Boundary: A Four-Agent Interaction-Style Rewriting Kernel with a Lean 4 Reference Semantics

Richard Goodman
Institute for Applied Ontological Mathematics
contact@ontologicalmathematics.org

April 2026

Abstract

We present *Boundary* (∂), a small interaction-style graph-rewriting language built from exactly four agent kinds and five reduction rules. The kernel inherits the structural discipline of Lafont's interaction combinators (principal-port active pairs, one rule per unordered agent pair, local rewrites) while introducing generation-bearing endpoint identity, a total dispatch table over all ten unordered active-pair combinations, and a locality-first staged parallelism model. We deliver a Lean 4 reference semantics (707 lines of kernel code, zero `sorry`) in which dispatch determinism, mode preservation, and interface-arity laws are proved, together with a detailed specification for a Rust executable runtime connected to the reference by differential testing against a canonically deterministic reference mode. The intended denotational reading of the four agents is bi-Heyting (negation, conjunction, co-Heyting subtraction, nucleus/boundary verification). The bi-Heyting algebraic substrate for this reading is largely complete: five boundary-operator properties are proved via Mathlib, the nucleus/re-entry theorem is proved, and the NOTW term algebra on an ICC carrier with a non-trivial three-element witness is proved. The bridging theorem connecting the operational kernel to this substrate is an explicitly tracked conjecture, currently blocked after two compatible-closure semantics were formally refuted by machine-checked counterexamples. We describe a methodological discipline of eight named anti-hack gates that prevent common failure modes in interaction-net implementations, and we argue that this discipline, together with the Lean/Rust differential bridge, constitutes a reusable methodology for proof-assistant-first interaction-net design.

1 Introduction

Interaction nets, introduced by Lafont [1] and refined in the interaction-combinator framework [2], provide a model of computation in which reduction proceeds by contracting *active pairs*: two agents whose distinguished *principal ports* are connected. The discipline is attractive because reduction is inherently local (only the two agents and their immediate neighbours participate in a rewrite step), the dispatch table over unordered agent pairs can be made total, and the one-rule-per-pair constraint eliminates nondeterministic critical pairs.

Despite these structural advantages, practical interaction-net implementations face recurring engineering difficulties that the theoretical literature does not address directly. Arena-based runtimes that reuse agent slots must cope with stale tokens: queued references to agents that have been freed and reallocated. Parallel schedulers must guarantee that concurrently selected redexes do not interfere. And implementations that span both a proof assistant and a systems-language runtime must maintain semantic agreement across tracks without allowing optimiser-driven drift.

We present *Boundary* (∂), a four-agent interaction-style rewriting kernel designed to address these concerns from first principles. Boundary's contributions are:

1. A **total dispatch table** over all ten unordered combinations of four agent kinds (ν, α, ε, τ), with priority $\varepsilon > \tau > \nu > \alpha$ and exactly five named rules (Section 3).
2. **Generation-bearing endpoint identity**: every endpoint is a triple $(id, gen, port)$, making stale-token invalidation an $O(1)$ check against the current generation counter (Section 2.2).
3. **Locality-first staged parallelism**: parallel reduction is permitted only by selecting batches of active pairs with disjoint touched sets, computing edit scripts in parallel without mutating the shared net, and committing in deterministic batch order (Section 5).
4. A **Lean 4 reference semantics** with zero `sorry` in kernel modules, in which dispatch determinism, mode preservation, and the open-net tensor's interface-arity law are proved (Section 6).
5. A **differential testing discipline** that bridges the Lean reference and a Rust runtime via a canonically deterministic reference mode, together with eight named **anti-hack gates** (H1–H8) that prevent specific failure modes (Section 7).
6. A **bi-Heyting algebraic substrate** that is largely proved, with the bridging theorem to the kernel tracked as an explicit conjecture after two proposed semantics were formally refuted (Section 8).

The intended denotational reading of Boundary's agents is bi-Heyting: ν models Heyting negation, α models conjunction (meet), ε models co-Heyting subtraction/erasure, and τ models the nucleus (boundary verification) operator [7, 8]. Unlike typical interaction-net systems that target λ-calculus or linear-logic denotations, Boundary targets bi-Heyting algebra: the co-Heyting boundary operator $\partial a = a \sqcap (\top \setminus a)$ [8] is the intended semantic content of the τ agent, and the non-triviality of this boundary on non-Boolean elements is the formal "Witness" that motivates the language's name.

