Following the Evaluator: An ASCII Language for Bounded State Transition

Flyxion
Independent Researcher

September 2026

Abstract

Programming languages usually treat punctuation as a device for disambiguating parse trees. This essay develops the opposite discipline for a small symbolic language: every primitive symbol must correspond to a distinct restriction on the transitions the evaluator may perform. The starting point is the circles-of-evaluation notation used to teach nested expressions in Lisp-family languages. Extending the circle to a bounded region within which a proposed state transition is evaluated yields an ASCII language whose layout mirrors its execution structure. The essay separates concrete syntax, abstract syntax, and operational semantics, and proves a chain of properties for a deliberately small core calculus: unique parsing with canonical round trip, determinism, traversal decomposition, boundary locality with staged effects, legality of ledger histories with prefix extension, preservation of state under refusal and fault, verified commit, a two-directional correspondence between commits and valid witness records, threaded compositionality, and monotone narrowing under restrictive enclosing boundaries. These are combined in a single soundness theorem for bounded transitions. Beneath the proof chain lies a second chain of non-collapse principles, each instantiated in the calculus: representation is not referent, proposal is not realization, evaluation is not exposure, admission is not verification, event is not witness, refusal is not absence, persistence is not truth. Novelty is claimed only for the conjunction of individually familiar components.

1 Introduction: From Notation to Motion

In most languages a parenthesis is a grouping symbol. It resolves ambiguity in a string of tokens and then disappears into the syntax tree. In Lisp-family languages the parenthesis is the visible edge of a region of computation, and the nesting of parentheses is the nesting of evaluation. Teaching notations that draw expressions as nested circles make this explicit: to evaluate the outer circle, the evaluator first enters each inner circle, reduces it to a value, and hands that value outward.

That observation raises a design question narrower than the usual concerns of syntax design.

> What would a programming language look like if its symbolic vocabulary were chosen according to restrictions on evaluator transitions rather than conventional syntactic categories?

The answer developed here is a small language over printable ASCII. Its governing constraint is exclusionary: a symbol enters the language only if it corresponds to a restriction or

operation on state transition that no other symbol already performs. The constraint prevents drift toward keyword-heavy syntax and keeps the relation between source text and execution checkable.

Every evocative sentence in the essay is meant to have a plain mathematical counterpart. Table 1 collects the principal pairs. The geometric vocabulary of entering, following, and returning is an explanatory reading of a transition relation and not part of its definition. No claim is made that computation is spatial.

Evocative statement	Formal counterpart
Follow inward, return outward	Traversal decomposition (Prop. 3) and threaded compositionality (Prop. 4)
Entering a sphere is not committing	Boundary locality (Prop. 5)
No state change does not mean no event	$\Delta\sigma = 0,\ \Delta\omega = 0$ but $\lambda \preceq \lambda'$ (Prop. 7)
Committed means verified and witnessed	Trace invariants (Props. 8 and 9)
History cannot be rewritten	Prefix extension of the ledger (Prop. 6)
Return carrying only what every boundary admits	Restrictive operators narrow monotonically (Prop. 10)
The doorway, not the body	$r \neq x$ and $\mathrm{resolve}(r, \mathcal{E}) \rightharpoonup x$ (Section 13)

Table 1: Metaphor and mathematics, paired.

The essay therefore has two spines. The first is a proof chain:

$$\text{Parsing} \Rightarrow \text{Determinism} \Rightarrow \text{Staging isolation} \Rightarrow \text{Ledger legality}$$
$$\Rightarrow \text{Refusal and fault preservation} \Rightarrow \text{Commit soundness} \Rightarrow \text{Transaction soundness}.$$

The second is a family of non-collapse principles, collected in Section 17, each of which is the statement that two objects ordinary description tends to identify are in fact distinct, and each of which is instantiated at a specific place in the first chain.

2 Three Levels: Concrete Syntax, Abstract Syntax, Semantics

Three objects are kept apart throughout:

$$\text{ASCII source} \xrightarrow{\text{parse}} \text{AST} \xrightarrow{\text{evaluate}} \text{machine transitions}.$$

The concrete syntax fixes which byte strings are programs. The abstract syntax fixes their structure. The operational semantics fixes what they do. The words “sphere,” “enter,” and “return” name readings of the third object and are not permitted to do work in the first two. A design that lets one level silently define another invites two kinds of objection: a reviewer may reject the formalization while accepting the idea, or a reader may take an illustration for a definition.

3 Design Principle: Motion Economy

The alphabet is small. Table 2 distinguishes symbols in the core calculus, for which proofs are given, from extensions specified only informally.

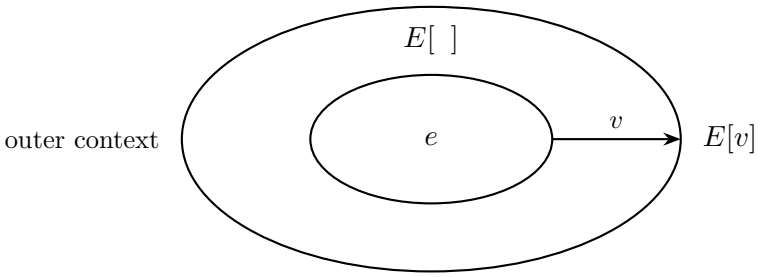

Figure 1: Nested evaluation. The inner term e is reduced under context E, and the value v is returned to the context (Prop. 3). The figure depicts the decomposition of the transition relation and does not assert that evaluation moves through space.

Symbol	Motion	Status	Semantic role
`( )`	enter, return	core	pure nested evaluation region
`<( )>`	open a bounded transition	core	delimits a frame with a staged store
`?`	admit	core	predicate on pre-state and proposal
`~`	transform	core	pure change to the staged store
`!`	verify	core	predicate on the pair pre-state, staged result
`@`	attribute	core	construct provenance for the pending record
`>`	commit	core	atomically install the staged store and record
`#`	persist	extension	flush the ledger to a durable host store
`^`	reconstruct	extension	recovery from a recorded continuation
`|`	bind	extension	environment extension

Table 2: Alphabet and status.

3.1 The admission rule and its two directions

Let Σ be the primitive symbols and M the primitive evaluator motions, with $f : \Sigma \to M$ assigning each symbol its motion. The design principle has two halves.

> **Economy.** f is injective: $f(s_1) = f(s_2) \Rightarrow s_1 = s_2$. No two primitive symbols denote the same motion.
>
> **Completeness.** For every motion m in the declared core vocabulary M_{core} there is a symbol s with $f(s) = m$.

Together they push the core toward a bijection $\Sigma_{\text{core}} \cong M_{\text{core}}$. The distinction between the two halves matters: economy prevents redundancy, completeness prevents the silent use of unnamed motions. In the core of Table 2, the seven rows marked core form the seven motions, and the correspondence is checked by inspecting the transition rules of Section 4: each rule belongs to exactly one motion, and each motion has at least one rule.

