`), search by erasure and incremental copying all rely on reducing **only what is demanded**. The paper also names the obstacle to laziness on a parallel machine: it would require "uplinks" and "make the task scheduler way more

complex to implement efficiently, specially in the GPU version."

Design consequence in HVM4: a lazy, sequential, explicit-stack WNF machine. `wnf` (`src/hvm.c:4722`) has two phases, written as labels with `goto`:

```
__attribute__((hot)) fn Term wnf(Term term) {
  ...
  enter: {                                  // walk down to the head
    switch (term_tag(next)) {
      case VAR: {                           // a variable: substituted yet?
        u64 loc = term_val(next);
        Term cell = heap_read(loc);
        if (term_sub_get(cell)) { next = term_sub_set(cell, 0); goto enter; }
        whnf = next; goto apply;
      }
      case APP: {                           // an application: remember it, evaluate the function
        u64  loc = term_val(next);
        Term fun = heap_read(loc);
        stack[s_pos++] = next;
        next = fun;
        goto enter;
      }
      ...
  apply: {                                  // pop frames, fire the interaction
    while (s_pos > base) {
      Term frame = stack[--s_pos];
      switch (term_tag(frame)) {
        case APP: {
          ...
          switch (term_tag(whnf)) {
            case LAM: { next = wnf_app_lam(whnf, arg); goto enter; }
            ...
```

- **enter** walks to the head of the term, pushing every eliminator it passes (APP, the side of a DUP, MAT, OP2, ...) on an explicit stack. Nothing below the head is touched: that is laziness.
- **apply** pops a frame and dispatches on the pair *(frame tag, value tag)*. This is the active pair of the net model: the eliminator on the stack and the value it now faces. Each pair calls one `wnf_*` function and returns into `enter` with the result.

The explicit stack (not C recursion) means deep terms do not overflow the C stack, and the loop can stop at any interaction (`STEPS_ITRS_LIM`, used by `-D`) and **rebuild** the partially reduced term (`wnf_rebuild`) for printing. That is how `-D` can print the whole term after every step.

What the lazy machine gives up: parallelism. HEAD's `--help` says "A sequential Interaction Calculus runtime". The net model's free parallelism (§2.2) is still true of the calculus, but this machine does not exploit it.

8B.4 Lesson 3: graphs are slow; words are fast

From the start of the lineage *(H.46)*:

> "all implementations of this algorithm, until now, represented terms as graphs. This demanded a lot of pointer indirection, making it slow in practice."

Design consequence: nodes are consecutive heap words, and pointers go one way. The memory table *(L.6)*: an APP node is two words `[func, arg]`, a LAM node one word `[body]`, a SUP two words `[tm0, tm1]`, a DUP term `[expr, body]`. The term word pointing at a node carries the tag and label; the node itself carries no header. A port graph needs back-pointers for every wire; HVM4 needs none, because each variable occurs at most once (affinity) and points **to its binder**.

Linked variables and the SUB bit. A VAR's VAL is the location of its lambda's body slot. β is then a single write *(L.6)*:

> "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."

```
fn Term wnf_app_lam(Term lam, Term arg) {     // src/hvm.c:3732
  ITRS_INC("APP-LAM");
  u64  loc     = term_val(lam);
  u32  lam_ext = term_ext(lam);
  Term body    = heap_read(loc);              // the body is the result
  if (lam_ext & LAM_ERA_MASK) {
    return body;                              // unused variable: drop the argument
  }
  heap_subst_var(loc, arg);                   // the variable will now find the argument here
  return body;
}

fn void heap_subst_var(u64 loc, Term val) {   // src/hvm.c:704
  heap_set(loc, term_sub_set(val, 1));
}
```

Read it against §4.1: "var↔arg" is `heap_subst_var(loc, arg)` (the binder's slot now holds the argument, flagged SUB); "body↔result" is `return body`. Constant work, no traversal of the body, no environment, no map from names to values. An unused variable is a flag on the lambda (`LAM_ERA_MASK`), so the ε cell of §4.2 costs nothing at all, which is why the C2 capstone found no erasure step.

8B.5 A duplication is one shared slot

A DUP node is the most delicate structure in the machine *(L.6)*:

> "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."

Both sides of a duplication (`x₀` = DP0, `x₁` = DP1) point at the **same** heap word. Whichever side is forced first runs the interaction and gets its copy; the other copy is parked in the shared slot with the SUB bit, waiting:

```
fn Term heap_subst_cop(u8 side, u64 loc, Term r0, Term r1) {   // src/hvm.c:712
  heap_set(loc, term_sub_set(side == 0 ? r1 : r0, 1));        // park the twin's copy
  return side == 0 ? r0 : r1;                                  // hand back mine
}
```

and the WNF machine, meeting a DP0/DP1, first checks whether its twin already left the copy there:

```
case DP0:
case DP1: {
  u64 loc = term_val(next);
  Term cell = heap_take(loc);
  if (term_sub_get(cell)) {            // the twin already duplicated: take the parked copy
    next = term_sub_set(cell, 0);
    goto enter;
  }
  stack[s_pos++] = next;               // otherwise: evaluate the shared value first
  next = cell;
  goto enter;
}
```

This is "copy incrementally" in its final form. One shared word, two readers; the first reader does one layer of copying (DUP-LAM, DUP-SUP, DUP-NOD) and leaves the other reader's layer behind; the children of that layer get new duplications with the same label (`term_clone_at(…, lab)`), which will be forced only if someone reads them. Notice the line `stack[s_pos++] = next; next = cell;` : when a copier finds a redex in the shared slot, it **evaluates the redex first**, in place, and only then copies the result. That is §4.5 and the single `OP2-NUM-NUM` of §7.5, in two lines of C.

8B.6 The book: immutable definitions, lazy instances

A definition `@f = …` is parsed once into the **book**, an immutable array indexed by name (`BOOK[nam]` , at most 2^{24} names), with variables as de Bruijn levels. Calling `@f` never copies the whole definition:

```
case REF: {
  u32 nam = term_ext(next);
  if (BOOK[nam] != 0) {
    next = term_new_alo(0, 0, BOOK[nam]);   // a lazy instance with an empty bind list
    goto enter;
  }
  ...
```

An **ALO** term points at the book term plus a list of bindings, and is expanded **one layer** each time the evaluator looks at it (sheets `alo_lam` , `alo_dup` , `alo_nod` , `alo_var` , …). The memory doc states the reason *(L.6)*:

> "This keeps static definitions compact while still allowing dynamic sharing during execution."

Why have references at all, if the combinators are Turing-complete? Not because pure combinators cannot express recursion: the HVM2 paper says commutation "can be used to clone data and to perform loops and recursion, although these are preferably done via calls" *(M.1, §4.5)*. The reason is cost: expanding a REF copies a whole definition in one pass, whereas duplicator nodes would copy it one layer per interaction *(M.1, §4.2)*. It also fixes an HVM2 problem: HVM2's eager unfolding of recursive definitions forced programmers to split functions so that a REF would stop the recursion *(M.1, §13.2)*. With ALO, a recursive call is just a lazy instance; it unfolds only if its value is demanded.

8B.7 Memory: a bump allocator and no collector

```
fn u64 heap_alloc(u64 size) {                // src/hvm.c:317
  u64 at   = HEAP_NEXT;
  u64 next = at + size;
  if (__builtin_expect(next <= HEAP_CAP && next >= at, 1)) {
    HEAP_NEXT = next;
    return at;
  }
  fprintf(stderr, "Out of heap memory (need %llu words)\n", (unsigned long long)size);
  exit(1);
}
```

The heap is one anonymous `mmap` of `HEAP_CAP` = 2^{32} words (the VAL field's range), allocated by bumping a pointer; at HEAD nothing is ever freed. The history shows the trade-off being explored: a semi-space copying GC was added "for bounded memory on tail recursion" (`224670d`) and then removed from the pure runtime (`be02279`). Affinity is what makes this coherent: every node has exactly one reader, so there is no question of *when* it may be reclaimed; the current runtime simply chooses the fastest possible allocator and lets very long runs exhaust the heap. (BendRT, the Bend 2 runtime, uses the same ownership fact to emit explicit frees instead; that is the subject of its own paper, *M.2*, and not an interaction-net design.)

8B.8 Collapse: answers live inside terms

A superposition can sit anywhere inside a result, even under a lambda (§6.1). To print plain answers, CNF lifts the first SUP of a WNF term to the root, turning `C[&L{a,b}]` into `&L{C[a], C[b]}` , and the evaluator enumerates the branches with a priority queue *(L.8)*. Two design choices matter:

- **Breadth-first with priorities, not depth-first.** An infinite superposition (`@X` , §6.5) would trap a depth-first enumerator in its first infinite branch. A queue ordered by key, where SUP increases the key and INC lowers it, explores shallow alternatives first and lets programs steer the search. The newest commit (`6defdfc`) generalizes this ordering for search programs ("filter-credit scheduler").
- **Readback is work.** Converting the net-shaped normal form into a λ-term costs interactions (§8B.9). That is why `-C` counts can exceed the plain counts.

8B.9 A full machine trace: Church 2^2 in 14 interactions

`@c2 = λs. !S&C=s; λx. S₀(S₁(x))` , `@k2` the same with label K, `@main = @c2(@k2)` . Without `-C` , HVM4 reduces to normal form in **14 interactions**, the count HVM4's theory doc calls optimal *(L.4, §Church 2^2)*. The rule sequence from `-D` , annotated:

#	Rule	What happens
1	APP-LAM	`@c2` receives `@k2` as `s` ; the outer duplication `!A&C = @k2` now holds the inner numeral
2	DUP-LAM	the C-copier meets `@k2` 's λ: two λ-copies, the inner variable becomes `&C{b,c}`
3	APP-LAM	the outer numeral's inner application fires: copy 1 is applied to `x`

#	Rule	What happens
4	DUP-LAM	the C-copier meets the inner numeral's second λ (`λx`)
5	DUP-SUP	C-copier meets a **K** superposition: different labels, **commute** (this is where `c2(c2)` would project and break)
6	APP-SUP	an application meets a C-superposition: distribute, duplicate the argument
7	DUP-SUP	C meets C: **project**
8	APP-LAM	β inside the copied numeral
9	DUP-LAM	K-copier copies a λ
10	APP-LAM	β
11	APP-SUP	distribute over a C-superposition
12	DUP-SUP	C meets C: project
13	DUP-SUP	C meets C: project
14	APP-LAM	the last β

The result printed after step 14 is **not** a λ-term:

```
λa.λb.A₀;!A&K=B₀(B₁(&K{A₁,b}));!B&K=a;
- Itrs: 14 interactions
```

It is the normal form *as a net*: two live K-duplications sharing `a` (the successor function) between four uses. That is what optimal sharing looks like when it is finished: the four applications of `a` in `4 = λa.λb.a(a(a(a b)))` are represented by **two** duplications, not four copies. Only `-C` reads it back to the λ-term, spending 6 more interactions (total 20). Normal forms in this calculus are graphs; λ-terms are their unfolding.