2 Core Entities

2.1 Agent Kinds

Boundary has exactly four agent kinds, each with a fixed set of meaningful ports.

Table 1: Agent kinds, ports, and intended algebraic roles.

Symbol	Name	Ports	Intended role
ν	Nu	principal, aux1, witness	Heyting negation
α	Alpha	principal, aux1, aux2	Meet / conjunction
ε	Eps	principal	Erasure / subtraction
τ	Tau	principal, aux1, aux2	Nucleus / CHECK

Each port is typed on the agent kind: a port is *meaningful for* a kind if and only if it appears in that kind's required-port list. Any wire targeting a meaningless port (e.g. ν.aux2) is malformed and rejected at load time.

In Lean, the intended algebraic readings are recorded as a `ConnectiveRole` enumeration and a `connectiveFootprint` mapping, but these are design-level annotations, not theorems:

```
inductive ConnectiveRole : Type
| duplicator    -- intended role of
| tensorNode    -- intended role of
| eraser        -- intended role of
| witnessCheck -- intended role of
```

2.2 Endpoints and the Generation Discipline

A *wire* is an undirected link between exactly two *endpoints*. An endpoint is a triple:

$$Endpoint = (agent_id,\ generation,\ port)$$

The generation field is mandatory everywhere: the Lean specification, the runtime arena, the file format, and the observable-interface emission. When an arena slot is freed and reallocated, its generation counter is incremented, silently invalidating all references bearing the previous generation. Queued *pair tokens* $(a_id, a_gen, b_id, b_gen)$ carrying a stale generation are discarded without error.

Anti-hack gate **H2** ("Endpoint identity always includes generation") enforces this invariant: $(id, port)$ shortcuts are forbidden at every layer of the system.

2.3 Nets, Well-Formedness, and Active Pairs

A *net* consists of a finite arena of agents (each carrying its current generation), a list of wires, an optional source interface listing the legal free-principal endpoints at load time, and a reduction mode (bootstrap or typed).

An *active pair* is exactly two live agents whose principal ports point at each other. Auxiliary-to-principal connections are not active pairs and do not fire reductions. This invariant lets the dispatch table be defined purely on $(left.kind,\ right.kind)$.

Well-formedness is split into two predicates. *Source well-formedness* (enforced at load time) requires that free principals appear only at endpoints declared in the source interface. *Runtime well-formedness* (preserved by reduction) requires no self-loops, no multiple occupancy of a port, and liveness of both endpoints on every wire:

```
def RuntimeWellFormed (net : Net) : Prop :=
  (∀ wire ∈net.wires, wire.left ≠wire.right) ∧
  (∀ wire ∈net.wires, net.endpointLive wire.left) ∧
  (∀ wire ∈net.wires, net.endpointLive wire.right) ∧
  (∀ ep, List.countP (fun wire →wire.usesEndpoint ep)
        net.wires ≤1)
```

2.4 Reduction Modes

The kernel supports two modes. In **bootstrap mode**, τ acts as a pass-through verifier: it always succeeds, and type-side payloads on τ.aux2 are erased operationally. In **typed mode**, τ invokes an abstract `boundaryMatch` callback; on failure, reduction is blocked with an explicit `CheckFailed` result and the net is not mutated. The Lean encoding makes this callback a parameter defaulting to $\lambda\ _\ _ \Rightarrow true$:

```
abbrev BoundaryMatch :=AgentRef →AgentRef →Bool

def step? (net : Net)
    (boundaryMatch : BoundaryMatch :=fun _ _ →true)
    : StepResult :=...
```

Anti-hack gate **H5** forbids implementing agents from inventing verification semantics in bootstrap mode.

3 Reduction Rules

3.1 The Total Dispatch Table

Dispatch priority is $\varepsilon > \tau > \nu > \alpha$ (rank 0 = ε fires first). Active pairs are canonically ordered by priority rank, then by (*id*, *generation*). The total dispatch table over all ten unordered combinations is:

Table 2: Total dispatch table. Rule 4b is a specialisation of Rule 4 for the $\tau \bowtie \tau$ case.