3.2 The principle applied to its own notation

The principle is useful only if it can reject the author's own choices. Three pressure tests are recorded.

Admission and verification. If `?` and `!` were predicates over the same arguments they would be redundant. They are not. Admission reads only the committed store and the proposal, so

it can be evaluated before any transformation has run. Verification reads the committed store, the staged store, and the pending effects together, so it can relate before and after. The two have different signatures, $\mathrm{Pred}(\sigma, v)$ and $\mathrm{Pred}(\sigma, \sigma^*, \pi, v)$, and hence $[\![?]\!] \neq [\![!]\!]$ for a semantic reason. Stated abstractly:

admission is prospective; verification is relational.

Before an operation one asks whether it may begin given what is known. After transformation one asks whether the proposed before-and-after relation satisfies an invariant. These are different epistemic positions.

Attribution and persistence. An earlier draft blurred `@` and `#`. They are now separated: `@` constructs the provenance that will form part of the pending record, and `#` is the flush of the ledger to a durable store. The first is a change to a frame, the second a change to the host. The ledger itself is appended at commit, discussed below.

Return and reconstruction. An earlier draft assigned `^` both to outward propagation and to reconstruction from a recorded continuation. Those are different motions and the assignment violated the principle. Return is performed by the closing delimiter `)`. The symbol `^` is reserved for reconstruction and lives in the extensions. The commit symbol `>` likewise performs one motion, the atomic publication of a transition and its record. The atomicity is the point of the motion: the correspondence theorem of Section 8 depends on the store change and the record being one step.

One token is reserved. Refusal is a result form written $\mathsf{refuse}(t, r)$ in the semantics, and the bare identifier `x` is reserved in the concrete syntax for it. Such collisions are treated as costs of the constraint and not as grounds for relaxing it.

4 The Core Calculus

4.1 Abstract syntax

Terms are

$$\begin{aligned} e &::= a \mid (e_1 \ \cdots \ e_n) \mid \langle\!\langle\, e \ ; \ c_1 \cdots c_m \,\rangle\!\rangle \\ c &::= \ ?\,p \mid \sim f \mid \,!\,q \mid \ @\,s \mid \ >\, g \end{aligned}$$

where a ranges over atoms and p, f, q, g over meta-level functions with the signatures below, drawn from a fixed signature environment:

$$p : (\sigma, v) \rightharpoonup \{\top, \bot\}, \quad f : (\sigma^*, v) \rightharpoonup (\sigma^*, \pi, v), \quad q : (\sigma, \sigma^*, \pi, v) \rightharpoonup \{\top, \bot\}, \quad g : (\sigma^*, v) \rightharpoonup v,$$

with $\rightharpoonup$ marking partial functions whose undefinedness is a fault. The head expression e of a bounded transition is the *proposal*. It is evaluated to a value before the frame opens.

The commit function g returns only a result value. It cannot supply a store. Commit installs the staged store σ^*, the store on which the verification predicates were evaluated, and never an independently supplied one. An earlier formulation in which the commit clause carried its own target state would have let a program manufacture a durable state unrelated to the verified one.

Runtime terms add the live frame

$$\mathsf{live}(t,\ v,\ \sigma_0,\ \sigma^*,\ \pi,\ A,\ \vec{c}),$$

holding transition identifier t, proposal value v, snapshot σ_0 of the committed store, staged store σ^*, pending effects π, pending provenance A, and remaining clauses $\vec{c}$, together with the two non-committing results $\mathsf{refuse}(t, r)$ and $\mathsf{fault}(t)$. Equivalently, the machine executing inside a frame can be read as the five-tuple $\langle \vec{c}, \sigma, \sigma^*, \lambda, \omega \rangle$. The frame notation packages σ^* and π with the transition that owns them.

4.2 Static semantics: well-formedness

A bounded transition is well-formed only if its clause sequence matches

$$?^* \;\; \sim^* \;\; !^+ \;\; @^+ \;\; >.$$

That is: any number of admission tests, then any number of transformations, then at least one verification, then at least one attribution, then exactly one commit, last. The ordering is a static judgment and not an accident of grammar. Section 10 shows what breaks when it is relaxed. Parsing is the partial function from ASCII source to well-formed ASTs.

4.3 Configurations and outcomes

The machine keeps four components apart,

$$C = \langle e,\ \sigma,\ \lambda,\ \omega \rangle,$$

where σ is the committed program store, $\lambda \in \mathrm{Entry}^*$ is the ledger as a finite *sequence*, and ω is the ordered trace of emitted external effects. Continuation is carried by the evaluation context in which the redex sits. Writing $\lambda \,\|\, \ell$ for appending an entry, and $\lambda \preceq \lambda'$ for the prefix relation, the ledger is append-only in the strong sense that every later ledger has every earlier ledger as a prefix. Order matters: a set of entries cannot distinguish the history $\mathsf{adm}, \mathsf{ver}, \mathsf{wit}$ from the history $\mathsf{wit}, \mathsf{adm}, \mathsf{ver}$, and sequencing is what the legality theorem of Section 7 constrains.

The outcome of a bounded transition is one of

$$\mathrm{Outcome} \in \{\mathsf{commit},\ \mathsf{refuse},\ \mathsf{fault}\}.$$

Refuse means a predicate was evaluated and returned false. *Fault* means evaluation could not be completed, for example because a predicate or transformer was undefined on its arguments, a store was unavailable, or a resource limit was reached. All three outcomes leave an entry in the ledger. The outcome of a run of a frame is the tag of the last rule applied to it.

4.4 Evaluation contexts and reduction

The places where reduction may occur are given by contexts,

$$E ::= [\,] \;\mid\; (v_1 \;\cdots\; v_{i-1}\ E\ e_{i+1} \cdots e_n) \;\mid\; \langle\!\langle\, E \;;\; \vec{c} \,\rangle\!\rangle,$$

with the contextual rule

$$\frac{\langle e, \sigma, \lambda, \omega \rangle \longrightarrow \langle e', \sigma', \lambda', \omega' \rangle}{\langle E[e], \sigma, \lambda, \omega \rangle \longrightarrow \langle E[e'], \sigma', \lambda', \omega' \rangle}$$

and left-to-right call-by-value order. A pure application whose arguments are values reduces by a primitive function table δ. A fault propagates: $E[\mathsf{fault}(t)] \longrightarrow \mathsf{fault}(t)$ for $E \neq [\,]$. A refusal is an ordinary value and propagates as data.

The rules for bounded transitions follow. Throughout, $\sigma_0, \sigma^*, \pi, A, v, t$ are the live frame's components.