HVM4 TRACE · `start` APP-LAM ×5 DUP-LAM ×3 DUP-SUP ×4 APP-SUP ×2

step ·

```
@main
```

step ·

```
@{!A&C=@{@k2};λb.A₀(A₁(b))}
```

step ·

```
λa.(λb.A₀)(@{λc.A₁(a)});!A&C=@{!A&K=&C{b,c};λb.A₀(A₁(b))};
```

step ·

```
λa.A₀;!A&C=@{!A&K=&C{@{λb.A₁(a)},b};λb.A₀(A₁(b))};
```

step ·

```
λa.λb.A₀;!A&C=@{B₀(B₁(&C{b,c}))};!B&K=@{&C{@{λd.λc.A₁(a)},d};
```

step ·

```
λa.λb.A₀;!A&C=&C{B₀,C₀}(@{&C{B₁,C₁}(&C{b,c})});!B&K=@{λd.λc.A₁(a)};!C&K=d;
```

step ·

```
λa.λb.A₀;!A&C=&C{B₀(C₀),D₀(C₁)};!B&K=@{λc.λd.A₁(a)};!C&C=@{&C{B₁,D₁}
(&C{b,d})};!D&K=c;
```

step ·

```
λa.λb.A₀(B₀);!A&K=@{λc.λd.C₀(B₁)(a)};!B&C=@{&C{A₁,C₁}(&C{b,d})};!C&K=c;
```

step ·

```
λa.λb.A₀(B₀);!A&K=λc.C₀(B₁);!B&C=@{&C{A₁,C₁}(&C{b,c})};!C&K=@{a};
```

step ·

```
λa.λb.(λc.A₀)(B₀);!A&K=C₀(B₁);!B&C=@{&C{λd.A₁,C₁}(&C{b,&K{c,d}})};!C&K=@{a};
```

step ·

```
λa.λb.A₀;!A&K=B₀(C₁);!B&K=@{a};!C&C=@{&C{λc.A₁,B₁}(&C{b,&K{C₀,c}})};
```

step ·

```
λa.λb.A₀;!A&K=B₀(C₁);!B&K=a;!C&C=&C{λc.A₁(D₀),B₁(D₁)};!D&C=@{&C{b,&K{C₀,c}}};
```

step ·

```
λa.λb.A₀;!A&K=B₀(B₁(C₁));!B&K=a;!C&C=@{&C{b,&K{λc.A₁(C₀),c}}};
```

step ·

```
λa.λb.A₀;!A&K=B₀(B₁(&K{λc.A₁(b),c}));!B&K=a;
```

step ·

```
λa.λb.A₀;!A&K=B₀(B₁(&K{A₁,b}));!B&K=a;
```

Recorded from HVM4 HEAD `6defdfc` with `-D`; rule names are HVM4's own.

8B.10 One file, one thread

At HEAD the runtime is a single translation unit with no threads, no FFI and no AOT compiler (§8.8). The commit that made it so is titled "Refactor HVM runtime into pure single source" and carries no further rationale; this book does not invent one. The observable consequences:

1. The whole runtime can be read top to bottom in one file of about 6,400 lines, with the interaction rules and their sequent comments side by side.
2. `Itrs` is deterministic and identical across runs, and `-D` can print every step.
3. The net model's parallelism is a property of the calculus that this implementation does not use. The parallel engines of the lineage are HVM2 (interaction combinators on CPU and GPU, *M.1*) and the CUDA experiment that preceded the refactor (§8.8).

What to remember from Part 8B

- **The problem is clones; the answer is a lazy clone primitive**, not references.
- **HVM2's lessons became HVM4's design:** one duplicator kind is unsound for higher-order sharing → **labels** (an integer compare instead of a 10× oracle); eager reduction cannot be lazy → a **lazy explicit-stack WNF machine**.
- **Words, not graphs:** nodes are consecutive heap words; variables point to their binder; β is one write with the SUB bit.
- **A DUP is one shared slot:** the first reader copies one layer and parks the twin's copy; a redex in the slot is evaluated before it is copied.
- **Book + ALO:** immutable definitions, lazily instantiated one layer at a time.
- **Memory:** a 2^{32}-word bump allocator, no collector at HEAD.
- **Normal forms are nets:** 2^2 finishes in 14 interactions with two live duplications; readback costs 6 more.

Exercises — Part 8B

E8B.9 ★★ Taelin's 2022 comment lists "some upward edges can't disappear (variables)". In HVM4's memory format, which pointer is that upward edge, and which interaction writes through it?

▾ SOLUTION

1. A linked VAR's VAL field is the heap location of its λ's body slot: a pointer from a leaf of the expression back up to its binder.
2. APP-LAM (`wnf_app_lam`) writes the argument into that slot with the SUB bit (`heap_subst_var`), so the variable finds its value by following its one upward pointer.
3. All other edges point downward and need no back-pointer, which is the "turn the graph into a tree" step.

E8B.1 ★ Which single line of `wnf_dup_sup` decides between projection and commutation, and what does it cost?

▾ SOLUTION

`if (lab == sup_lab)` : one integer comparison of two EXT fields. Lamping's oracle would need run-time bookkeeping nodes, estimated in the HVM2 paper at about a 10× slowdown.

E8B.2 ★★ Trace `heap_subst_cop` for `!x&A = &A{1,2}; [x₁, x₀]` when the list forces `x₁` first. What is written where, and what does `x₀` find later?

▾ SOLUTION

1. Forcing `x₁` (DP1, side 1) finds the shared slot holding `&A{1,2}` , not yet substituted: it pushes itself and evaluates the slot to a SUP; DUP-SUP with equal labels projects.
2. `heap_subst_cop(1, loc, 1, 2)` writes the **twin's** result `r0 = 1` into the slot with SUB, and returns `r1 = 2` to side 1.
3. Later `x₀` (DP0) reads the slot, sees the SUB bit, and takes `1` without any interaction.
4. Result `[2, 1]` , 1 interaction.

E8B.3 ★★ Why can HVM4 store an unused λ-variable as a flag instead of an eraser node, and where is the flag tested?

▾ SOLUTION

1. An unused variable has no occurrence, so nobody will ever read the binder's slot; there is nothing to erase *to*.
2. The flag `LAM_ERA_MASK` in the lambda's EXT is tested in `wnf_app_lam` : if set, the argument is dropped (`return body`) without a substitution.
3. The argument itself is simply never referenced again; with a bump allocator there is no free either.

E8B.4 ★★ HVM2's paper says a function written `foo = x => x == 0 ? 0 : 1 + foo(x-1)` hangs. Why, and why does the same function terminate on HVM4?

▾ SOLUTION

1. HVM2 reduces all redexes eagerly: the recursive call `foo(x-1)` is unfolded even in the branch that will be discarded, forever.
2. HVM4's WNF machine only reduces the head; a REF becomes a lazy ALO instance that is expanded only when its value is demanded, and the discarded branch is never demanded.

E8B.5 ★★ In §8B.9, which step is the one where `@c2(@c2)` goes wrong, and what does it do instead?

▾ SOLUTION

Step 5: the C-copier meets the inner numeral's superposition. With `@k2` its label is K, so they commute. With `@c2` it is C, so DUP-SUP takes the `lab == sup_lab` branch and **projects**, wiring the outer copies to the inner numeral's two uses. The rest of the reduction then computes the degenerate `λa.λb.a` .

E8B.6 ★★★ The 2^2 normal form `λa.λb.A₀;!A&K=B₀(B₁(&K{A₁,b}));!B&K=a;` contains two duplications. Read it back by hand to `λa.λb.a(a(a(a(b))))` .

▾ SOLUTION

1. The body is `A₀`, side 0 of `!A&K = B₀(B₁(&K{A₁,b}))`.
2. Side 0 of a duplication of an application is that application with its parts' side-0 copies: `B₀'(B₁'(&K{A₁,b})₀)` with B duplicated under K again. `B` duplicates `a`, so each `Bᵢ` reads as `a`.
3. `(&K{A₁,b})₀` under label K projects to `A₁`; `A₁` unfolds the same expression on side 1: `a(a(&K{A₁,b}₁))`, and `&K{A₁,b}₁` projects to `b`.
4. Altogether `A₀ = a(a(a(a(b))))`: four uses of `a` from two duplications. The collapser does exactly this, in 6 interactions.

E8B.7 ★★ Why does a breadth-first, prioritized collapse matter for `@X = &Z{#Z{}, #S{@X}}` but not for `&A{1,2}`?

▾ SOLUTION

`&A{1,2}` has two finite branches; any order prints both. `@X` has an infinite branch (`#S{@X}` unfolds forever); a depth-first enumerator could descend into it without end and never print the finite answer, while a priority queue that increases the key at each SUP visits shallow branches first.

E8B.8 ★★★ The HEAD allocator never frees. Name one property of the calculus that makes freeing *easy* in principle and one reason a runtime might still choose not to.

▾ SOLUTION

1. Affinity: every node is read by exactly one consumer, so a node can be freed (or reused) the moment its interaction consumes it — no reachability analysis is needed.
2. Freeing costs work on every interaction (free lists, fragmentation, cache effects); a bump allocator is the cheapest possible allocator. The repository tried a copying GC and removed it, choosing simplicity and speed at the cost of bounded runs.

<hr>

9 · Optimality and parallelism, told honestly

9.1 What "optimal" means

Lévy (1978–80) defined optimal reduction for the λ-calculus. Redexes that are "copies of the same redex" (created by duplicating one subterm) form a *family*; an evaluator is **optimal** if it reduces each family in one step, never duplicating a redex before reducing it. Lévy proved that no strategy on *terms* can do this in general: terms force you to copy before you know what will be needed.

Taelin's own statement of the guarantee *(H.53)*:

> "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."

Read it word by word:

- **"in Lévy's sense"**: the measure is **β-reductions** (APP-LAM), not interactions, not time.
- **"admissible"**: terms for which the label discipline is correct (the clone-of-a-clone cases of §5.6 are excluded; see 9.3).
- **"minimal number of β-reductions"**: every family once.

9.2 Lamping: the abstract algorithm and the oracle

Lamping (1990) gave the first optimal algorithm, as a graph-rewriting system: the λ-net of Part 4, plus fans for sharing, plus **control nodes** (brackets and croissants) that track the *nesting level* of each fan so that the right fans annihilate and the others commute. Taelin's history of the field *(H.41)* separates the two halves:

> "Lamping's algorithm is actually divided into two parts: the **Abstract Algorithm** and the **Oracle**."

- The **abstract algorithm** is the fan-and-λ part: exactly the rules of Parts 3–5.
- The **oracle** is the bookkeeping that decides, for two fans that meet, whether they belong together. It is what makes the algorithm correct for *all* λ-terms, and it is expensive.