Pair	Rule	Name
$\nu \bowtie \nu$	Rule 1	Annihilate
$\nu \bowtie \alpha$	Rule 2	Commute
$\nu \bowtie \varepsilon$	Rule 3	Erase
$\nu \bowtie \tau$	Rule 4	Check
$\alpha \bowtie \alpha$	Rule 5	Associate
$\alpha \bowtie \varepsilon$	Rule 3	Erase
$\alpha \bowtie \tau$	Rule 4	Check
$\varepsilon \bowtie \varepsilon$	Rule 3	Erase
$\varepsilon \bowtie \tau$	Rule 3	Erase
$\tau \bowtie \tau$	Rule 4b	CheckTauTau

The Lean dispatch function is a single match expression, total over canonical pairs:

```
def dispatchRule (lhs rhs : AgentRef) : RuleName :=
  match lhs.kind, rhs.kind with
  | .nu, .nu →.annihilate
  | .nu, .alpha →.commute
  | .eps, _ →.erase
  | .tau, .tau →.checkTauTau
  | .tau, _ →.check
  | .alpha, .alpha →.associate
  | _, _ →.erase
```

3.2 Rewiring Helpers

Five normative helper operations mediate all graph edits:

Anti-hack gate **H3** ("No implicit principal overwrite") protects `take_over_principal`: implementations must `disconnect` first and only then reconnect. This ordering requirement applies to every rule body that rewires a principal.

3.3 Rule 1: $\nu \bowtie \nu$ (Annihilate)

Two ν agents A and B connected principal-to-principal. Let a_{aux}, b_{aux} be the partners of A.aux1, B.aux1, and a_{wit}, b_{wit} the partners of A.witness, B.witness:

1. `reconnect_optional`(a_{aux}, b_{aux}).
2. If a_{wit} exists, `erase_branch`(a_{wit}).
3. If b_{wit} exists, `erase_branch`(b_{wit}).

Table 3: Normative rewiring helpers.

Helper	Behaviour
`disconnect`(p)	If p is connected to q, clear both sides. Idempotent on free p.
`connect`(p, q)	Connect p and q symmetrically. Precondition: both free and meaningful.
`take_over_principal`(n, f)	Disconnect the redex-side aux first; if f exists, connect n to f; else leave n free.
`reconnect_optional`(l, r)	`Some, Some` $\to$ connect; any `None` $\to$ leave free; both `None` $\to$ no-op.
`erase_branch`(t)	Allocate a fresh ε and connect its principal to t. Enqueue if this creates a new active pair.

4. Delete A and B. Enqueue newly formed active pairs.

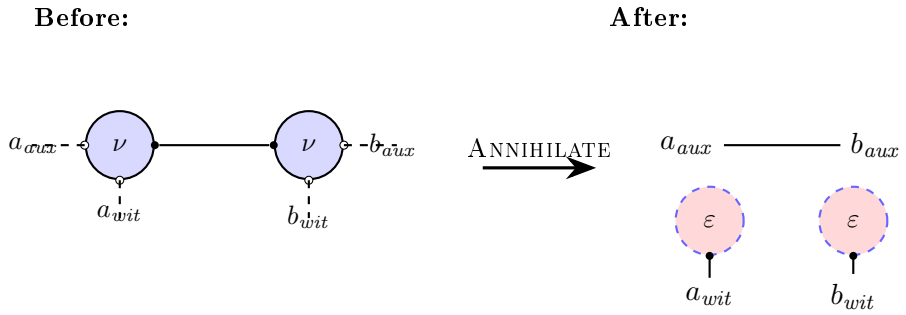


Figure 1: Rule 1: ANNIHILATE. Filled circles = principal ports; open circles = auxiliary. Data ports reconnect; witness targets are erased by fresh ε agents.

3.4 Rule 2: $\nu \bowtie \alpha$ (COMMUTE)

Allocate $N_1, N_2 : \nu$ and $X_1, X_2 : \alpha$. N_1, N_2 take over B's auxiliary partners; X_1, X_2 take over A's auxiliary and witness partners. Internal wiring: X_1.aux1 $\leftrightarrow$ N_1.aux1, X_1.aux2 $\leftrightarrow$ N_2.aux1, X_2.aux1 $\leftrightarrow$ N_1.witness, X_2.aux2 $\leftrightarrow$ N_2.witness. Delete A and B.