Open. With t fresh (one more than the number of open entries in λ) and Q the set of verification clauses of the transition:

$$\langle \langle\!\langle v; \vec{c} \rangle\!\rangle, \sigma, \lambda, \omega \rangle \longrightarrow \langle \mathsf{live}(t, v, \sigma, \sigma, \varepsilon, \varnothing, \vec{c}),\ \sigma,\ \lambda \parallel \mathsf{open}(t, h(\sigma), Q),\ \omega \rangle.$$

Admit. Let $p(\sigma_0, v)$ be the predicate value.

$$\begin{array}{ll} p(\sigma_0, v) = \top: & \mathsf{live}(\ldots, (?p) :: \vec{c}) \longrightarrow \mathsf{live}(\ldots, \vec{c}),\quad \lambda \parallel \mathsf{adm}(t, p) \\ p(\sigma_0, v) = \bot: & \mathsf{live}(\ldots, (?p) :: \vec{c}) \longrightarrow \mathsf{refuse}(t, ?p),\quad \lambda \parallel \mathsf{ref}(t, ?p) \\ p(\sigma_0, v) \text{ undefined}: & \mathsf{live}(\ldots, (?p) :: \vec{c}) \longrightarrow \mathsf{fault}(t),\quad \lambda \parallel \mathsf{flt}(t, ?p) \end{array}$$

Transform. If $f(\sigma^*, v) = (\sigma^{*\prime}, \pi', v')$ then

$$\mathsf{live}(t, v, \sigma_0, \sigma^*, \pi, A, (\sim f) :: \vec{c}) \longrightarrow \mathsf{live}(t, v', \sigma_0, \sigma^{*\prime}, \pi \cdot \pi', A, \vec{c}),$$

with σ, λ, and ω untouched. The rule changes the staged store and the pending effects and nothing else. If f is undefined the frame faults with $\mathsf{flt}(t, \sim f)$.

Verify. With $q(\sigma_0, \sigma^*, \pi, v)$: on $\top$ the frame advances and $\lambda \parallel \mathsf{ver}(t, q)$; on $\bot$ it reduces to $\mathsf{refuse}(t, !q)$ with $\lambda \parallel \mathsf{ref}(t, !q)$; if undefined it faults with $\mathsf{flt}(t, !q)$.

Attribute. $\mathsf{live}(\ldots, A, (@s) :: \vec{c}) \longrightarrow \mathsf{live}(\ldots, A \cup \{s\}, \vec{c})$. The provenance is not yet in the ledger. It becomes part of the single record produced at commit.

Commit. Let $V = \{q \mid \mathsf{ver}(t, q) \text{ occurs in } \lambda\}$ and $w = \langle t, h(\sigma_0), h(\sigma^*), V, A \rangle$. Then

$$\langle \mathsf{live}(t, v, \sigma_0, \sigma^*, \pi, A, [>g]),\ \sigma,\ \lambda,\ \omega \rangle \longrightarrow \langle g(\sigma^*, v),\ \sigma^*,\ \lambda \parallel \mathsf{wit}(w),\ \omega \cdot \pi \rangle.$$

The commit rule is the only rule that changes σ or ω. It installs the verified staged store, emits the pending effects, and appends exactly one witness record, in one step. If g is undefined the frame faults and nothing is installed.

5 Parsing, Determinism, and Traversal

Proposition 1 (Unique parsing and canonical round trip)**.** *Let* print *be the canonical printer from well-formed ASTs to ASCII. Then* $\mathrm{parse}(\mathrm{print}(a)) = a$ *for every well-formed* a*, and* $\mathrm{print}(\mathrm{parse}(s)) = s$ *for every canonical source* s*. Consequently, for every parsable* s*,*

$$\mathrm{parse}(\mathrm{print}(\mathrm{parse}(s))) = \mathrm{parse}(s).$$

Proof. The delimiters `(` and `<(` are distinct tokens, each clause symbol is a single reserved character, and atoms are maximal runs of non-reserved, non-whitespace characters. The grammar is therefore uniquely readable by one-token lookahead, so the parse tree of a source is unique. The printer emits tokens in tree order with fixed spacing and indentation, and reading that output recovers the same tree. The second and third statements follow by composition. □

Proposition 2 (Determinism)**.** *If the meta-level functions* p, f, q, g, δ *are deterministic, then* $C \longrightarrow C_1$ *and* $C \longrightarrow C_2$ *imply* $C_1 = C_2$*.*

Proof. Every term other than a value decomposes uniquely as $E[r]$ for a redex r, by the left-to-right definition of contexts. Redexes are distinguished by their head form, and a live frame by the kind of its first remaining clause and by the value of the deciding predicate, so at most one rule applies. The fresh identifier is a function of λ. Hence the successor is unique. □

Determinism is a property of the core. Interleaving, external resources, and unavailable stores would each introduce named sources of nondeterminism, identified when they are added.

Proposition 3 (Traversal decomposition)**.** *Let e not be a value and let E be an evaluation context.*

(a) If $\langle e, \sigma, \lambda, \omega\rangle \longrightarrow^* \langle v, \sigma', \lambda', \omega'\rangle$ then $\langle E[e], \sigma, \lambda, \omega\rangle \longrightarrow^* \langle E[v], \sigma', \lambda', \omega'\rangle$.

(b) Conversely, in any run of $E[e]$ the first steps are steps of e under E, until e is a value or faults, and the run factors through $E[v]$ for that value v.

Proof. (a) follows from the contextual rule by induction on the length of the run of e. For (b), while e is not a value it contains the leftmost redex of $E[e]$, since E consists of values to the left of the hole. By Proposition 2 no other step is available. If e faults, the fault rule collapses $E[\mathsf{fault}(t)]$ to $\mathsf{fault}(t)$. Otherwise the run reaches $E[v]$. □

This is the exact content of "follow inward, return outward." The traversal of a nested term factors, as a matter of the transition relation, into an inner run followed by an outer run.

Proposition 4 (Threaded compositionality)**.** *Write $S = (\sigma, \lambda, \omega)$ and $\langle e, S\rangle \Downarrow \langle v, S'\rangle$ for $\langle e, S\rangle \longrightarrow^* \langle v, S'\rangle$. Then*

$$\frac{\langle e, S\rangle \Downarrow \langle v, S'\rangle \qquad \langle E[v], S'\rangle \Downarrow \langle r, S''\rangle}{\langle E[e], S\rangle \Downarrow \langle r, S''\rangle}.$$

Moreover, if e is pure*, meaning that its run leaves σ, λ, ω unchanged, then $E[e] \approx E[v]$, where the observer sees the final value together with σ, λ, and ω. If e is not pure, then $E[e]$ is observationally equivalent to $E[v]$ run from S', and not to $E[v]$ run from S.*

Proof. The rule is Proposition 3(a) followed by the second premise. For a pure e, $S' = S$, so the two runs coincide. For a non-pure e the two runs differ in starting store, ledger, or effect trace, and the observer can see all three. □

The qualification matters. The naive statement $C[e] \equiv C[v]$ fails as soon as e has an effect on the ledger.