Labels (Part 5) are a **static, local oracle**: the program, not the run time, says which fans belong together. That is cheap and decidable, and it is incomplete: it fails exactly when a fan must meet a same-labelled fan that is not its partner (§5.6, §7.2).

9.3 Why optimal reduction was not used, and the two reasons

Taelin's first implementation of Lamping's algorithm, Optlam, surprised him *(H.41)*:

> "It was miraculously fast. Impossibly so."

His diagnosis of why the field nonetheless abandoned it, in his words:

> "1. A *negative result* that was misinterpreted by researchers back then"
>
> "2. The theoretical optimality didn't translate to real-world performance"

The negative result is **Asperti–Mairson (1998)**: for any implementation of Lévy-optimal reduction, the time needed for n parallel β-steps (n redex families) is not bounded by any elementary function of n (not by 2^n, nor 2^{2^n}, and so on). Optimality in Lévy's sense counts β-steps; the rest of the work is not counted, and it can be enormous. A later result sharpens *where* that work goes. **Asperti, Coppola and Martini (2004)** showed that the *duplication* interactions alone are already not elementary: the blow-up is not caused by the oracle's bookkeeping but by the inherent sharing of λ-terms. The same work gives the fact that matters most for HVM (as summarized in Laretto's seminar on the complexity of optimal reduction, *M.5*):

> "If a term can be given a type in EAL (for which they give a type-inference algorithm), bookkeeping is not necessary and a simple labelling of fan nodes suffices."

(EAL is Elementary Affine Logic.) This is the closest theoretical ground for HVM's choice of **labels instead of an oracle** (§5, §8B.2): there is a large, typable class of terms for which static fan labels are enough. HVM does not run an EAL checker; it leaves label correctness to the program or compiler (§5.6).

What happened when the full algorithm was engineered is recorded too *(M.5)*. The Bologna Optimal Higher-order Machine, **BOHM** (Asperti, Giovannetti and Naletto, 1996), was much faster than SML, Caml Light and Yale Haskell on pure λ-terms with heavy sharing, often turning exponential cases into polynomial ones, but about one order of magnitude slower than call-by-value languages on ordinary programs. BOHM 1.1 added **safe** fans, fans that can never pair with another fan and may therefore be collected early; that idea returns in §9.4. The second reason is engineering: graph reducers with many pointer indirections are slow on real hardware. Taelin, on what HVM changed *(H.46)*:

> "all implementations of this algorithm, until now, represented terms as graphs. This demanded a lot of pointer indirection, making it slow in practice."

HVM's answer is the term machine of Part 8: the same interactions, a compact memory format.

9.4 The remaining gap: the quadratic slowdown

Optimal in Lévy's sense is not the end. Taelin, after a user found a program that HVM ran slower than GHC *(H.53)*:

> "function which HVM computes quadratically slower than GHC"

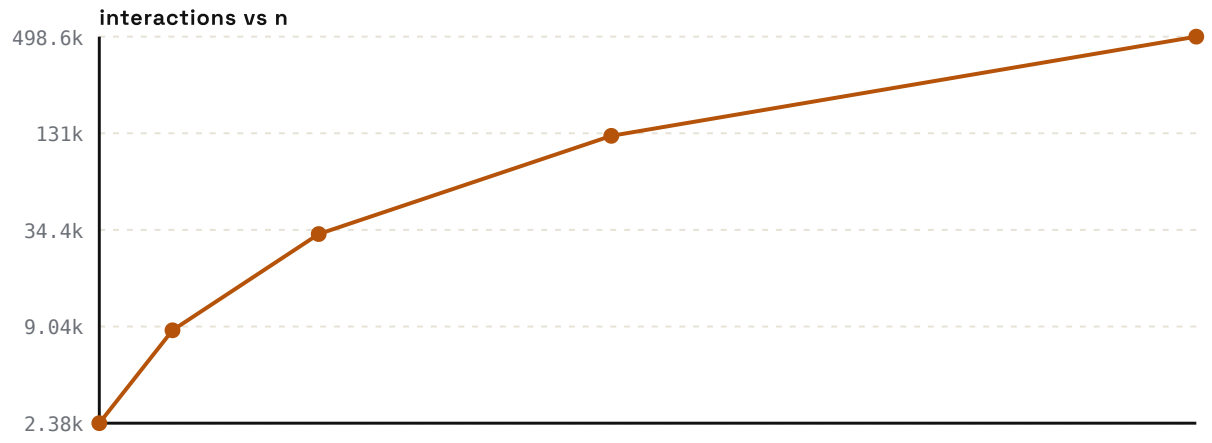
> "The culprit is the accumulation of "redundant" control nodes or duplication nodes in some cases."

Measure it on HVM4. The essay's example is "solve X = n", with X the superposition of all naturals from §6.5. A good logic language unifies this in O(1); the best a superposition search can do is linear (it must compare n constructors). Here is the program, with the nat literal written out:

```
@eq   = λ{ #Z: λ{ #Z: #T{}; #S: λbp. #F{} }; #S: λap.λ{ #Z: #F{}; #S: λbp. @eq(ap, bp) } }
@X    = &Z{#Z{}, #S{@X}}
@N    = #S{#S{#S{#S{#Z{}}}}}
@main = @eq(@X, @N)
```

Measured on HEAD `6defdfc` with `-s` (no collapse), for `@N` = n:

n	interactions	ratio to the previous row
16	2,376	—
32	8,576	3.6
64	32,496	3.8
128	126,416	3.9
256	498,576	3.9




```
n interactions interactions

Plotted from the table above.
```

Doubling n multiplies the work by almost 4: **quadratic**. The control `@eq(@N, @N)` at n = 256, the same comparison without a superposition, costs 1,538 interactions. Counting the rule names in the `-D` trace shows where the time goes: at n = 32, **4,455** of the interactions are `DUP-NOD` (a duplicator copying a constructor), against 97 `APP-LAM`.

The mechanism. Every branch of X that fails still leaves its duplicator behind, so each later `#S` layer of n must pass through a longer chain of duplications before it reaches the comparison. Taelin's hand trace of `X == 4` marks those steps "WASTED" and describes them *(H.53)*:

> "the constructor has to jump between a chain of "redundant" duplications"

> "To fully copy the Peano Nat `1000`, we would need about 1 million interactions!"

Three answers.

1. **Safe nodes** (Asperti, BOHM 1.1, §9.3): remove a duplicator whose other side is erased. Taelin's assessment *(H.53)*: Asperti, BOHM's author, "patched" this by using "safe nodes", which "works well in practice" but is "more of a workaround that doesn't work in all cases". The rule is unsafe in general because a SUP with the same label elsewhere may still need the DUP.
2. **Unordered duplications.** The essay's key observation, in its own heading: "Thunks are Unordered DUP Nodes". If the runtime may permute a DUP's two outputs, a chain of duplications collapses into one shared value that every reader reaches in O(1): this is a Haskell thunk inside an interaction system. Taelin implemented it experimentally in HVM3 and reports, for `X + 12345 = 1000000` *(H.53)*:

   > "The program above correctly outputs `987655` in just `24m` interactions, showing it is clearly linear on the length of the Nat (which is `1m`)."

3. **The price.** Duplicating a λ with an unordered DUP can swap the bodies of the two copies, and the essay notes that unordered nodes "might break strong confluence". HVM4 at HEAD has **no** unordered superpositions (the word does not occur in its source or docs); the measurement above is what HEAD does.

9.4a Effects: where a confluent net meets the world

A pure net can be reduced in any order (§2.2), but printing, reading input or sending a message must happen in a fixed order. Marvin Borner's thesis *Optimal Interaction with the Real World* (Tübingen, 2025, *M.6*) is a dedicated study of this tension. Its reconstruction of HVM2's IO (from HVM2's source, §2.1 of the thesis) is the clearest description available:

1. reduce the whole net to normal form;
2. read the result back as a Church tuple carrying a tag, a magic number, the name of an effectful function, its arguments and a continuation;
3. if it is an IO call, run the named host function and apply the continuation to its result; repeat.

The thesis's assessment: this keeps the parallel reduction intact, but no effect can run in parallel with pure reduction or with another effect, and the source language must be written in continuation (monadic) style, as in Bend. The thesis then proposes token-passing semantics to sequence effects inside an otherwise unordered net. **HVM4 at HEAD has no IO at all:** none of its 45 tags is an effect, and the only file operation in `src/hvm.c` loads the program.

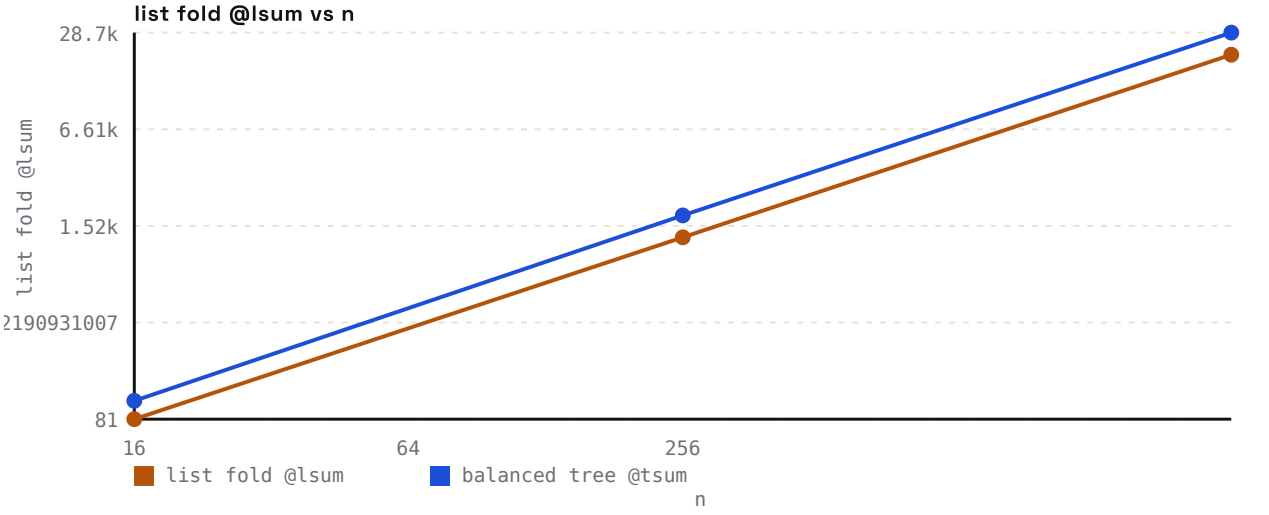
9.5 Parallelism: what locality gives and what it does not

From §2.2: all active pairs present at one moment are disjoint, so a machine can fire them all at once, in any order, with the same total count. That is a real theorem and the foundation of HVM2's parallel GPU runtime (the engine behind the first Bend). Two limits:

1. **Speedup needs many simultaneous redexes.** A program whose net exposes one active pair at a time (a sequential fold, bubble sort) gains nothing. Programs whose nets fan out (tree merges, independent recursive calls) can.
2. **HVM4 at HEAD is sequential** (`--help` : "A sequential Interaction Calculus runtime"). The March 2026 tree had a multi-threaded collapse and the CUDA experiment reported tens of billions of interactions per second on the authors' benchmarks (§8.8); neither is at HEAD.