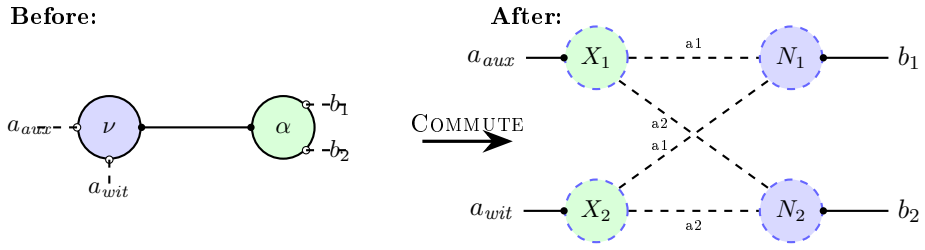


Figure 2: Rule 2: COMMUTE. ν distributes across α. X_1 collects the data path; X_2 collects the witness path.

3.5 Rules 3–5

Rule 3 (ERASE): ε dominates any opponent. For each non-principal partner of the opponent, spawn a fresh ε via `erase_branch`; delete both. Termination: the count of live non-ε agents

with non-trivial auxiliary connections strictly decreases.

Rule 4 (CHECK): τ fires against non-ε opponents. *Critical ordering*: $T.\mathsf{principal} \leftrightarrow V.\mathsf{principal}$ must be disconnected before reconnection (gate H3). In bootstrap mode, reconnect $T.\mathsf{aux1}$ to V's former principal partner, erase $T.\mathsf{aux2}$ if present, delete T. In typed mode, `boundaryMatch` gates the reduction. **Rule 4b** (CHECKTAUTAU): lower (id, gen) fires; nested checks flatten deterministically.

Rule 5 (ASSOCIATE): $\alpha \bowtie \alpha$: reconnect $a_1 \leftrightarrow b_1$, $a_2 \leftrightarrow b_2$, delete both. Local, $O(1)$, staged-parallel-compatible.

4 Worked Example: Applying NOT to TRUE

Under Boundary's encoding conventions, TRUE is an exposed ν and FALSE is an exposed ε. We trace $\neg$ TRUE in bootstrap mode.

Initial net.

```
{ "version": 2, "mode": "bootstrap",
  "interface": [[0, 0, "aux1"]],
  "agents": [
    { "id": 0, "gen": 0, "type": "nu",
      "principal": [1,0,"principal"],
      "aux1": null, "witness": null },
    { "id": 1, "gen": 0, "type": "nu",
      "principal": [0,0,"principal"],
      "aux1": [2,0,"principal"], "witness": null },
    { "id": 2, "gen": 0, "type": "eps",
      "principal": [1,0,"aux1"] } ] }
```

Agent 0 (ν, the NOT) and Agent 1 (ν, TRUE) are principal-to-principal. Agent 2 (ε) is connected to Agent 1's $\mathsf{aux1}$.

Dispatch. $\nu \bowtie \nu \mapsto$ ANNIHILATE.

Apply ANNIHILATE. `reconnect_optional`(*None*, $(2, 0, \mathsf{principal})$): *None*, *Some* case, Agent 2's principal is left free. Witnesses: both free, no-op. Delete agents 0 and 1.

Result. One live agent: Agent 2 (ε) with free principal. Observable interface: $\{(2, 0, \mathsf{principal})\}$, an exposed ε = FALSE. Correct.

5 Locality-First Staged Parallelism

For a redex, the *touched set* consists of both principal endpoints, every endpoint that may be rewired, and every opposite endpoint connected to those ports. Two redexes are independent iff their touched sets are disjoint.

Definition 5.1 (Stage). *A* stage *is a maximal set of valid pair tokens with pairwise disjoint touched sets.*

Parallel reduction: select a stage; compute each redex's edit script in parallel without mutating the shared net; commit in deterministic batch order; enqueue new pair tokens. Direct shared-mutation parallelism is forbidden (gate H7).

The Lean encoding makes the touched-set and stage notions executable:

```
def pairTouchIds (net : Net) (pair : ActivePair)
    : List Nat :=
  ([pair.left.id, pair.right.id] ++
     neighborIdsForRef net pair.left ++
     neighborIdsForRef net pair.right).eraseDups

def pairDisjoint (net : Net) (lhs rhs : ActivePair)
    : Bool :=
  !((pairTouchIds net lhs).any
    fun id →(pairTouchIds net rhs).contains id)
```

The staged parallel reducer (`Parallel.lean`, 111 lines) is executable inside Lean as well as in Rust, providing the foundation for the differential test.

6 The Lean 4 Reference Semantics

6.1 Module Structure

The Lean reference ships as 707 lines of kernel code and 384 lines of Core stable-surface code. Additionally: 2 052 lines for the surface language, 441 lines for the flat/arena compilation form, and 4 345 lines for research-track modules. Total: 8 361 lines across 39 files.