6 Staging: Locality, Exposure, and Isolation

Proposition 5 (Boundary locality)**.** *Between the open step and the commit step of a bounded transition, the committed store σ and the effect trace ω are equal to their values at the open step.*

Proof. By inspection of the rules applicable to a live frame: admit, transform, verify, and attribute read σ_0 and modify only the frame components σ^*, π, A and the ledger. Only the commit rule writes σ or ω. Because the core is sequential and reduction is deterministic, no other redex runs while the frame is live. □

Entering a sphere is not equivalent to committing its intermediate states. The result depends on the design and not on the notation: it holds because $\sim$ acts on a staged store and returns proposed effects, and not because transformations are written inside delimiters. Abstractly,

$$\text{evaluation need not imply exposure:} \qquad \text{computed}(x) \not\Rightarrow \text{exposed}(x).$$

Intermediate computation can occur while durable state and external effects remain unchanged, and commit is the operation that converts a private, provisional consequence into a public, durable one. The pattern is familiar from databases, simulation, publishing, and review processes. Figure 2 shows it.

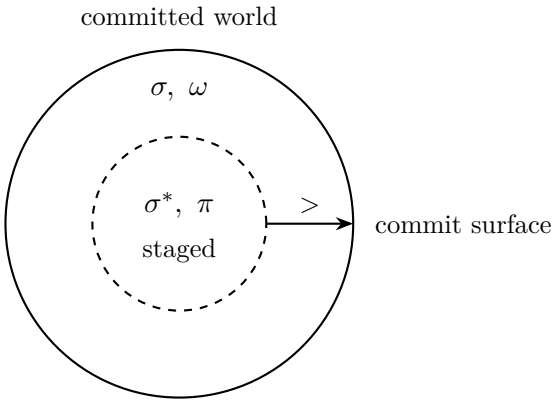

Figure 2: Boundary locality. The interior may change while the exterior does not: $\sigma^*_{t+1} \neq \sigma^*_t$ while $\sigma_{t+1} = \sigma_t$. Only crossing the commit surface permits $\sigma \leftarrow \sigma^*$ and $\omega \leftarrow \omega \cdot \pi$.

Rollback versus isolation. If a transformation could perform an irreversible external effect, for instance send a packet or actuate hardware, then

```
~ transform
! verify
> commit
```

would not provide the bounded behaviour the notation suggests, because a later refusal could not undo an earlier effect. The core rules out this case by typing: f returns *proposed* effects, and ω is written only at commit. An extension that permits effects classified as compensable, reversible, or irrevocable must say which class may occur before verification. The conservative rule is that only staged or compensable effects may precede it, and irrevocable effects may occur only through the commit clause.

Which guarantees are claimed. The word "transaction" brings database expectations. The core provides atomicity of the change to (σ, ω), in the sense that both change together at one step or not at all, and a form of consistency, in the sense that verification predicates hold on the staged result at commit. Isolation is trivial in a sequential core and would need a validation step under concurrency. Durability is not provided by the core and belongs to the flush motion `#`. For these reasons the essay uses *bounded transition* and reserves *transaction* for statements limited to these properties.

7 Ledger Legality, Refusal, and Fault

Prefix extension says only that history is not rewritten. A stronger property is that only legal histories can be written at all. Each bounded transition, viewed through the ledger, follows a small protocol, shown in Figure 3.

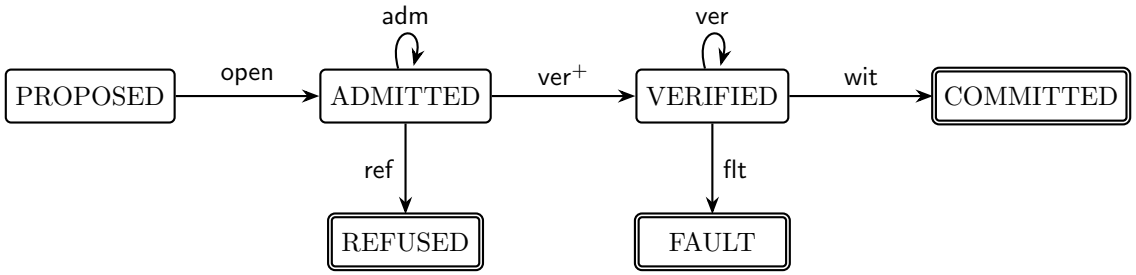

Figure 3: The per-transition ledger protocol. Refusal and fault are reachable from admitted and verified states alike. The diagram abbreviates the regular language $\mathcal{L}_t$ of Proposition 6.

Let $\mathcal{L}_t$ be the set of words over ledger entries tagged with identifier t given by

$$\mathsf{open}\ \mathsf{adm}^*\ \mathsf{ver}^*\ (\mathsf{ref} \mid \mathsf{flt}) \quad \Big| \quad \mathsf{open}\ \mathsf{adm}^*\ \mathsf{ver}^+\ \mathsf{wit}.$$

Proposition 6 (Ledger legality and prefix extension)**.** *Let $\lambda_0 = [\,]$ and let λ be the ledger at any point of a run. Then*

(a) for every step $C \longrightarrow C'$, $\lambda \preceq \lambda'$;

(b) λ is a prefix of a concatenation $w_1 w_2 \cdots w_k$ with each $w_i \in \mathcal{L}_{t_i}$, in order of opening;

(c) every frame that opens terminates in exactly one of commit, refuse, fault *after at most $m + 1$ steps, where m is its clause count.*

Proof. (a) Every rule either leaves λ unchanged or appends. (b) is by induction on the number of steps. A frame is created by the open rule, which appends open. Each clause rule appends the entry its clause kind licenses: adm only from an admission clause, ver only from a verification clause. By well-formedness, admission clauses precede verification clauses, so entries appear in the order of the regular expression. A frame ends only with a ref, flt, or wit, after which no further entry carries its identifier. Because the core is sequential, each frame runs to completion before another opens, so the words for distinct transitions do not interleave. (c) Each clause rule consumes one clause and leads to a further frame, a refuse or fault value, or, for the last clause, the commit rule. The commit rule is defined only for the single remaining clause, which well-formedness guarantees is $> g$. □

The proposition says something stronger than that history persists. Illegal histories cannot be generated by the semantics. The ledger cannot contain a witness before a verification, a witness after a refusal, or two witnesses for one transition, whatever the program.

Proposition 7 (Refusal and fault preservation)**.** *Suppose a bounded transition opened in $(\sigma, \lambda, \omega)$ reduces to* refuse(t, r) *or to* fault(t) *with final components $(\sigma', \lambda', \omega')$. Then $\sigma' = \sigma$, $\omega' = \omega$, and $\lambda \preceq \lambda'$, with $\lambda' = \lambda \parallel w$ for some $w \in \mathcal{L}_t$ ending in* ref *or* flt.

