Position-Dependent Admissibility in Combinatory Reduction: Confluence with Replayable Evidence and the Limits of Supervision

Flyxion
Independent Researcher

November 2026

Abstract

Safe execution of untrusted computation is usually bought with a supervising process: a monitor that holds the authority to permit or refuse each transformation, watches the worker, and keeps the record. This essay examines a small system in which part of that authority lives in the term being reduced instead. The system is the circle calculus of SpherePOP, a combinatory reduction system whose structural combinators (contraction W, weakening K, exchange C) are gated by permissions attached to bracketed scopes, together with a reference implementation in Go and an independent checker for the records it produces. The essay draws together two earlier technical results and one body of implementation evidence. From the first, on duplication and universality, it takes the facts that the unrestricted calculus does not terminate, that termination under a given strategy is a separate matter from agreement of results, and that the halting direction of Turing completeness is proved while its converse remains conditional. From the second, on scoped structural rules, it takes the separation of the availability of a structural rule from its admissibility at a redex, the exact condition under which boundary rules preserve confluence, the descent of effective permission under reduction, and the comparison of copying with sharing in term graphs. From the implementation it takes a confluence theorem for a restricted scope constructor, the independent replay of traces, scheduler logs and effect logs, and a conformance corpus on which two implementations must agree. The thesis is then stated in a narrower form than the one with which the project began. Authority to transform is carried by the position of a step in the term, read against a table from scope labels to permissions that belongs to the agreed configuration and not to the term, and the effective permission only shrinks along a reduction; together with confluence this makes the result independent of evaluation order. Authority to accept a result and authority to act on it are deliberately not carried by the term, and remain with the host. Several guarantees that a supervisor is usually asked to supply are shown to follow from the rule table, a fuel bound and a replayable record, and the places where a supervisor is still needed are listed with the same care. A supervised baseline built on the same rule-application routine, with a flat worker, an independently written policy and a hash-chained audit log, is compared with the scoped design on 184,860 runs: the two never disagree about admissibility, the supervised checker that matches the detection power of the scoped one needs about a third more trusted code, it detects nothing that replay of the scoped record misses, and the supervised run costs between 1.06 and 2.0 times as much per step. What the scoped design offers is economy and a record with fewer places to tamper, not additional detection, and it gives up the record of refused attempts that a supervisor keeps. The confluence of the safe scope constructor is open, and the proof offered for the settled constructor is a paper proof that has not been reviewed.

Contents

1 Introduction **2**

2 The Calculus **3**

3 Duplication, Nontermination, and the Dependence on Strategy **4**
3.1 Confluence and the Ability to Find Results 5
3.2 Universality 6

4 Scoped Permission **7**
4.1 Confluence and the Boundary Rules 8
4.2 Restricted Absorption and the Settled Scope Constructor 9
4.3 Descent of Admissibility 10
4.4 Copying, Sharing and Erasure 11

5 The Reference Implementation **11**

6 Evidence: Computation, Record and Acceptance **12**

7 Scheduling and Effects **13**
7.1 The Scheduler 13
7.2 The Effects Boundary 14

8 What the Term Carries and What a Supervisor Keeps **15**

9 The Supervised Baseline **16**

10 Open Obligations and Conclusion **19**

1 Introduction

A computation that arrives from outside raises three questions that are easy to run together. The first is whether each transformation it performs is licensed, that is, whether it is an instance of a rule that the host has agreed to allow at the place where it is applied. The second is whether the result should be accepted, which is a decision made by whoever has the standing to make it and depends on what the result is for. The third is whether an effect on the world should follow, which is a further decision with its own standing. Systems that execute foreign code usually answer all three with one mechanism, a supervising process that interposes on the worker, holds the permissions, and writes the audit trail. The mechanism is powerful because it is general, and it is expensive for the same reason: a monitor that can refuse any action must be trusted to see every action, and the trusted code grows with the variety of actions it has to see.