Trees, not lists. Parallelism in a net is a property of its dependency shape, and the cleanest way to see it is to separate **work** (total interactions) from **span** (the longest chain of interactions that must happen one after another). Summing n ones on HVM4 HEAD, measured with `-s` :

n	list fold `@lsum`	balanced tree `@tsum`
16	81	107
256	1,281	1,787
4,096	20,481	28,667



Both are linear (exactly 5n + 1 and 7n − 5): the tree does **more** work, not less. What differs is the span. In the list fold every addition waits for the result of the rest of the list, a chain of n dependent steps; in the tree the two halves share no wire, so the chain is only as long as the tree is deep, about $\log_2 n$. On a sequential machine like HVM4 HEAD the tree is simply 40% more expensive; on a parallel interaction machine (HVM2, M.1) it is the version that can use many cores. That is the practical rule behind "use balanced trees, not lists" in the Bend lineage.

The locality argument is also why affinity matters. From the IC README *(H.33)*:

> "Finally, being fully affine makes its garbage collector very efficient, and greatly simplifies parallelism."

9.5a Where nets come from: proof nets, linear logic, the geometry of interaction

Interaction nets were not invented for evaluators. Mazza's introduction *(M.4, §1)*:

> ”Based on graph-rewriting, they can be seen as a generalization of multiplicative linear logic proof-nets”

In a proof net a **cut** connects a proof of A with a proof of its dual A⊥, and cut elimination rewrites it locally: the cut is the active pair, cut elimination is interaction. Linear logic's exponential `!A`, ”A may be used many times”, is what a duplicator implements. Asperti made that precise for optimal reduction: his 1993 paper relates the two kinds of bracket nodes of the Gonthier–Abadi–Lévy presentation of Lamping's algorithm to the two operations of the comonad `!` *(Appendix F)*.

The third root is Girard's **geometry of interaction** (GoI), a semantics that describes the *execution* of a proof as paths through its graph rather than only its result. Interaction nets ”have been shown to be the ”true” syntax underlying Girard's Geometry of Interaction” *(M.4, §1)*; Gonthier, Abadi and Lévy's paper ”The Geometry of Optimal Lambda-Reduction” (1992) used GoI to explain Lamping's algorithm *(M.5)*. For the combinators the link is a theorem (Lafont 1997, as stated by Mazza, *M.4* Theorem 4.7): a net is interpreted as a matrix-like operator, and the interpretation of its normal form is computed by Girard's **execution formula**, a geometric series that sums the paths through the net's active pairs. The formula is an invariant of reduction, so a net and its normal form have the same GoI meaning, which is confluence seen from the semantic side.

None of this is needed to write HVM programs. It explains two things the rest of the book uses: why the six rules are confluent and local (they come from cut elimination), and why the duplicator, `!`, is the one place where cost and correctness become hard (Parts 5 and 9).

9.6 A claim-class table

Use this table whenever you read a sentence about these systems:

Claim	Class	Status
Interaction-net reduction is strongly confluent; counts are scheduler-independent	theorem	proved (Lafont)
The interaction combinators can encode any interaction system	theorem	proved (Lafont 1997)
HVM reduces admissible terms with the minimal number of β-steps	theorem + definition	in Lévy's sense; ”admissible” carries the weight
n parallel β-steps of any Lévy-optimal implementation can take non-elementary time	theorem	Asperti–Mairson 1998
Duplications alone are already non-elementary; EAL-typable terms need only simple fan labels	theorem	Asperti–Coppola–Martini 2004 (via M.5)
HVM's interactions for `X = n` by superposition grow quadratically	measurement	§9.4, `HEAD 6defdfc`
The symmetric combinators are universal for every interaction system	**false as stated**	universal for *polarized* systems (Mazza, M.4)

Claim	Class	Status
"Sharing computations inside lambdas, which GHC can't"	mechanism + claim	the mechanism is real (§7.5); "can't" is about GHC's semantics, not every program
"Some computations exponentially faster"	measurement on chosen families	true for specific programs; never in general
HVM4 at HEAD runs N G interactions/s	measurement	HEAD is sequential; throughput claims refer to specific runtimes and benchmarks
Bend 2 runs on interaction nets	**false for the released compiler**	BendRT is a flat evaluator (Part 10)

What to remember from Part 9

- Lévy-optimal = each redex family reduced once; it counts **β-steps**, not total work.
- Lamping = abstract algorithm (Parts 3–5) + oracle (control nodes). Labels are a cheap, static, incomplete oracle.
- Two reasons optimality was abandoned: Asperti–Mairson's non-elementary bound, and slow graph implementations. Asperti–Coppola–Martini locate the blow-up in duplication itself and show that simple fan labels suffice for EAL-typable terms; HVM uses labels and a term machine.
- The quadratic slowdown is measurable on HVM4 (`X = n` : ×3.9 per doubling, dominated by DUP-NOD); safe nodes and unordered duplications are the known remedies, neither at HEAD.
- Effects need an order that a confluent net does not have; HVM2 ran them between full normalizations, HVM4 HEAD has none.
- Locality gives scheduler-independent counts and free parallelism; speedup needs many simultaneous redexes; HVM4 HEAD is sequential.

Exercises — Part 9

E9.1 ★ What does "optimal in Lévy's sense" measure, and what does it ignore?

▾ SOLUTION

It measures the number of β-reductions (one per redex family). It ignores the cost of duplication, erasure and oracle bookkeeping, and wall-clock time.

E9.2 ★★ Why is the label discipline called a "static oracle", and when does it fail?

▾ SOLUTION

1. Lamping's oracle decides at run time whether two meeting fans are partners. Labels decide it ahead of time: equal labels are partners, different labels are not.
2. It fails when two fans with equal labels meet without being partners, which happens when a fan duplicates a term containing a fan of the same label (`@c2(@c2)` , §5.6).

E9.3 ★★ Classify: (a) "HVM computes the normal form of every λ-term with the minimal number of β-steps." (b) "Interaction counts are the same under any scheduler." (c) "Superposition makes SAT polynomial."

▾ SOLUTION

1. (a) **False as stated**: only admissible terms; Lévy's measure only.
2. (b) **Theorem** (strong confluence).
3. (c) **False**: sharing can help on structured instances, but 2^n distinct sub-computations still cost 2^n.

E9.4 ★★ A program's net always has exactly one active pair until normal form. How much can a parallel interaction machine speed it up?

▾ SOLUTION

Not at all: each step creates at most the next active pair; there is never more than one rewrite to do. The theorem guarantees *safety* of parallelism, not its availability.

E9.5 ★★★ Explain how Asperti–Mairson's result and Taelin's "quadratically slower than GHC" report are related but different.

▾ SOLUTION

1. Asperti–Mairson: the total work of n parallel β-steps can be non-elementary in n for any Lévy-optimal implementation; Asperti–Coppola–Martini later showed the duplications alone already are.
2. The HVM report: a concrete program where *redundant duplication nodes* accumulate, making HVM's total interactions grow quadratically where GHC's work is linear.
3. Both say "β-optimal is not work-optimal". The first is a worst-case theorem about every optimal implementation; the second is an avoidable overhead of a particular implementation, on a specific family of programs (§9.4 measures it on HVM4).

E9.9 ★★ In the §9.5 table the balanced tree costs $7n - 5$ interactions and the list fold $5n + 1$. Why can the tree still be faster on a parallel interaction machine?

▾ SOLUTION

1. Interactions measure work; wall-clock on many cores is bounded by the span, the longest chain of dependent interactions.
2. In the list fold each addition needs the sum of the rest: a chain of n steps.
3. In the tree, the two subtrees share no wire, so their active pairs can fire at the same time; the chain is about $\log_2 n$ levels.
4. With p cores the tree needs roughly work/p + span time, the list roughly n whatever p is. On HVM4 HEAD (sequential) only work counts, and the tree is slower.

E9.10 ★★ Match each linear-logic notion to its interaction-net counterpart: cut, cut elimination, `!A`, normal form of a proof.

▾ SOLUTION

1. Cut → an active pair (two principal ports wired together).
2. Cut elimination → interaction (local rewriting of the pair).
3. `!A` (reusable resource) → what duplicators implement; copying and erasing are explicit cells.
4. Cut-free proof → cut-free net: no active pair and no vicious circle (§2.4).

E9.6 ★★ From the §9.4 table, estimate the interactions for n = 512 and explain in one sentence why the control `@eq(@N, @N)` is linear.

▾ SOLUTION

1. The ratio is approaching 4, so about 4 × 498,576 ≈ **2.0 million**.
2. The control has no superposition: there is no duplicator chain for each `#S` layer to pass through, so each layer costs a constant number of interactions (1,538 for n = 256, about 6 per layer).

E9.7 ★★★ Why is "delete a duplicator whose other output is erased" unsafe in general, and what does BOHM's "safe" flag certify?

▾ SOLUTION

1. A duplicator with label L may still be the partner of a superposition with label L somewhere else in the net; deleting it would make that SUP later meet the wrong DUP (or none), changing the result.
2. A safe fan is one that can never pair with another fan, so no partner exists and removing it cannot change any later interaction. BOHM 1.1 marks such fans and collects them early.

E9.8 ★★ Strong confluence says the order of interactions is unobservable. Why does that make IO hard, and how did HVM2 avoid the problem?

▾ SOLUTION

1. Effects are observable in their order (print A then B differs from B then A); a system whose rules may fire in any order cannot express that order by itself.
2. HVM2 reduced the net to normal form, read the result back as a request (function name, arguments, continuation), ran the effect outside the net, and continued with the result: effects happen *between* normalizations, one at a time.
3. The cost: no effect runs in parallel with reduction or with another effect, and programs are written in continuation or monadic style.

<hr>

10 · The lineage: HVM1 → HVM4, Bend 2, BendRT, HVM5

10.1 The runtimes

Runtime	Character	Notes
HVM1 (HigherOrderCO/HVM)	lazy IC runtime in Rust, Haskell-like surface language	"HVM: how it works" *(H.46)* explains its memory format
HVM2	strict, massively parallel (CPU threads, CUDA) interaction-net runtime	the backend of the first Bend (2024)
HVM3	consolidation: lazy evaluation and labels again	Haskell/C *(H.29)*
HVM4 (this book's machine)	the IC with labeled DUP/SUP, numbers, constructors, matching, collapse	HEAD: one sequential C file (Part 8)

10.2 Bend 2 and BendRT: the nets are not in the released compiler

This book is about interaction nets, so one paragraph settles Bend 2: **its runtime does not use them.** The BendRT paper *(M.2)* says so in its introduction, "Readers of the author's earlier runtimes may expect interaction nets [7, 13] here; there are none", and states the trade in its related-work section:

> "BendRT keeps the goals and drops the mechanism: affinity gives unique ownership directly, so the graph and its duplication machinery become unnecessary. Lost is optimal reduction of shared redexes; gained are native-speed sequential code, flat memory and a cost model a programmer can read."