6.2 Proved Properties

Table 4: Proof status of the Lean reference semantics.

Property	**Status**
Dispatch determinism (`step_deterministic`)	Proved
Observable interface = free-principal frontier	Proved
Source lowering exports only principals	Proved
Open-net tensor arity law	Proved
Mode preservation under all helpers	Proved
Dispatch sub-cases (`dispatch_eps_left`, etc.)	Proved
`stepPreservesRuntimeWellFormed`	**Deferred**
`attachAlphaPreservesSourceWellFormed`	**Deferred**
`compiledLoweringPreservesPrincipalInterface`	**Deferred**

On the deferral of well-formedness preservation. The theorem `stepPreservesRuntimeWellFormed` is the highest-priority outstanding kernel theorem. The individual helpers each preserve mode and reciprocity independently; the difficulty lies in composing these local guarantees across a full rule body that performs multiple helpers in sequence. The project records this deferral explicitly in `Core/AdequacyPlan.lean`, a module whose sole purpose is to maintain an honest ledger of what is proved, what is algebraically specified, and what remains to be discharged. Only mode preservation is proved in `Preservation.lean` (six `*_preserves_mode` lemmas), which is a weaker invariant than runtime well-formedness preservation.

6.3 The Open-Net Algebra

The `CoreAlgebra` typeclass abstracts the kernel away from any particular carrier:

```
structure CoreAlgebra (carrier : Type) where
  interface : carrier →Interface
  tensor    : carrier →carrier →carrier
  cut?      : carrier →carrier →Option carrier
  weaken?   : carrier →Option carrier
```

The proved law: $|tensor(l, r).interface| = |l.exports| + |r.exports| + 1$.

7 Methodology: Differential Testing and Anti-Hack Gates

7.1 The Differential Discipline

The Lean reference and the Rust runtime are bridged by *differential testing* against a canonically deterministic *reference mode*: no slot reuse, deterministic pair-selection order, canonical emission. Comparing Lean against the optimised runtime is forbidden by gate **H4**.

7.2 Anti-Hack Gates

Table 5: Anti-hack gates H1–H8.

Gate	Forbids	Enforcement
H1	Semantic shortcuts	No abstract evaluators bypassing graph edits
H2	Generation-free endpoints	Parser rejects $(id, port)$ without generation
H3	Implicit principal overwrite	Disconnect-before-reconnect tests
H4	Optimiser in reference mode	No slot reuse or hash-map iteration
H5	Bootstrap typed semantics	τ pass-through only
H6	Kernel growth for surface	No new redex shapes [4]
H7	Shared-mutation parallelism	Staged by disjoint touched sets [3]
H8	Wishful canonical emission	Structure comparison, not isomorphism

7.3 Acceptance Gates and Failure Recovery

Every milestone must satisfy six gates: (1) Rust tests pass; (2) Lean kernel has zero `sorry`; (3) parser rejects malformed nets; (4) deterministic on the chosen redex; (5) stale-token tests pass; (6) differential tests pass.

The specification ships a failure-recovery table prescribing corrective actions for each obstruction type: parser accepts malformed input (add a failing test first), Lean proof does not close (shrink to minimal helper lemma), differential test disagrees (freeze optimisation, bisect in reference mode), parallel tests fail (blame touched-set computation before rule semantics).

8 The Bi-Heyting Algebraic Bridge

The algebraic content of Boundary is organised in three layers: a *substrate* of abstract bi-Heyting theorems (proved), a *carrier* of concrete normal forms with non-trivial witness (proved), and a *bridge* connecting the operational interaction-net semantics to that carrier (blocked, with two formal refutations). This section describes all three layers.

8.1 Layer 1: The Bi-Heyting Substrate (Proved)

`Numbers/Surreal/BoundaryLogic.lean` (336 lines) packages Mathlib's `Coheyting.boundary` into a `BoundaryOps` structure and proves five boundary-operator properties for any `BiheytingAlgebra` α:

1. $\partial \bot = \bot$ (`canonical_boundary_bot`)
2. $\partial \top = \bot$ (`canonical_boundary_top`)
3. $\partial(\partial a) = \partial a$ (`canonical_boundary_idem`)
4. $\partial(a \sqcap b) \leq \partial a \sqcup \partial b$ (`canonical_boundary_inf`, the Leibniz rule [8])
5. $\partial(a \sqcup b) \leq \partial a \sqcup \partial b$ (`canonical_boundary_sup_le`)

These correspond to Phase A.2 of the project's algebraic-expansion plan, which is therefore essentially complete via Mathlib plus this wrapper.