Proof. By Proposition 5 the store and effect trace are unchanged until commit, and the refuse and fault rules are reached only from a live frame before the commit clause. The ledger difference is given by Proposition 6(b). □

Hence

$$\text{REFUSE}(e) \;\Rightarrow\; \Delta\sigma = 0 \;\wedge\; \Delta\omega = 0, \qquad \text{while } \Delta\lambda \neq 0.$$

No state change does not imply that no event occurred. Writing I for information about an attempted transition, $\Delta\sigma = 0 \not\Rightarrow \Delta I = 0$. A refused proposal leaves the world unchanged and leaves evidence that it was evaluated. Negative events are events: the three-way outcome makes all of commit, refuse, and fault part of the history of evaluation, and the ledger tags refusal and fault differently, so that a timeout, a malformed transformer, or an unavailable store is not confused with a decision.

7.1 Proposal, evaluation, realization

The results above prove that three notions ordinary description merges are distinct. Let $\mathcal{P}$ be the proposals made, $\mathcal{E}$ those evaluated, and $\mathcal{R}$ those realized in the committed store. Then $\mathcal{R} \subseteq \mathcal{E} \subseteq \mathcal{P}$, and refusal and fault occupy $\mathcal{E} \setminus \mathcal{R}$ (Figure 4). A proposal can exist without being evaluated, can be evaluated without being realized, and can leave evidence of evaluation without becoming state. The category error avoided is the treatment of the representation of a possibility as the execution of that possibility.

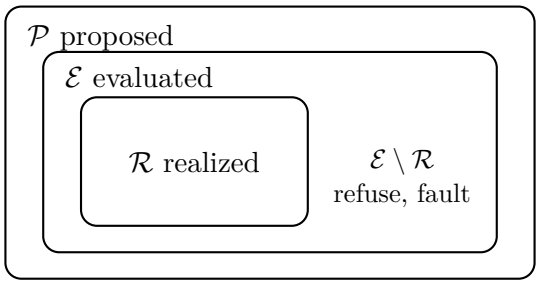

Figure 4: Proposal, evaluation, realization. Refusal and fault are evaluated but not realized.

7.2 Non-retroactivity

Refusal is terminal in a precise sense. By Proposition 6(a) the entry $\mathsf{ref}(t, \cdot)$, once appended, is in every later ledger, and the only rule that appends wit for t is the commit rule applied to the frame of t, which no longer exists. No rule takes $\mathsf{refuse}(t, r)$ as input and produces a commit. An enclosing context may inspect the value as data, react to it, and run a fresh bounded transition, and it may not make the historical claim that t committed true. Generalized:

a later context may reinterpret an outcome but cannot rewrite its occurrence.

8 Verified Commit and Witness Correspondence

8.1 Recorded versus valid

Presence of a ledger entry is not evidence that its content is correct. A witness is *recorded* when it occurs in λ. It is *valid* with respect to a trace prefix ρ when it satisfies conditions that can be checked against the trace independently of the commit rule.

Definition 1 (Valid witness)**.** ValidWitness(w, ρ) *holds when* $w = \langle t, a, b, V, A\rangle$ *and in* ρ*: there is a unique* $\mathsf{open}(t, a', Q)$ *with* $a = a'$*; the committed store immediately before* t*'s commit step has hash* a*; the committed store immediately after has hash* b*;* $Q \subseteq V$*; and* $A \neq \varnothing$*.*

The hash function is assumed injective on reachable stores (Assumption H). This is a modelling assumption that lets hash equality stand for store equality and is not a cryptographic claim.

8.2 Verified commit

Proposition 8 (Verified commit)**.** *Let* ρ *be any run of a program whose bounded transitions are well-formed. If a commit step for transition* t *occurs in* ρ*, then before that step the ledger contains* $\mathsf{ver}(t, q)$ *for every* $q \in Q_t$*, contains no* $\mathsf{ref}(t, \cdot)$ *or* $\mathsf{flt}(t, \cdot)$*, and* $A_t \neq \varnothing$*.*

Proof. The clause sequence is consumed head first. By well-formedness the commit clause is last, so the frame reaches $[>g]$ only after every earlier clause has been consumed. A verification clause is consumed only by the verify-true rule, which appends $\mathsf{ver}(t, q)$. A refusal or fault ends the frame and no later clause is reached. Well-formedness requires at least one attribution clause, and each such clause adds to A. □

The result follows from the interaction between the static ordering discipline and the small-step rules. It is not a definitional tautology, since it fails for programs that are not well-formed (Section 10).

8.3 Correspondence

Proposition 9 (Witness correspondence)**.** *For every run* ρ *from an initial configuration with empty ledger and every prefix of it:*

(a) every step that changes σ *or* ω *is a commit step;*

(b) every commit step appends exactly one wit *entry, and that entry is valid with respect to the prefix ending at that step;*

(c) every wit *entry in the ledger arises from exactly one commit step.*

Consequently the number of commit steps equals the number of valid witness records in the ledger, at every prefix.

Proof. (a) is established by inspecting the rules: only commit writes σ or ω. For (b), the commit rule appends one record. Its open entry exists because the frame was created by the open rule, its pre-hash equals $h(\sigma_0)$ and, by Proposition 5, σ_0 equals the committed store immediately before the commit step. Its post-hash is $h(\sigma^*)$, the store installed by the step.

The verified set V is read from the ledger and contains Q_t by Proposition 8. For (c), no rule other than commit appends a wit entry. □

The correspondence gives what may be called conservation of committed history:

$$\#\text{Commit} = \#\text{ValidWitness}.$$

It is stronger than ordinary auditability. Auditability asserts Commit → Witness. The invariant here is Commit ↔ ValidWitness: committed change cannot appear without a corresponding witness record, and records of that class cannot appear without committed change. Part (a) catches back doors. Parts (b) and (c) give the converse. Figure 5 shows the matching and its failure modes.

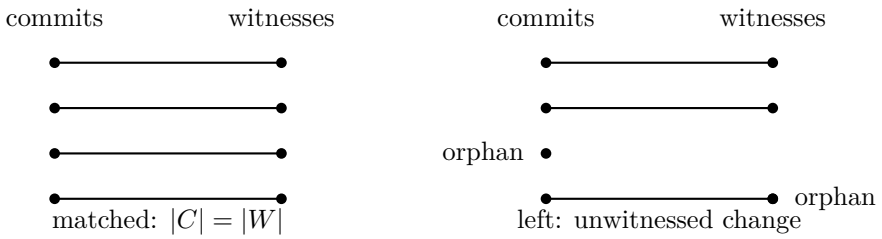

Figure 5: Commit and witness correspondence (Prop. 9). In the right panel, a left orphan is a state change without a witness and a right orphan is a witness without a state change. Both are excluded.

8.4 Event and witness

A witness relates the endpoints of a transition and is not the transition. Figure 6 shows the relation, and the record is $w = \langle t, h(\sigma_0), h(\sigma_1), V, A\rangle$. Yet the record can constrain later events, and Section 12 returns to that.

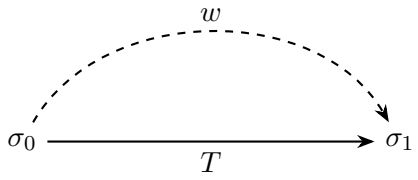

Figure 6: A witness relates the endpoints of a transition ($w \neq T$).