(The paper discloses that it "was written by Claude Fable 5.1 from the author's code and design notes, and reviewed by the author", so treat its wording as the authors' reviewed statement, not as Taelin's own prose.) That is the clearest statement of the whole lineage's central trade-off: **optimal sharing versus predictable native speed.** HVM keeps the first; BendRT chose the second. Released Bend 2 has no `sup`, no DUP-LAM and no SupGen; the language itself (affine dependent types, laws and proofs, targets) is documented in the compendium, Part A and papers M.2–M.3, and is outside the scope of this book.

10.3 HVM5 in this workspace: what it is

HVM5 is a design and prototype in `bend-runtime/hvm5/`, in two versions. Its thesis, from the v2 README *(G.51)*:

> "Code is an immutable, content-addressed book of interaction-calculus terms."

and the three operations that follow from it:

> "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."

10.4 Homoiconicity, concretely

"Code as data" here means four precise things:

1. **A program is a book**: a map `Name → DefHash`, where each definition is an IC term identified by the hash of its contents. Two books that share a definition share its hash and its storage.
2. **A running instance is `(book, state, epoch)`**: the immutable book plus a heap of live state.
3. **Fork = duplicate the state**: `!&F{live, shadow} = state`. One DUP node; copying happens lazily, only where one side rewrites — the incremental copy of Part 4, applied to a whole program.
4. **Test = superpose the book**: every definition whose hash changed becomes `@d = &T{@d_old, @d_new}`; unchanged definitions stay single. One reduction runs both versions, and everything computed before control reaches a changed definition is computed once.

The v2 review *(G.52, §C3)* insists on the precision of the sharing claim:

> "Two *independently written* code paths that happen to compute the same thing are not shared. Report the measured `sharedRatio`, never a promised one."

10.5 HVM5 status, honestly

From the v2 README's status table (dated 2026-10-23, *G.51*; re-read the file for later updates):

Milestone	Status in the README
M1 IC kernel	"done (sequential)"; `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"
M3 code as data	"done on one island"; `tests/e2e_vault.bend` PASS
M4 membrane (capabilities)	"checks, untested"
M5 islands	in-process mesh done; "TCP not started"

v1 is explicitly **superseded**. Its own README *(G.2)*:

> ”Its reducer is a name-substitution tree evaluator (not an inet), and its fork, hashing and partition models are corrected in”

and the v2 review *(G.52)* grades v1’s reducer **”F as an inet”** and lists what to stop claiming, including:

> ”Bend 2 compiles to interaction nets.” Upstream it does not.

> ”Fork = DUP, merge = SUP collapse.” Fork is `DUP` of state, test is `SUP` of books, merge is CAS.

Note K3: the HVM5 kernel reproduces the same-label accident of §5.6 as a deliberate test. Two kernels, the same lesson.

What to remember from Part 10

- HVM1 lazy → HVM2 parallel/strict (first Bend) → HVM3 → **HVM4** (IC + labels + collapse; HEAD sequential, single file).
- **BendRT is not a net reducer**: it keeps affinity and drops the graph, trading optimal sharing for native speed; released Bend 2 has no `sup`.
- HVM5 v2: code = content-addressed book; **fork = one DUP of state; test = SUP of books; merge = 3-way merge + CAS**. Kernel done sequentially; capabilities untested; networking not started.

Exercises — Part 10

E10.1 ★ Which runtime ran the first Bend’s programs, and is it a net reducer?

▾ SOLUTION

HVM2, a strict parallel interaction-net runtime: yes. Released Bend 2 runs on BendRT, which is not.

E10.2 ★★ Why is ”fork = SUP on the live root” wrong, according to the HVM5 review, and what replaces it?

▾ SOLUTION

1. Superposing the live root makes every live message run in two worlds, and the live answer then depends on a collapse policy: the program is no longer one program.
2. Replacement: fork = one **DUP** of the state (two independent instances, sharing until they diverge); testing = **SUP of books**, run deliberately on a test input.

E10.3 ★★ In superposed testing, a request reaches a changed definition after 1,000 interactions and then runs 200 more on each side. Roughly how much work does the superposed run cost versus two separate runs?

▾ SOLUTION

1. Separate: 2 × (1000 + 200) = 2400.
2. Superposed: 1000 shared + 2 × 200 + the overhead of the superposition (APP-SUP/DUP-SUP bookkeeping) ≈ 1400 + overhead.
3. The saving is the common prefix only; report the measured shared ratio (≈ 1000/2400 here).

E10.4 ★★ What does "content-addressed book" buy for merging, and what does it not decide?

▾ SOLUTION

1. A three-way merge compares definition *hashes*: a definition changed on one side only is taken; unchanged definitions are identical by hash; conflicts are definitions changed differently on both sides.
2. It does not decide *semantic* compatibility of the state with the new code: that is the optional `migrate` step, and conflicting definitions still need a human or an application rule.

E10.5 ★★★ Explain why HVM5's K3 test and HVM4's `@c2(@c2)` show the same phenomenon although the two kernels are independent programs.

▾ SOLUTION

Both implement the Interaction Calculus with labels, and the phenomenon is a property of the calculus: when a duplication with label L copies a term containing a duplication with label L, the two fans annihilate instead of commuting. Any faithful implementation must reproduce it; a kernel that "fixed" it would not be implementing IC.

<hr>

11 · Victor Taelin's insights, verbatim

Every quote below is copied exactly from the file named; the compendium id points to the full text. Paraphrase is marked as such.

I-1 · Substitution is the problem. *(H.42, `victor/hoc-optimal-computer.md`)*

> "The fundamental operation of the Lambda Calculus, substitution, may trigger an unbounded amount of copies of an unboundedly large argument."

Meaning: the cost of β is not a constant. *Mechanism:* the interaction rules split it into bounded steps (Part 3). *Class:* argument.

I-2 · Two laws. *(H.42)*

> "on which substitution is broken down into 2 fundamental laws: commutation, which creates and copies information, and annihilation, which observes and destroys information."

Mechanism: γδ commutation (§3.6) vs the annihilations (§3.2–3.4). *Class:* characterization of Lafont's combinators.

I-3 · Why not SKI. *(H.42)*

> "SKI combinators still include non-atomic operations: K may erase an unboundedly large structure, and S may copy an unboundedly large structure."

Mechanism: compare with erasure (one cell per interaction, §3.5) and copying (one layer per interaction, §3.6). *Class:* argument.

I-4 · Copy incrementally. *(H.43, `victor/hvm-10min.md`)*

> "Fortunately, it turns out there IS a proper way to do it: just **copy incrementally**."

Mechanism: DUP-LAM / DUP-NOD one layer at a time; the copier waits at a redex (§4.5, §7.5). *Class:* the core idea.

I-5 · Linear like Rust, lazy clone like Haskell. *(H.46, `victor/hvm-how.md`)*

> "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*."

Mechanism: affine variables; a value with one reference can be freed or reused when consumed. *Class:* design claim.

I-6 · Cloning is layer by layer. *(H.46)*

> "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."

Caveat (ours): "essentially free" means "pay only for the layers that are demanded", not zero cost (§7.5: 11 vs 8).

I-7 · Sharing inside lambdas. *(H.46)*

> "it is capable of **sharing computations inside lambdas, which GHC can't**."

Mechanism: the showpiece's single `OP2-NUM-NUM` (§7.5). *Class:* mechanism plus a comparison with GHC's semantics.

I-8 · The bottleneck was the graph, not the idea. *(H.46)*

> "all implementations of this algorithm, until now, represented terms as graphs. This demanded a lot of pointer indirection, making it slow in practice."

Mechanism: HVM's 64-bit term words and linked variables (Part 8). *Class:* engineering claim.

I-9 · The IC in three lines. *(H.33, Interaction-Calculus README)*

> "1. Vars are affine: they can only occur up to one time."
>
> "2. Vars are global: they can occur anywhere in the program."

Mechanism: global scope is what DUP-LAM's escaping variable needs (§5.3). *Class:* definition.

I-10 · What global lambdas buy. *(H.33)*

> "Global lambdas allow the IC to express concepts that aren't possible on the traditional λC, including continuations, linear HOAS, and mutable references."

Mechanism: HVM4's unscoped binders (tag UNS, the `unscoped_*` tests). *Class:* expressiveness claim.

I-11 · Affinity and parallelism. *(H.33)*

> "Finally, being fully affine makes its garbage collector very efficient, and greatly simplifies parallelism."

I-12 · The day optimal reduction was miraculous. *(H.41, `victor/hoc-history.md`)*

> "It was miraculously fast. Impossibly so."

Context: his first implementation of Lamping's algorithm (Optlam) and the Stack Overflow question that followed. *Class:* anecdote.

I-13 · The two reasons. *(H.41)*

> "1. A *negative result* that was misinterpreted by researchers back then"
>
> "2. The theoretical optimality didn't translate to real-world performance"

Mechanism: Part 9.3. *Class:* historical diagnosis (his reading of the literature).

I-14 · Optimal, in Lévy's sense, and its gap. *(H.53, `victor/hvm-unordered-sup.md`)*

> "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."

> "The culprit is the accumulation of "redundant" control nodes or duplication nodes in some cases."

Class: theorem-backed claim, plus an acknowledged limitation.

I-15 · Search with superpositions. *(H.50, `victor/hvm-search1.md`)*

> "On HVM, instead of trying each candidate separately, we can create a superposition of all possible functions."

Mechanism: §6.4. *Class:* method; the synthesizer built on it (SupGen) is not public.

11.1 Where to read Taelin first

Source	Where	Why
Interaction-Calculus README	`github.com/VictorTaelin/Interaction-Calculus` · *H.33*	the calculus, its rules, worked examples
"Towards an optimal computer"	*H.42*	the manifesto: why interaction combinators
"HVM: how it works"	*H.46*	memory format, lazy clone, sharing inside lambdas
"Optimal evaluation in 10 minutes"	*H.43*	the incremental-copy argument from first principles

Source	Where	Why
HOC history	*H.41*	Optlam, Lamping's two halves, why the field stalled
"Truly optimal evaluation with unordered superpositions"	*H.53*	the limits of Lévy optimality
Program search posts	*H.50*, *H.51*	superposition as search
HVM4 repository	`github.com/HigherOrderCO/HVM4` · *L.1–L.8*	the current machine and its docs
Bend	`github.com/bendlang/bend` · *H.1–H.4*	the language today

Exercises — Part 11

E11.1 ★ Which of I-1…I-15 is a theorem, which an anecdote, which a design claim? Name one of each.