`LoF/BoundaryHeyting.lean` (52 lines) proves the nucleus fixed-point theorem [7]: for any `Reentry` operator R, its fixed-point set Ω_R carries a Heyting algebra structure, with a distinguished boundary element `R.eulerBoundary` that is least among boundary candidates:

```
instance (R : Reentry ) :
    HeytingAlgebra (R.Omega) :=inferInstance

theorem boundary_isLeast (R : Reentry ) :
    IsLeast (R.boundaryCandidates) (boundary R)

theorem boundary_modusPonens (R : Reentry )
    (a b : R.Omega) : a ⊓(a ⇒b) ≤b
```

This discharges Step 10 of the project's 11-step ontological derivation.

8.2 Layer 2: The NOTW Term Algebra and Carrier (Proved)

The bi-Heyting bridge is being proved at the level of the Interaction Combinator Calculus (ICC), one layer below the Boundary kernel. `LoF/ICC/NOTW/Defs.lean` (84 lines) defines the NOTW term combinators:

```
def notw (a : Term) : Term :=
    .ann a (.bridge (.var 0))
def andw (a b : Term) : Term :=
    .app (.lam (.app (.lam (notw (.app (.var 1)
      (.var 0)))) b)) a
def nandw (a b : Term) : Term :=notw (andw a b)
def witness (a : Term) : Term :=andw a (notw a)
```

The definition `witness a := andw a (notw a)` is the formal content of "the Witness is the Lawvere boundary $a \sqcap (\top \setminus a)$" at the term level, where `notw` is the term-level analogue of co-Heyting complementation. These are term-level combinators, not lattice operations; the correspondence with bi-Heyting operations is what the bridge proves (or refutes).

`LoF/ICC/NOTW/Order.lean` (86 lines) defines the closed-Y-free normal-form carrier:

```
def IsNF (t : Term) : Prop :=
    Term.Closed t ∧YFree t ∧IsStuck t
def NF :={ t : Term // IsNF t }
```

and proves three concrete normal forms (`nfId` $= \lambda x.\, x$, `nfFalse` $= \lambda x.\lambda y.\, y$, `nfTrue` $= \lambda x.\lambda y.\, x$) are pairwise distinct:

```
theorem carrier_has_three :
    ∃ a b c : NF, a ≠b ∧b ≠c ∧a ≠c
```

This is not a placeholder. It is the formal content of "Plurality" (Step 1 of the ontological chain) lifted to the ICC carrier.

8.3 Layer 3: The Blocked Bridge and Its Formal Refutations

The bridging theorem is tracked as conjecture `notw_biheyting_algebra_20260407`: *the NOTW interaction rules on the NF carrier realise a `BiheytingAlgebra` with `witness` = ∂.* Its current status is **blocked**.

The blocking issue is confluence. The ICC `Step` relation is provably non-confluent on the closed Y-free fragment: the `app_bridge` and `ann_lam` rules freeze disagreements under `bridge` and `lam` constructors. Two attempted compatible-closure redesigns have each been formally refuted:

Refutation 1. `compatStep_not_preserve_closedYFree` (`LoF/ICC/Compatible.lean`): a compatible closure of `Step` under `lam` and `bridge` fails to preserve the `ClosedYFree` fragment. The counterexample is `bridge(ann(var 0, bridge(ann(var 1, var 1))))`, which is `ClosedYFree` but reaches `legacyY` under `CompatStep`.

Refutation 2. `safeCompat_closed_ann_lam_shift_peak_has_no_join` (`LoF/ICC/Compatible.lean`): a guarded `SafeCompatStep` that blocks `legacyY` synthesis. Closed-fragment preservation was proved, but two suspected `ann_lam`-under-shift branches stabilise to distinct terms, yielding an explicit no-join witness on the closed Y-free fragment.

Both refutations are persisted as Lean theorems: machine-checked counterexamples to proposed semantics. These are positive results, not failures. They narrow the space of viable confluence strategies and demonstrate the project's commitment to honest mathematical progress: when a proposed semantics turns out to be wrong, the counterexample is formalised and kept.