8.5 Persistence and truth

A valid witness certifies structural facts about a step: which transition ran, from what pre-state, to what post-state, with which predicates having returned true, and under what declared provenance. It does not certify that the content it carries is true:

$$\text{Persisted}(w) \not\Rightarrow \text{True}(\text{content}(w)).$$

What persistence supports is the weaker and more defensible statement that this record entered the ledger under this procedure. A downstream consumer that folds only valid witnesses into a derived state $M_{t+1} = \mathcal{M}(M_t, w_t)$ has the following property.

Corollary 1 (Refused proposals are not folded as state)**.** *If the transition t refuses, no* wit *entry for t exists, so $\mathcal{M}$ is not applied for t and the derived state does not represent the proposal as having occurred. The ledger may record that it was proposed and refused.*

9 Transaction Soundness

The preceding results combine into a single statement about bounded transitions.

Theorem 1 (Transaction soundness)**.** *Let $\langle\!\langle v; \vec{c} \rangle\!\rangle$ be a well-formed bounded transition opened in $(\sigma, \lambda, \omega)$. Its frame reaches exactly one outcome, with final components $(\sigma', \lambda', \omega')$, and in every case $\lambda \preceq \lambda'$. Moreover:*

(i) if the outcome is $\mathsf{commit}(u)$*, then $\sigma' = \sigma^*_{\text{final}}$, every declared verification is recorded, and exactly one valid witness is appended;*

(ii) if the outcome is $\mathsf{refuse}(r)$ *or* $\mathsf{fault}(f)$*, then $\sigma' = \sigma$ and $\omega' = \omega$;*

(iii) $\lambda' = \lambda \parallel w$ with $w \in \mathcal{L}_t$.

Proof. Exhaustiveness, prefix extension, and $w \in \mathcal{L}_t$ are Proposition 6. The commit clause follows from Propositions 8 and 9(b), together with the fact that the commit rule installs σ^*. The refuse and fault clauses are Proposition 7. □

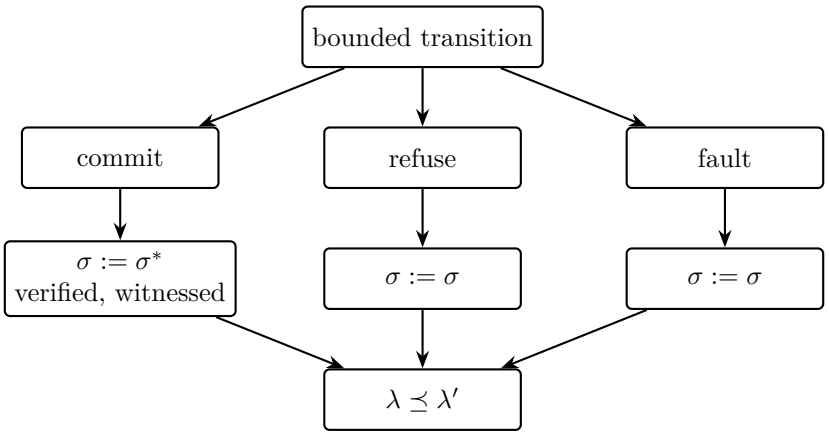

Figure 7: Theorem 1: three outcomes, one prefix-preserved ledger.

10 Negative Examples

The ordering discipline is justified by programs that must be rejected. The following are ill-formed under the static judgment.

```
<( req  ~ write  ? allowed  ! chk  @ src  > done )>
```

Admission after transformation. In the core this is harmless, since `?` reads σ_0. In any extension where a transformer can produce an effect, the test would run after the effect and refusal could not preserve state. The static judgment excludes the pattern in the core so that extensions cannot introduce it silently.

```
<( req  ? allowed  ~ write  @ src  > done  ! chk )>
```

Verification after commit. The commit clause is not last. Verification cannot establish the invariant after the store has been installed, and Proposition 8 fails for the resulting trace: the commit precedes any `ver` entry.

```
<( req  ? allowed  ~ write  @ src  > done )>
```

No verification clause. A commit reachable without verification would let a transition become durable on the strength of a pre-state test alone.

```
<( req  ? allowed  ~ f  ! chk  @ src  > c1  > c2 )>
```

Two commits. Exactly one commit clause is allowed, since Proposition 9 counts one witness per commit and one commit per transition.

11 Enclosing Boundaries

A boundary is more than a grouping. If B relates an outside and an inside, then for the boundary to be semantically real there must be a property P with $P_{\text{inside}} \neq P_{\text{outside}}$. Applied to the core: the parenthesis `( )` marks an evaluation region, and fixes the order of reduction, but changes no admissibility, so it is a context and not a semantic boundary. The delimiters `<( )>` are a boundary in the strong sense, since inside a frame the visible store is σ^* and outside it is σ. A boundary with no change in admissibility is representational decoration, and the criterion classifies the two pairs accordingly.

Crossing is asymmetric. Entering with a proposal and leaving with an outcome are not inverse operations, $\text{exit}(\text{enter}(v)) \neq v$: what returns is a value, a refusal, or a fault, together with changes to state and ledger. The boundary is an operator on the conditions of return.

Consider an enclosing sequence of layers $1, \ldots, n$, each contributing a guard applied to a proposed transition. Let $\mathcal{T}_0$ be the set of transitions a bounded transition may perform, and let F_i be the operator by which layer i modifies the permitted set:

$$\mathcal{T}_i = F_i(\mathcal{T}_{i-1}).$$

A boundary may transform transitions and not only accept or reject them. A purely restrictive layer satisfies $F_i(\mathcal{T}) \subseteq \mathcal{T}$.

Proposition 10 (Restrictive boundaries narrow monotonically)**.** *If every F_i is restrictive, then $\mathcal{T}_n \subseteq \cdots \subseteq \mathcal{T}_1 \subseteq \mathcal{T}_0$. If in addition each F_i is a pointwise filter, $F_i(\mathcal{T}) = \mathcal{T} \cap A_i$, then $\mathcal{T}_n = \mathcal{T}_0 \cap \bigcap_i A_i$.*

Proof. The first claim is induction using $F_i(\mathcal{T}) \subseteq \mathcal{T}$. The second follows by unrolling the definition and using associativity and commutativity of intersection. □

The intersection formula is a special case that requires independence of the guards and a shared transition universe. Constraint composition need not be set intersection. Together with the terminality of refusal (Section 7) the result gives the formal counterpart of "return outward carrying only what every enclosing boundary admits," in the case of restrictive layers. Figure 8 shows the nesting.