▾ SOLUTION

Theorem-backed: I-14 (Lévy-optimality for admissible terms). Anecdote: I-12. Design claim: I-5 or I-8.

E11.2 ★★ I-6 says cloning is "essentially free". Give a measured counterexample from this book and state the correct reading.

▾ SOLUTION

§7.5: sharing one partial application costs 11 interactions, calling twice costs 8. Correct reading: cloning costs work proportional to the layers actually demanded, never a full upfront copy; on large shared work this is a large saving, on tiny examples it is overhead.

E11.3 ★★ Connect I-9's "Vars are global" to a specific line of an HVM4 trace in this book.

▾ SOLUTION

§4.4/§5.3: after DUP-LAM, `λa.A₀ … !A&A=@{&A{a,b}}` — the variables `a` and `b` of the two new lambdas appear *outside* their lambdas, inside a duplication elsewhere in the term. Lexical scoping cannot express that intermediate state; global variables can.

<hr>

12 · Drills and capstones

12.1 Drill set (short, mixed)

Answers are one or two lines; check yourself before opening.

D1 ★ Which rule fires for `@` meets λ ?

▾ **ANSWER**

APP-LAM, a γγ annihilation: var↔arg, body↔result.

D2 ★ Which rule grows a net?

▾ **ANSWER**

Commutation (γδ, and δδ with different labels): 2 cells → 4.

D3 ★ A λ binds a variable used 5 times. How many duplicators?

▾ **ANSWER**

4 (n − 1).

D4 ★ Which HVM4 term erases? `*` , `&{}` or `ε` ?

▾ **ANSWER**

`&{}` . `*` is the wildcard ANY; `ε` is the net-level name.

D5 ★ What does `#N` after a collapsed answer mean?

▾ **ANSWER**

The running interaction count when that answer was printed (`src/hvm.c:6193`).

D6 ★ `!x&A = &A{a,b}` — project or copy?

▾ **ANSWER**

Project (same label): x_0 = a, x_1 = b.

D7 ★ `!x&A = &B{a,b}` — project or copy?

▾ **ANSWER**

Copy the choice (different labels): x_0 = x_1 = &B{…} with a and b duplicated.

D8 ★ Build command at HVM4 HEAD?

▾ **ANSWER**

```
clang -O2 -o src/hvm src/hvm.c .
```

D9 ★★ Why can `-D` traces be compared line by line with the net rules?

▾ **ANSWER**

Each traced line is one interaction, named by `ITRS_INC(name)`; the core names map to the net rules (APP-LAM = γγ, DUP-SUP = δδ, DUP-LAM/APP-SUP = γδ).

D10 ★★ In 64-bit HVM4 words, where is a duplication's label?

▾ **ANSWER**

EXT (bits 32–55).

D11 ★★ `&A{1,2} + &A{10,20} + &B{100,200}` : how many worlds?

▾ **ANSWER**

4: the two A-superpositions pair (2 worlds), the B one crosses them (×2).

D12 ★★ What is the label of `λx&.` 's duplication?

▾ **ANSWER**

A fresh label chosen by the parser, different from every user label (§5.6).

D13 ★★ (Debug) `@main = !x&A = 5; x₀ + x₀` . What does HVM4 say, and why?

▾ **ANSWER**

A parse error: `x₀` used twice. Each side of a duplication is affine; duplicate again or use `!&x` .

D14 ★★ Is `fib(34)` in 79.5 M interactions a failure of optimal reduction?

▾ **ANSWER**

No: the recursive calls are independent work; optimality only avoids duplicating the *same* work.

D15 ★★ What does "admissible" protect in Lévy-optimality claims for HVM?

▾ **ANSWER**

That the label discipline is correct for the term (no same-label fans meeting without being partners).

D16 ★★ Is BendRT an interaction-net reducer?

▾ **ANSWER**

No. It keeps affinity, refcounted sharing and fork parallelism, but not cells/ports/rewrites.

D17 ★★ HVM5 v2: fork = ?, test = ?, merge = ?

▾ **ANSWER**

Fork = one DUP of state; test = SUP of books (`&T{old,new}`); merge = 3-way merge of definition hashes + CAS epoch swap.

D18 ★★★ Why must the superposed variable in DUP-LAM carry the duplication's own label?

▾ **ANSWER**

So that the later duplication of the shared body *projects* it (same label), giving each copy its own variable.

D19 ★★★ Why does the same-label accident not raise an error?

▾ **ANSWER**

The result is a valid IC term; no rule is missing. It is a semantic mismatch with the intended λ-term, not a runtime fault.

D20 ★★★ Give the one-sentence reason HVM is a term machine rather than a graph machine (Part 8B.4).

▾ **ANSWER**

Graph implementations of optimal reduction were slow because of pointer indirection (I-8); 64-bit term words with linked variables implement the same interactions compactly.

12.2 Capstones

C1 ★★ — Hand reduction to the end. Translate `(λf. λx. f (f x)) (λy. y) z` to a net (z free) and reduce it by hand to normal form, drawing every step. State the count and compare with HVM4 on `@main = (λf&. λx. f₀(f₁(x)))(λy.y)` applied to a free-standing argument of your choice.

▾ **SOLUTION OUTLINE**

1. Net: Church 2 (§E4.3, 5 cells) applied to the identity, then to z.
2. β (f ↔ id); the δ on f meets the identity's λ: DUP-LAM + δδ on the loop (2) → two identities.
3. β (x ↔ z); the inner application `id z` → β; the outer `id (z)` → β.
4. Count: 1 + 2 + 1 + 1 + 1 = **6** net interactions; result `z` .
5. On HVM4, `@main = (λf&. λx. f₀(f₁(x)))(λy.y)(7)` prints `7` with **6** interactions; the `-D` rule sequence is APP-LAM, APP-LAM, DUP-LAM, APP-LAM, DUP-SUP, APP-LAM. The machine interleaves the steps lazily (it applies the second argument before copying `f`), but the multiset of rules matches the hand reduction: 4 β, 1 DUP-LAM, 1 DUP-SUP.

C2 ★★ — Commutation by hand, then by machine. Draw a δ meeting the net of `λx. λy. x`. Reduce until no active pair remains. Then run `@main = !f&A = λx.λy.x; [f₀, f₁]` with `-D` and match every line to your drawing.

▾ **SOLUTION OUTLINE**

1. DUP-LAM copies the outer λ; its variable x becomes `&A{x0,x1}` and the body `λy.x` sits behind `!G&A`.
2. DUP-LAM again for the inner λ (the copier meets `λy`); y becomes a superposition too.
3. The copier reaches `&A{x0,x1}`: DUP-SUP, same label → projection.
4. The unused y is bound to an eraser in the net picture, which must eventually meet y's superposition.
5. HVM4 HEAD: `[λa.λb.a,λa.λb.a]` in **3** interactions, trace DUP-LAM, DUP-LAM, DUP-SUP. It records no erasure interaction for the unused y: compare that with your drawing and explain the difference (the term machine never materializes a variable nobody reads).

C3 ★★ — Labels as semantics. Write two programs that differ only in one label and produce different answers (other than c2/k2). Explain the difference with DUP-SUP.

▾ **SOLUTION OUTLINE**

1. `@main = !x&A = &A{3,4}; x₀ + x₁` → 7 (projection).
2. `@main = !x&A = &B{3,4}; x₀ + x₁` → 6, 8 (copy the choice).
3. Explanation: §5.3; the label decides whether the duplication is the pair's partner.

C4 ★★ — Search by erasure. Find all pairs `(x, y)` in 0..7 × 0..7 with `x*y == 12` using superpositions. Predict the number of answers before running.

▾ **SOLUTION OUTLINE**

1. Candidates: two 3-bit superpositions with six distinct labels (§6.6 pattern).
2. Test with `@keep([x,y], (x*y) == 12)`.
3. Prediction: (2,6), (3,4), (4,3), (6,2) → **4 answers**. HEAD prints exactly these, after 849 interactions.

C5 ★★ — Two numeral forms. Write `@double` for constructor chains and `@double'` for n-literals; show the runtime error when you swap their inputs.

▾ **SOLUTION OUTLINE**

1. `@double = λ{ #Z: #Z{} ; #S: λp. #S{#S{@double(p)}} }`.
2. The n-literal version uses `0n: / 1n+:` patterns (see `docs/hvm/syntax.md`).
3. Swapping inputs gives `RUNTIME_ERROR: cannot apply a number` (n-literal into a `#Z/#S` match).

C6 ★★★ — Sharing crossover. Build the §7.7 experiment: a shared partial application versus two calls, with a shared computation of tunable cost. Find the cost at which sharing starts to win, and plot interactions against cost.

▾ SOLUTION OUTLINE

1. Shared: `@f = λx.λy. !z&=x; [@work(z₀, z₁), y]` ; unshared: call `@f` twice.
2. Shared ≈ cost + overhead (≈ 3–10); unshared ≈ 2 × cost. Crossover at cost ≈ overhead.
3. Report the measured counts; the curve is the honest form of "sharing inside lambdas".

C7 ★★★ — Reading the C. Pick three sheets (one APP-*, one DUP-*, one OP2-*). For each, find the implementing `wnf_*` function in `src/hvm.c` , the `ITRS_INC` name, and a one-line program whose `-D` trace shows it.

▾ SOLUTION OUTLINE

1. `grep -n "ITRS_INC(\"APP-SUP\")" src/hvm.c` → its function (`wnf_app_sup`).
2. Program: `@main = (&A{λx.x, λy.10})(7)` (§5.3).
3. Repeat for DUP-NOD (`!x&A = [1,2]; [x₀, x₁]`) and OP2-NUM-NUM (`@main = 2 + 2`).

C8 ★★★ — Collapse order. Write a superposition of four values whose collapse order you change only by wrapping one branch in the priority-lowering `INC` form. Predict the new order using §8.6.

▾ SOLUTION OUTLINE

1. Baseline: `&A{&B{1,2},&B{3,4}}` collapses in queue order.
2. Lower one branch's priority with the INC syntax from `docs/hvm/syntax.md` ; lower-key branches are popped first, so a priority-lowered branch comes later.
3. Run both and compare the order and the `#N` counters.

C9 ★★★ — Claim audit. Take any public sentence about HVM or Bend (a README line, a post) and classify each clause with the table of §9.6. Rewrite it so every clause is true as stated.

▾ SOLUTION OUTLINE

Example: "Bend runs on interaction nets and is exponentially faster than Haskell." → (1) false for released Bend 2 (BendRT); (2) true only for chosen program families on HVM. Rewrite: "HVM, an interaction-calculus runtime, can be exponentially faster than GHC on programs that share work inside lambdas; released Bend 2 runs on a separate flat runtime."

C10 ★★★ — Design: superposed testing. A book has definitions `@a` , `@b` , `@c` ; `@b` changes. Write the superposed book and estimate the shared ratio for a request that spends 70% of its old-version time before calling `@b` .