The successor conjecture (`notw_biheyting_algebra_parallel_20260408`, status: `partial`) returns to the Takahashi-style complete-development infrastructure in `Confluence.lean` (1 134 lines), exploring whether `Par.develop_cofinal` can be proved on a sufficiently strong development target.

8.4 The ICC Encoder: Quotient-Level Semantic Preservation

`Boundary/Research/ICCEncoding.lean` (1 914 lines) structurally encodes the Boundary kernel into the ICC term algebra. Beyond the structural surface (the `frame n` constructor with proved arity, the `tau_free` invariant), the encoder now ships a quotient-level semantic-preservation theorem.

The `recursiveAdministrativeTarget` function normalises every ICC term into a canonical administrative form. The preservation theorem:

```
theorem recursiveAdministrativeTarget_compatibleSteps
    : ∀ t : ICC.Term,
      ICC.CompatibleSteps t
        (recursiveAdministrativeTarget t)
```

states that every ICC term is connected to its encoder-chosen administrative target by multi-step compatible reduction. `ICCSemanticEquivalence.lean` lifts this to the equivalence closure:

```
abbrev AdministrativeEq : ICC.Term →ICC.Term →Prop
    := Relation.EqvGen ICC.CompatibleStep

theorem recursiveAdministrativeTarget_sameClass
    (t : ICC.Term) :
```

```
    AdministrativeEq t
      (ICCEncoding.recursiveAdministrativeTarget t)
```

and `ICCSemanticContext.lean` proves full congruence under all ICC term contexts:

```
theorem AdministrativeEq.plug {t u}
    (h : AdministrativeEq t u)
    (ctx : TermContext) :
    AdministrativeEq (ctx.plug t) (ctx.plug u)
```

So `AdministrativeEq` is a true semantic congruence, and every term lies in the same equivalence class as its administrative target. These theorems are the project's honest replacements for earlier false raw-equality claims. The remaining gap is raw equality of normal forms, which coincides with the blocked bi-Heyting bridge in Layer 3.

8.5 Summary of Algebraic Status

Table 6: Algebraic proof status. Each layer contributes to the intended bi-Heyting reading of the Boundary kernel.

Component	**Status**	**Source**
Five boundary properties	Proved	`BoundaryLogic.lean`
Nucleus fixed points are Heyting	Proved	`BoundaryHeyting.lean`
NOTW term combinators	Proved	`NOTW/Defs.lean`
Non-trivial carrier (3 elements)	Proved	`NOTW/Order.lean`
ICC encoder (structural + quotient)	Proved	`ICCEncoding.lean`
`AdministrativeEq` congruence	Proved	`ICCSemanticContext.lean`
`CompatStep` preserves fragment	**Refuted**	`Compatible.lean`
`SafeCompatStep` semi-confluence	**Refuted**	`Compatible.lean`
Bridge: `BiheytingAlgebra NF`	**Blocked**	Conjecture 20260407

9 Related Work

Table 7: Comparison of interaction-net systems.

	Lafont IC [2]	**HVM/Bend** [6]	**Huber/Yi** [5]	**Boundary** (this work)
Agent kinds	3	6	3	4
Dispatch total?	Yes (6)	Yes (21)	Yes (6)	Yes (10)
Generation disc.	No	Impl.	No	Mandatory
Parallel spec.	None	Runtime	None	Formal
Proof-asst. ref.	No	No	No	Lean 4
Differential	N/A	No	No	Lean/Rust
Anti-hack gates	N/A	No	No	H1–H8

Lafont's interaction combinators [2] fix the structural discipline that Boundary inherits. Boundary adds a fourth agent (τ) and departs from the λ-calculus reading in favour of bi-Heyting algebra.

HVM and Bend [6] implement interaction-combinator reduction on GPUs with lazy duplication and parallel fan-in/fan-out. HVM2 uses six agent kinds with a total 21-pair dispatch table. No proof-assistant reference semantics exists.

Mackie and Sato [3] show that parallel speedup depends on net structure, motivating Boundary's locality-first design.

Hassan, Jiresch, and Sato [4] implement nested pattern matching by extending the agent vocabulary; Boundary takes the opposite approach via gate H6.

Huber and Yi [5] encode interaction nets in OCaml, prioritising correctness-first implementation.