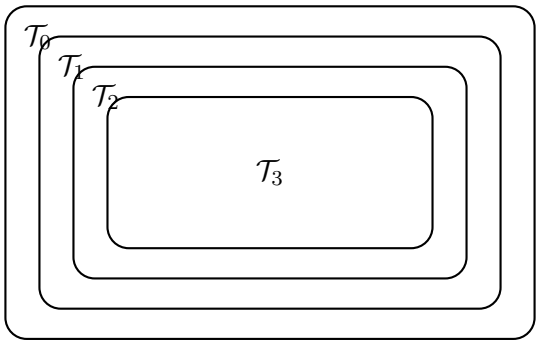

Figure 8: Restrictive boundaries only: $\mathcal{T}_3 \subseteq \mathcal{T}_2 \subseteq \mathcal{T}_1 \subseteq \mathcal{T}_0$. General boundaries may transform the set and are not drawn this way.

12 General Principles

Several principles separate from the particular calculus and apply to any system with boundaries, proposals, and records.

Operational iconicity. Syntax usually has conventional meaning. Here the design goal is that

$$\text{structure(representation)} \approx \text{structure(evaluation)}.$$

Nested regions correspond to nested evaluation, opening to entering, closing to returning, and sequential operators to successive motions. The relation is structure-preserving and not an identity between text and execution, and it is testable: the order of region entry in the trace must equal the pre-order of the AST (Section 16).

Witness feeds admissibility. The event and the record of the event are different objects, $T \neq W(T)$, but the record can constrain later events, $W(T_t) \to A(T_{t+1})$. In the core the admission predicates do not read the ledger. An extension in which they do makes the admissible space a function of history,

$$A_{t+1} = F(A_t, W_t),$$

so that an event does not merely add a record but can change what may happen next. Prefix extension and legality are unaffected, since predicates remain pure.

Depth versus size. A small expression can induce a long traversal. Run length in the core is bounded by program size, since there is no recursion, so the point applies to extensions and to the doorway analogy below: representations can be small because they preserve paths and not bodies.

Continuation over snapshot. A witness stores $(t, h(\sigma_0), h(\sigma_1), V, A)$, that is, an edge and its authorization, and not a copy of either store. Figure 9 contrasts what is preserved. To preserve a process it suffices to preserve enough to continue it, and not necessarily enough to copy it.

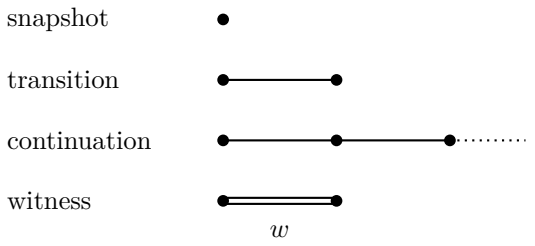

Figure 9: Preservation targets: a vertex, an edge, a path, and an edge together with its record.

13 Representation as Doorway

A concrete illustration is a 52-character string that encodes a video link. It is offered as an analogy, not as evidence for the semantics. The bytes do not contain the video. The URL does not contain the video. Even a time parameter such as `t=86` does not contain the event at that timestamp. The string preserves a traversable specification:

$$b \xrightarrow{\text{decode}} s \xrightarrow{\text{parse}} r \xrightarrow{\text{resolve}}_{\mathcal{E}} x.$$

Each layer admits only certain transformations. Changing one byte may invalidate the string, whereas changing `86` to `87` preserves the resource and changes the requested coordinate. The important property is partiality:

$$r \neq x, \qquad \text{resolve}(r, \mathcal{E}) \rightharpoonup x.$$

A doorway can fail to resolve (Figure 10), which strengthens the analogy: a representation is a preserved possibility of traversal and not a guarantee of arrival. The witness of Section 8 is the corresponding artifact for a state transition: it does not contain the transition but preserves enough structure for another evaluator to check it.

Three preservation targets are distinguished:

$$\begin{aligned} \text{object preservation} &: x_t = x_0, \\ \text{representation preservation} &: r_t = r_0, \\ \text{reachability preservation} &: \text{resolve}(r_t, \mathcal{E}_t) \rightharpoonup x_t. \end{aligned}$$

The third is the interesting case: something can change substantially while remaining reachable through a valid continuation, and persistence of access is not persistence of object.

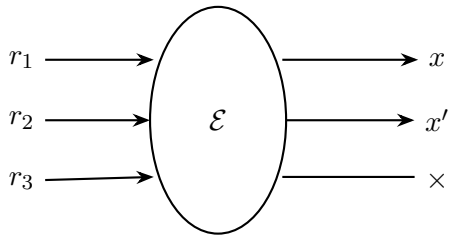

Figure 10: Partial resolution. Two representations reach referents; the third does not resolve.

14 Extensions Outside the Core

The following are specified informally and are kept out of the proofs.

Persistence and reconstruction. The flush motion `#` binds the ledger to a host store. Durability guarantees, and the failure modes of an unavailable store, belong here, and an unavailable ledger is a fault, not a refusal. The reconstruct symbol `^` recovers state from recorded continuations. The intended obligation is a non-invention property. If R is an explicitly declared set of reconstruction rules and $\mathrm{Closure}(W, R)$ is what can be derived from witnessed records W under R, then

$$\mathrm{Recover}(W) \preceq \mathrm{Closure}(W, R),$$
$$\mathrm{Recover}(W) \not\models q \quad \text{whenever } q \text{ is not derivable from } W \text{ under } R.$$

Recovery may reconstruct what the evidence licenses and may not silently manufacture continuity. As stated this is close to a definition, so the substantive obligation falls on the implementation: to show that the recovery procedure is sound with respect to the closure.

Closed nesting and concurrency. In the core a bounded transition placed in the proposal position of another is evaluated to completion, including its own commit, before the outer frame opens. It is a sequential composition. Closed nesting, in which an inner commit installs into the outer staged store, requires an additional rule, and concurrency requires a validation step at commit. Each introduces sources of nondeterminism that Proposition 2 excludes.

Larger interpretations. Semantic interpretations of the calculus into other frameworks are treated as maps $\mathcal{I} : C \to \Phi$ out of the calculus. The proofs do not depend on any such framework, and a framework may consume the semantics without being presupposed by it.

15 ASCII as a Representational Invariant

The restriction to printable ASCII fixes a representational invariant, and not a semantic one. Replacing a symbol by a Unicode glyph need not change its denotation, and the essay does not claim otherwise. What the restriction secures is the round trip of Proposition 1: with a canonical layout, $\mathrm{print}(\mathrm{parse}(s)) = s$, so the source that displays the structure and the bytes that store it are the same object, and the claim that the same source yields the same diagram becomes testable. A program survives terminals, shells, version-control diffs, serial

links, minimal editors, and archival formats without a rendering layer. The proposition under test is that a spatially suggestive notation does not require a graphical interface: indentation and nesting of ASCII delimiters can carry the structural information of a diagram if the operators are chosen so that layout and evaluation order agree.