▾ SOLUTION OUTLINE

1. Superposed book: `@a` and `@c` unchanged; `@b = &T{@b_old, @b_new}` .
2. If the old run costs W and the new one about the same, the superposed run ≈ 0.7W + 2 × 0.3W = 1.3W versus 2W separately.
3. Shared ratio ≈ 0.7W / 2W = 35% of the combined work saved. Report the measured value.

C11 ★★★ — Proof. Prove that firing the two erasures of §3.8 step 1 in either order yields the same net (a concrete instance of the diamond).

▾ SOLUTION OUTLINE

1. The two active pairs δ_1–ε_1 and δ_2–ε_2 share no cell.
2. Each erasure removes its two cells and adds two erasers on the removed δ's aux wires, which lead to γ_1/γ_2 aux ports, disjoint from the other pair.
3. After both, the same six cells remain with the same wires, whichever fired first. ∎

C12 ★★★ — Write a chapter. Choose one rule sheet from Appendix B that this book did not explain (for example `app_mat_sup`), explain it in the style of §5.3 with a tested program and its `-D` trace, and add two exercises.

▾ SOLUTION OUTLINE

1. `app_mat_sup` : a pattern match applied to a superposition distributes over both alternatives (the match analogue of APP-SUP).
2. Program: `@main = λ{#T: 1; #F: 0}(&A{#T{}, #F{}}) → 1, 0` in 8 interactions at HEAD; read its `-D` trace.
3. Exercises: predict the worlds; change the label of the argument and predict again.

<hr>

Appendix A · Reproduce everything in ten minutes

```
 mkdir -p ~/inet && cd ~/inet
git clone https://github.com/HigherOrderCO/HVM4 && (cd HVM4 && git checkout 6defdfc && clang -O2 -o src/hvm src/hvm.c)
git clone https://github.com/VictorTaelin/Interaction-Calculus && (cd Interaction-Calculus && git checkout e9f0185 &&
make)
export H=~/inet/HVM4/src/hvm IC=~/inet/Interaction-Calculus/bin/ic
(cd HVM4 && HVM_TEST_TIMEOUT_SECS=20 ./devs/test/_all_.sh | tail -1)     # All tests passed!
printf '@main = (&A{1,2} + 10)\n' > a.hvm && $H a.hvm -s -C               # 11 #3 / 12 #4 / Itrs 4
echo '(λx.x λy.y)' > a.ic && $IC run a.ic                                   # λ$a.$a · WORK: 1
```

Every program in this book can be pasted into a file and run with `$H file.hvm -s -C` (or `-D` for the trace).

<hr>

Appendix B · The 61 HVM4 interaction sheets (verbatim, HEAD `6defdfc`)

Copied from `docs/hvm/interactions/*.md`. Each sheet is a sequent: the redex above the line, the result below, the rule name on the line. `eql_nam.md` is empty upstream.

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)
```

alo_dry

```
@{s} ^(f x)
-------------- ALO-DRY
^(@{s}f @{s}x)
```

alo_dup

```
@{s} ! x &L = v; t
------------------ ALO-DUP
x' ← fresh
! x' &L = @{s} v
@{x',s} t
```

alo_lam

```
@{s} λx.f
------------ ALO-LAM
x' ← fresh
λx'.@{x',s}f
```

alo_nam

```
@{s} ^n
------- ALO-NAM
^n
```

alo_nod

```
@{s} T{a,b,...}
---------------- ALO-NOD
T{@{s}a, @{s}b, ...}
```

alo_var

```
@{s} n
------ ALO-VAR
s[n] or n when substitution missing (n is a de Bruijn level)
```

and_era

```
(&{} .&. b)
----------- AND-ERA
&{}
```

and_inc

```
(↑a & b)
--------- AND-INC
↑(a & b)
```

and_num

```
(#0 .&. b)
---------- AND-ZER
#0
(#n .&. b)   [n ≠ 0]
-------------------- AND-ONE
b
```

and_sup

```
(&L{a0,a1} .&. b)
------------------------- AND-SUP
! B &L = b
&L{(a0 .&. B₀), (a1 .&. B₁)}
```

app_ctr

```
(ctr a)
----------- APP-CTR (stuck)
^(ctr a)
```

app_dry

```
(^(f x) a)
----------- APP-DRY
^(^(f x) a)
```

app_era

```
(&{} a)
------- APP-ERA
&{}
```

app_inc

```
(↑f x)
-------- APP-INC
↑(f x)
```

app_lam

```
(λx.f a)
-------- APP-LAM
x ← a
f
```

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})
```

app_mat_num

```
(λ{#a:h; m} #a)
--------------- APP-MAT-NUM-MAT
h
(λ{#a:h; m} #b)
--------------- APP-MAT-NUM-MIS
(m #b)
```

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)}
```

app_nam

```
(name a)
--------- APP-NAM
^(name a)
```

app_sup

```
 (&L{f,g} a)
---------------- APP-SUP
! A &L = a
&L{(f A₀),(g A₁)}
```

ddu_era

```
 ! X &(&{}) = v; b
---------------- DDU-ERA
&{}
```

ddu_inc

```
 ! X &(↑x) = v; b
---------------- DDU-INC
↑(! X &(x) = v; b)
```

ddu_num

```
 ! X &(#n) = v; b
---------------- DDU-NUM
! X &n = v
b(X₀, X₁)
```

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}
```

dsu_era

```
 &(&{}){a, b}
------------ DSU-ERA
&{}
```

dsu_inc

```
 &(↑x){a, b}
------------ DSU-INC
↑(&(x){a, b})
```

dsu_num

```
&(#n){a, b}
----------- DSU-NUM
&n{a, b}
```

dsu_sup

```
&(&L{x,y}){a, b}
------------------------- DSU-SUP
! A &L = a
! B &L = b
&L{&(x){A₀,B₀}, &(y){A₁,B₁}}
```

dup_dry

```
! X &L = ^(f x)
--------------- DUP-DRY
! F &L = f
! A &L = x
X₀ ← ^(F₀ A₀)
X₁ ← ^(F₁ A₁)
```

dup_lam

```
! F &L = λx.f
---------------- DUP-LAM
F₀ ← λ$x0.G₀
F₁ ← λ$x1.G₁
x  ← &L{$x0,$x1}
! G &L = f
```

dup_nam

```
! X &L = name
------------ DUP-NAM
X₀ ← name
X₁ ← name
```

dup_nod

```
! X &L = T{a,b,...}
------------------- DUP-NOD
! A &L = a
! B &L = b
...
X₀ ← T{A₀,B₀,...}
X₁ ← T{A₁,B₁,...}
```

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}
```

eql_any

```
(* === b)
----------- EQL-ANY-L
1
(a === *)
----------- EQL-ANY-R
1
```

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
```

eql_dry

```
(^(af ax) === ^(bf bx))
---------------------- EQL-DRY
(af === bf) & (ax === bx)
```

eql_era

```
(&{} === b)
----------- EQL-ERA-L
&{}
(a === &{})
----------- EQL-ERA-R
&{}
```

eql_inc

```
(↑a === b)
----------- EQL-INC-L
↑(a === b)
(a === ↑b)
----------- EQL-INC-R
↑(a === b)
```

eql_lam

```
(λax.af === λbx.bf)
------------------- EQL-LAM
X := fresh_nam()
ax ← X
bx ← X
af === bf
```

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
```

eql_nam

```

```

eql_num

```
(#a === #b)
------------ EQL-NUM
#(a == b)
```

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)}
```

eql_use

```
(λ{af} === λ{bf})
----------------- EQL-USE
af === bf
```

mat_inc

```
(λ{...} ↑x)
------------ MAT-INC
↑(λ{...} x)
```

op2_era

```
@@opr(&{}, y)
------------- OP2-ERA
&{}
```

op2_inc

```
(↑x op y)
---------- OP2-INC-X
↑(x op y)
(#n op ↑y)
---------- OP2-INC-Y
↑(#n op y)
```

op2_num_era

```
(x op &{}) where x is NUM
-------------- OP2-NUM-ERA
&{}
```

op2_num_num

```
(#a op #b)
-------------- OP2-NUM-NUM
#(a opr b)
```

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)}
```

op2_sup

```
@@opr(&L{a,b}, y)
------------------------- OP2-SUP
! Y &L = y
&L{@@opr(a,Y₀), @@opr(b,Y₁)}
```

or_era

```
(&{} .|. b)
----------- OR-ERA
&{}
```

or_inc

```
(↑a | b)
--------- OR-INC
↑(a | b)
```

or_num

```
(#0 .|. b)
---------- OR-ZER
b
(#n .|. b)   [n ≠ 0]
-------------------- OR-ONE
#1
```

or_sup

```
(&L{a0,a1} .|. b)
------------------------- OR-SUP
! B &L = b
&L{(a0 .|. B₀), (a1 .|. B₁)}
```

uns

```
! ${f, v}; t
------------- WNF UNS
t(λy.λ$x.y, $x)
```

use_era

```
(λ{f} &{})
---------- USE-ERA
&{}
```

use_inc

```
(use ↑x)
--------- USE-INC
↑(use x)
```

use_sup

```
(λ{f} &L{a,b})
----------------- USE-SUP
! F &L = f
&L{(λ{F₀} a), (λ{F₁} b)}
```

use_val