The bi-Heyting algebraic substrate draws on Lawvere's co-Heyting boundary and Leibniz rule [8], Johnstone's nucleus theorem [7], and (for the n-ary extension) the hypermatrix algebra of Zapata-Carratalá et al. [9].

10 Future Work and Honest Disclosures

The bridging theorem is blocked, not absent. The bi-Heyting algebraic substrate is largely proved (five boundary properties, the nucleus theorem, the NOTW carrier with three elements). The bridge connecting the NOTW interaction rules to a `BiheytingAlgebra` instance is conjecture `notw_biheyting_algebra_20260407`, blocked after two formal refutations. The successor conjecture explores Takahashi-style complete-development. Additional anti-hack gates H11–H15 govern this work, including H15 ("No Boolean collapse masquerading as success"): the project is not complete without a `witness_nontrivial` theorem.

The runtime track does not yet exist as a standalone repository. The Lean reference is the executable specification; the Rust runtime is specified but not shipped.

Well-formedness preservation is deferred. `stepPreservesRuntimeWellFormed` is recorded in `AdequacyPlan.lean`; only mode preservation is proved.

A hypergraph extension lifting the binary kernel to n-ary motif rewrites with polyadic CHECK [9] is documented but not implemented in Lean.

The 11-step ontological chain partially overlaps with proved infrastructure. Steps 1, 6, 7, 8 are essentially proved via Mathlib plus `BoundaryLogic.lean`; Step 10 is proved via `BoundaryHeyting.lean`; Steps 3 and 4 have a concrete carrier in `NOTW/Order.lean`. The dedicated `Boundary.Research.OntologicalProof.lean` module has not yet been written, but most of its content already lives one layer below.

Immediate priorities. Discharge `stepPreservesRuntimeWellFormed`; close `Par.develop_cofinal` on the Takahashi development target to unblock the bi-Heyting bridge; stand up the Rust runtime (Phases 0–1); assemble the ontological chain into a single `OntologicalProof.lean` re-exporting the existing proofs.

11 Conclusion

We have presented Boundary, a four-agent interaction-style rewriting kernel with a Lean 4 reference semantics, generation-bearing endpoint identity, locality-first staged parallelism, and a differential testing discipline bridging a proof assistant and a systems-language runtime. The kernel's dispatch table is total over all ten unordered agent-pair combinations, its Lean modules carry zero `sorry`, and its eight anti-hack gates codify reusable engineering discipline for interaction-net implementations.

The bi-Heyting algebraic reading is not an unsubstantiated aspiration. The substrate is proved (five boundary properties, the nucleus theorem), the carrier exists (three-element NOTW normal forms), the ICC encoder ships quotient-level semantic preservation with full congruence,

and the bridging theorem is an explicitly tracked conjecture with two machine-checked refutations of proposed semantics narrowing the viable approach space. This demonstrates that honest proof-status accounting, active conjecture tracking, and a useful executable specification can coexist productively in a single formal development.

Acknowledgments

The Lean formalisation builds on the Mathlib library. The HeytingLean monorepo draws on the work of the Lean community and the Lean 4 development team.

References

[1] Y. Lafont. Interaction nets. In *POPL*, pages 95–108, 1990.

[2] Y. Lafont. Interaction combinators. *Inf. Comput.*, 137(1):69–101, 1997. doi:10.1006/inco.1997.2643.

[3] I. Mackie and S. Sato. Parallel evaluation of interaction nets. *ECEASST*, 73, 2016. doi:10.14279/tuj.eceasst.73.1034.

[4] A. Hassan, E. Jiresch, and S. Sato. Nested pattern matching in interaction nets. *EPTCS*, 21:14–25, 2010. doi:10.4204/EPTCS.21.2.

[5] N. Huber and W. Yi. An encoding of interaction nets in OCaml. *EPTCS*, 417, 2025. doi:10.4204/EPTCS.417.1.

[6] V. Taelin and HigherOrderCO. HVM2: A parallel interaction combinator evaluator. `https://github.com/HigherOrderCO/HVM`, 2024.

[7] P. T. Johnstone. *Stone Spaces*. Cambridge University Press, 1982.

[8] F. W. Lawvere. Intrinsic co-Heyting boundaries and the Leibniz rule in certain toposes. In *Category Theory (Como, 1990)*, LNCS 1488, pp. 279–281. Springer, 1991.

[9] C. Zapata-Carratalá et al. Beyond binary: Hypermatrix algebra and irreducible arity. *arXiv:2301.07494*, 2023.