16 Prototype and Evaluation Criteria

A reference interpreter follows the levels of Section 4. It tokenizes the reserved symbols, parses into well-formed ASTs, rejects ill-formed programs with a specific reason, executes the small-step rules, and emits a trace. Trace entries carry region identifiers,

$$t_i = (\mathit{region},\ \mathit{operation},\ \mathit{input},\ \mathit{result}),$$

since without them different programs could produce identical traces. The interpreter should be small enough to read in one sitting.

The language is not evaluated by asserting that it is better. Each result of the paper is a property that can be tested mechanically on generated and hand-written programs, in the manner of property-based testing, so that the prototype functions as an executable test of the formal claims and not as an illustration of them.

(i) **Round trip.** $\mathrm{print}(\mathrm{parse}(s)) = s$ on canonical sources (Prop. 1).

(ii) **Determinism.** Repeated runs from the same configuration produce identical traces (Prop. 2).

(iii) **Iconicity.** The order of region entry and exit in the trace matches the order derived from the AST. The trace determines the sequence of region entries and exits, and full source reconstruction is not claimed (Prop. 3).

(iv) **Isolation.** σ and ω are constant between an open step and the corresponding commit, tested with transformers that would emit effects if permitted (Prop. 5).

(v) **Legality.** Every ledger belongs to the prefix language of Proposition 6, checked by a small automaton.

(vi) **Inert refusal and fault.** On every refused or faulted transition, σ and ω are unchanged (Prop. 7).

(vii) **Verified commit and correspondence.** A trace checker computes ValidWitness from the trace alone and confirms Propositions 8 and 9, including count equality at every prefix.

(viii) **Negative cases.** Each program of Section 10 is rejected for the stated reason, and a variant interpreter that admits it exhibits the predicted failure.

(ix) **Motion table.** Each transition rule belongs to exactly one motion and each core motion has a symbol (Section 3).

A failed check is informative. If (iii) fails the layout does not mirror evaluation for that construct. If (ix) fails, some symbol is doing more than one job or some motion is unnamed, and the design principle has been violated.

17 Non-Collapse Principles

The results above can be summarized as a family of distinctions. Each is instantiated at a specific point of the calculus, and none is an imposed philosophical claim.

Distinction	Where it is instantiated
representation $\neq$ referent	partial resolution; parse versus evaluate (Sections 2, 13)
proposal $\neq$ realization	$\mathcal{R} \subseteq \mathcal{E} \subseteq \mathcal{P}$; refusal preservation (Prop. 7)
evaluation $\neq$ exposure	staged store; boundary locality (Prop. 5)
admission $\neq$ verification	signatures of `?` and `!`; prospective versus relational (Section 3)
event $\neq$ witness	recorded versus valid; $w \neq T$ (Section 8)
refusal $\neq$ absence	$\Delta\sigma = 0$ with $\Delta\lambda \neq 0$ (Prop. 7)
persistence $\neq$ truth	valid witness certifies structure, not content (Section 8)
recovery $\neq$ invention	closure bound on reconstruction (Section 14)

Table 3: Non-collapse principles and their instances.

The calculus works by refusing to collapse objects that ordinary description casually identifies. Evaluation proceeds inward by decomposition, and continuation proceeds outward by admissibility. The ASCII language is an economical demonstration of that thesis and is not the thesis itself. The essay is about the relations among representation, traversal, boundary, admissibility, witness, and continuation, and the notation is the small formal object through which those relations become visible.

18 Related Work and Claimed Contribution

None of the following is claimed as new individually: symbolic domain-specific languages, visible evaluation structure, ASCII notations, transaction-like semantics, or Lisp-style nesting. Lisp and Scheme supply the identification of program text with tree structure [1]. Teaching notations that draw nested circles motivate the reading of parentheses as evaluation boundaries [2]. Concatenative languages such as Forth show that a very small symbolic vocabulary can carry substantial semantics [3]. Evaluation contexts and syntactic operational semantics supply the machinery of Section 4 [4, 5, 6]. Effect systems classify computations by the effects they may perform [7], which is the natural home of the compensable and irrevocable distinction. The transactional properties are those of the database tradition [8] and of transactional memory [9]. The ledger is close to event sourcing [10], and the recorded-versus-valid distinction is close to work on provenance [11]. Capability-oriented design motivates restricting what a transformer may reach [12], and reversible computation [13] motivates the care taken over what recovery may claim.

The candidate contribution is the particular conjunction:

> evaluation-visible ASCII syntax, plus bounded admissibility, plus witness-producing transitions, plus formal traversal semantics.

Its value, if any, lies in the fact that each symbol is tied to a restriction on transitions, each restriction is tied to a proved property, and a small interpreter can check every such property.

19 Limitations

The core is sequential and deterministic, and the guarantees weaken when those assumptions are relaxed. The meta-level functions p, f, q, g are assumed given, and their correctness is outside the calculus. The hash assumption is a modelling device. Validity of a witness concerns structure and does not extend to the truth of predicates beyond the fact that they returned true. The proofs are by inspection of the rules and have not been mechanized. The evaluation criteria test internal consistency and legibility. Whether programmers make fewer errors in this notation is a separate empirical question that would require a user study.

20 Conclusion

Parentheses expose the structure of evaluation. A small set of further symbols can be assigned distinct restrictions on state transition: admit against the pre-state, transform a staged store, verify the before-and-after relation, attribute provenance, and commit as the only durable write. With well-formedness fixing their order, the core calculus yields a chain of results: parsing is unique, the machine is deterministic, nested evaluation decomposes into inner and outer runs, staging isolates committed state, ledger histories are legal and only extend, refusal and fault change nothing but the ledger, commit implies verification, commits correspond to valid witnesses, and restrictive boundaries only narrow what may pass. The chain closes in a single soundness theorem for bounded transitions. Beneath it lies a second chain of distinctions the calculus declines to collapse. The evocative language survives as commentary on these results and does not stand in for them.

References

[1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed., MIT Press, 1996.

[2] E. Schanzer, K. Fisler, S. Krishnamurthi, and M. Felleisen, "Transferring skills at solving word problems from computing to algebra through Bootstrap," *Proc. SIGCSE*, 2015.

[3] L. Brodie, *Starting Forth*, Prentice Hall, 1981.

[4] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theoretical Computer Science* 103(2), 1992.

[5] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information and Computation* 115(1), 1994.

[6] G. D. Plotkin, "A structural approach to operational semantics," *Journal of Logic and Algebraic Programming* 60–61, 2004.

[7] D. K. Gifford and J. M. Lucassen, "Integrating functional and imperative programming," *Proc. ACM Conference on LISP and Functional Programming*, 1986.

[8] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys* 15(4), 1983.

[9] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *Proc. ISCA*, 1993.

[10] M. Fowler, "Event sourcing," online article, 2005.

[11] P. Buneman, S. Khanna, and W.-C. Tan, "Why and where: a characterization of data provenance," *Proc. ICDT*, 2001.

[12] M. S. Miller, *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*, PhD thesis, Johns Hopkins University, 2006.

[13] C. H. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development* 17(6), 1973.