```
(λ{f} x)
--------- USE-VAL
(f x)
```

<hr>

Appendix C · Verified-run ledger (HVM4 HEAD `6defdfc`, 2026-11-06)

`-s -C` ; Itrs = total interactions. Heap sizes and times are deliberately omitted.

§	Program (abridged)	Output	Itrs
0.5	`(&A{1,2} + 10)`	11, 12	4
0.5/7.5	showpiece `!F& = @f(2); [F₀(10), F₁(20)]`	`[[4,10],[4,20]]`	11
7.5	`[@f(2,10), @f(2,20)]`	`[[4,10],[4,20]]`	8
4.3	`(λx.x)(λy.y)`	`λa.a`	1
4.4	`!f&A = λx.x; [f₀, f₁]`	`[λa.a,λb.b]`	2
5.3	`!x&A = &A{1,2}; [x₀, x₁]`	`[1,2]`	1
5.3	`!x&A = &B{1,2}; [x₀, x₁]`	`[1,1], [2,2]`	5
5.3	`(&A{λx.x, λy.10})(7)`	7, 10	4
5.3	`(λx.7)(&A{1,2})`	7	1
5.3	`!x&A = &{}; x₀`	(nothing)	1
5.4	`!f&A = λx.λy.[x,y]; [f₀(1,2), f₁(3,4)]`	`[[1,2],[3,4]]`	11
5.5	`&A{1,2} + &A{10,20}`	11, 22	4
5.5	`&A{1,2} + &B{10,20}`	11, 21, 12, 22	10
5.6	`@c2(@c2)`	`λa.λb.a`	9
5.6	`@c2(@k2)`	`λa.λb.a(a(a(a(b))))`	20
5.6	`(λx&. [x₀, x₁])(&A{1,2})`	`[1,1], [2,2]`	6
E5.1	`!x&A = &A{3,4}; x₀ + x₁`	7	2
E5.2	`!x&A = &B{3,4}; x₀ + x₁`	6, 8	7
E5.5	`(λx&. x₀ + x₁)(&A{1,2})`	2, 4	8
5.7	`[*, 1]`	`[*,1]`	0
6.1	`λx. (x + &A{1,2})`	`λa.(a + 1), λa.(a + 2)`	1
6.2	`(λa. &L!{(a + 1); (a * 10)})(5)`	6, 50	4
6.3	`@bits(3)`	8 strings	40
6.4	search `x*x == x+x` over 0..7	0, 2	167

§	Program (abridged)	Output	Itrs
E6.1	`&A{1,2} * &B{3,4}`	3, 4, 6, 8	10
E6.2	search `x*x + 1 == 26`	5	134
E6.6	pairs with `x + y == 3`	[0,3], [1,2], [2,1], [3,0]	258
6.5	`X + 2 = 4` over all naturals	`#S{#S{#Z{}}}` (at #306)	536
7.1	`@cnot(@cnot(@ctru))`	`λa.λb.a`	6
7.2	`@cadd(@c1,@c2)` / `(@c2,@c2)` / `(@c4,@c4)`	3 / 4 / 8	8 / 9 / 25
7.2	`@cmul(@c2,@k2)` / `(@c2,@c2)`	4 / 4	14 / 14
7.3	`@fac(3)` / `@fib(5)` / `@sum(4)` (chains)	6 / 5 / 10	109 / 274 / 55
7.4	`@map(λx.(x*2), [1,2,3])`	`[2,4,6]`	35
7.6	`fib_small` at 34 (machine ints)	5702887	79,522,600
7.1/E7.1	`(λ&x. x(x))(λy.y)`	`λa.a`	4
8.9	dup destructuring / auto-fork examples	see §8.9	1 / 1 / 5 / 4
E8.6	`!x&A = 5; [x₀, x₁]`	`[5,5]`	1

IC reference (`e9f0185`): `(λx.x λy.y)` → `λ$a.$a` , `WORK: 1` . Test suite at HEAD: 217 tests, all pass with `HVM_TEST_TIMEOUT_SECS=20` (with the default 2 s, `fib_small` alone can time out).

Appendix D · Glossary

Term	Meaning
active pair	two principal ports joined by a wire; the only place a rule fires
admissible	a term on which the label discipline is correct (no false same-label meetings)
ALO	HVM4 tag for a lazily allocated book term (`@{…}` in traces)
annihilation	rule that deletes both cells of an active pair and joins aux1–aux1, aux2–aux2
auto-dup	`λ&x.` / `!&x` / `λx&.` : the parser inserts duplicators, fresh labels where needed
book	the static, immutable definitions of a program; in HVM5, a content-addressed `Name → DefHash` map
BendRT	released Bend 2's runtime; a flat evaluator, not a net reducer
CNF / collapse	lifting superpositions to the top and enumerating worlds (`-C`)
commutation	rule where two cells copy each other (2 → 4)
DP0 / DP1	HVM4 tags for the two sides x_0 , x_1 of a duplication
DUP	duplication `!x&L = v; t` : one value, two locations
erasure	ε meeting a cell: delete it, send two erasers outward
EXT	24-bit field of an HVM4 term word holding labels, names, levels
fresh label	a label generated to differ from all others
γ / δ / ε	constructor / duplicator / eraser cells of the interaction combinators
IC	the Interaction Calculus: λ + labeled DUP/SUP, affine global variables
Itrs	HVM4's interaction counter (`ITRS_INC`)
label	the name of a DUP/SUP fan; equal labels project, different labels commute
Lévy-optimal	reduces each redex family once; measured in β-steps
oracle	Lamping's bookkeeping deciding which fans are partners; labels are a static version
principal port	the single port of a cell where it can interact
SNF / WNF	strong / weak normal form
SUB	high bit of the HVM4 tag byte: "this slot holds a substitution"
SUP	superposition `&L{a,b}` : two values, one location

Term	Meaning
superposed testing	HVM5 v2: running old and new definitions in one reduction via `&T{old,new}`

Appendix E · Map to the compendium

Topic	Compendium ids
Bend 2 tutorial (language, BendRT, proofs)	A.1–A.17
Site notes on nets and the four interactions	Part B
Bend 2 skills, HVM5-IC skill	Parts C, D
Knowledge base on Bend2/HVM5/INET	Part F
HVM5 v1 and v2 specifications (README G.51, REVIEW G.52, code-as-data G.57, v1 README G.2, model↔machine G.5)	Part G
Taelin and HigherOrderCO primary sources (H.33 IC README, H.41–H.55 posts)	Part H
HVM4 March 2026 snapshot (docs and tests)	Part I
HVM4 at HEAD `6defdfc` (README L.1, AGENTS L.2, primer L.3, theory L.4, core L.5, memory L.6, syntax L.7, collapser L.8, sheets L.9–L.69)	Part L
Papers: HVM2 (M.1), BendRT (M.2), BendTT (M.3); Mazza (M.4), Laretto (M.5), Borner (M.6); Taelin's HN comment (M.7)	Part M
Audit ledger and v1 → v2 changes	Appendix R

Appendix F · Further reading (links checked 2026-11-06)

All returned HTTP 200 on 2026-11-06 unless marked.

Source	Link	Why
Interaction-Calculus reference	https://github.com/VictorTaelin/Interaction-Calculus (https://github.com/VictorTaelin/Interaction-Calculus)	the calculus and its C reference reducer
HVM4	https://github.com/HigherOrderCO/HVM4 (https://github.com/HigherOrderCO/HVM4)	this book's machine (`src/hvm.c` , `docs/`)
HVM2 paper	https://github.com/HigherOrderCO/HVM2/blob/main/paper/HVM2.pdf (https://github.com/HigherOrderCO/HVM2/blob/main/paper/HVM2.pdf)	the parallel interaction-combinator runtime and its limitations (compendium M.1)
HVM1 "how it works"	https://github.com/HigherOrderCO/HVM1/blob/master/guide/HOW.md (https://github.com/HigherOrderCO/HVM1/blob/master/guide/HOW.md)	lazy clones, memory format (compendium H.46)
HVM3	https://github.com/HigherOrderCO/HVM3 (https://github.com/HigherOrderCO/HVM3)	the lazy-with-labels consolidation
Symmetric Interaction Calculus	https://github.com/VictorTaelin/Symmetric-Interaction-Calculus (https://github.com/VictorTaelin/Symmetric-Interaction-Calculus)	the memory format behind HVM1
abstract-algorithm	https://github.com/VictorTaelin/abstract-algorithm (https://github.com/VictorTaelin/abstract-algorithm)	Taelin's implementation of Lamping's abstract algorithm
optlam	https://github.com/VictorTaelin/optlam (https://github.com/VictorTaelin/optlam)	his first optimal evaluator (compendium H.41)
Interaction Calculus of Constructions	https://github.com/victortaelin/interaction-calculus-of-constructions (https://github.com/victortaelin/interaction-calculus-of-constructions)	types on top of the IC
"Supercompilation for free"	https://github.com/VictorTaelin/articles/tree/master/0004-supercompilation-for-free (https://github.com/VictorTaelin/articles/tree/master/0004-supercompilation-for-free)	an article on what optimal evaluation does to programs
Kindelia whitepaper	https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md (https://github.com/kindelia/Kindelia/blob/master/WHITEPAPER.md)	the earlier vision of an IC-based network

Source	Link	Why
BendRT / BendTT papers	https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendRT.pdf), https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf (https://github.com/bendlang/bend/blob/main/paper/BendTT.pdf)	the Bend 2 runtime (no nets) and type theory
Bend	https://github.com/bendlang/bend (https://github.com/bendlang/bend)	the language
Mazza, *A Denotational Semantics for the Symmetric Interaction Combinators* (MSCS 2007)	https://www-lipn.univ-paris13.fr/~mazza/papers/CombSem-MSCS.pdf (https://www-lipn.univ-paris13.fr/~mazza/papers/CombSem-MSCS.pdf)	the symmetric combinators, their difference from Lafont's, universality for polarized systems, vicious circles (compendium M.4)
Laretto, *The Complexity of Optimal Reduction and Sharing Graphs* (seminar slides, Pisa 2021)	https://iwilare.com/the-complexity-of-optimal-reduction-and-sharing-graphs.pdf (https://iwilare.com/the-complexity-of-optimal-reduction-and-sharing-graphs.pdf)	Lévy optimality, bookkeeping, Asperti–Mairson, Asperti–Coppola–Martini, BOHM benchmarks (M.5)
Asperti, *Linear Logic, Comonads and Optimal Reductions* (1993)	https://www.cs.unibo.it/~asperti/PAPERS/10.1.1.54.9299.pdf (https://www.cs.unibo.it/~asperti/PAPERS/10.1.1.54.9299.pdf)	the categorical reading of Lamping's brackets (not extracted: the PDF's text layer is damaged)
Borner, *Optimal Interaction with the Real World* (bachelor thesis, Tübingen 2025)	https://raw.githubusercontent.com/marvinborner/optimal-effects/refs/heads/bachelor/thesis.pdf (https://raw.githubusercontent.com/marvinborner/optimal-effects/refs/heads/bachelor/thesis.pdf)	effects in optimal interaction nets; HVM2's IO loop (M.6)
Taelin on how HVM was designed (Hacker News, 2022)	https://news.ycombinator.com/item?id=30227823 (https://news.ycombinator.com/item?id=30227823)	"learn the abstract algorithm … favor rewrite equations over case-trees" (M.7, coursebook §8B.0)
"Truly Optimal Evaluation with Unordered Superpositions" (Taelin, gist)	https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a (https://gist.github.com/VictorTaelin/93c327e5b4e752b744d7798687977f8a)	the quadratic slowdown, safe nodes, unordered DUPs (compendium H.53, coursebook §9.4)
HVM3's IC description	https://github.com/HigherOrderCO/HVM3/blob/main/IC.md (https://github.com/HigherOrderCO/HVM3/blob/main/IC.md)	the Interaction Calculus as stated for HVM3
GHC slowdown issue cited in H.53	https://github.com/HigherOrderCO/HVM/issues/60 (https://github.com/HigherOrderCO/HVM/issues/60)	**404** on 2026-11-06 (repository reorganized); the essay H.53 summarizes it

Classic papers to read next (not fetched here): Lafont, *Interaction Nets* (POPL 1990) and *Interaction Combinators* (Information and Computation, 1997); Lamping, *An algorithm for optimal lambda calculus reduction* (POPL 1990); Asperti and Guerrini, *The Optimal Implementation of Functional Programming Languages* (Cambridge, 1998); Asperti and Mairson, *Parallel beta reduction is not elementary recursive* (1998); Mazza, work on the symmetric interaction combinators.