The system studied here takes a different route to the first question and the same route as the supervisor to the other two. The licence to transform is made a function of position in the term. A term is built from combinators and from labelled bracketed scopes, the configuration maps each label to a set of the structural combinators that may act inside it, and a step at a given position is admissible only if every enclosing scope permits the combinator at its head.

What the term carries is therefore the labels and their nesting, and what the host agrees is the meaning of the labels; the table that gives them meaning is written into every record. Nothing watches the worker for this, because the rule table that performs the step also contains the condition under which the step is allowed, and a record of the steps can be replayed by anyone who holds the same table. The licence to accept and the licence to act are kept out of the term altogether: a replay checker establishes that each recorded step was licensed and says nothing about whether the result should be committed, and the effects of a program are read from a normal form and passed to a host-held authorizer that the reduction never touches.

The essay has three purposes. The first is to give one self-contained account of the formal core on which the argument rests, which until now has been spread over two essays. That core is a combinatory calculus whose application circle duplicates and discards terms, whose consequent nontermination and strategy dependence are exhibited exactly, and whose structural combinators can be gated by scope. The second is to report what a small reference implementation and its independent checker establish about the calculus, with the bounds attached to every statement. The third is to state precisely which parts of the work usually assigned to a supervisor are discharged by the term and its record, which are not, and what a measured comparison with a supervised baseline shows about the difference.

Every numbered statement carries one of four tags. A statement tagged status E is established by a complete proof in the text. A statement tagged status C is proved from a cited theorem, and the verification of that theorem's hypotheses is part of the proof. A statement tagged status T has been checked on a finite set of cases whose bounds are stated, and the check does not extend beyond them. A statement tagged status P is a proposal or an open problem, and nothing is claimed for it. Results that come from the implementation and not from the earlier essays are marked as such where they appear, and each carries the commit of the repository from which it was taken (Flyxion, 2026).

Two distinctions organize the material and recur throughout. The first is between a rule being available in a system and being admissible at a given redex. The second is between confluence, which says that different choices of redex can be reconciled, and determinism, which says that there is only one choice; none of the systems here is deterministic, and the guarantees below use confluence alone. A third distinction, between computation, evidence and acceptance, is the architectural principle of the implementation and is taken up in Section 6.

2 The Calculus

Terms are generated by atoms, the constants $\mathsf{I}, \mathsf{B}, \mathsf{C}, \mathsf{K}, \mathsf{W}, \mathsf{S}$, and binary application. In the circle reading every application is a circle containing two operands, and popping a circle applies a rule to it. The reduction rules are

$$\begin{aligned} \mathsf{I}\,x &\to x, & \mathsf{B}\,f\,g\,x &\to f\,(g\,x), & \mathsf{C}\,f\,x\,y &\to f\,y\,x,\\ \mathsf{K}\,x\,y &\to x, & \mathsf{W}\,f\,x &\to f\,x\,x, & \mathsf{S}\,f\,g\,x &\to f\,x\,(g\,x). \end{aligned}$$

In the pure fragment of the circle calculus, where every operator consumes values and returns a value, each pop removes exactly one circle, so evaluation terminates and independent pops commute. The unrestricted application circle, whose operands need not be values, together with the contraction rules above, is what the rest of the essay concerns. The combinators I and B neither copy, discard nor reorder. The other three realize the structural rules of logic

directly: C is exchange, K is weakening, and W is contraction in the literal sense that its type $(A \to A \to B) \to A \to B$ merges two occurrences of a hypothesis. The combinator S is contraction only operationally, because it duplicates its third argument while also applying two functions. The relation is exact.

Proposition 2.1 (status E)**.** *Over the base* $\{\mathsf{I}, \mathsf{B}, \mathsf{C}\}$ *the combinators* S *and* W *define one another:* $\mathsf{W} = \mathsf{C}\,\mathsf{S}\,\mathsf{I}$ *and* $\mathsf{S} = \mathsf{B}\,(\mathsf{B}\,\mathsf{W})\,(\mathsf{B}\,\mathsf{B}\,\mathsf{C})$.

Proof. For the first identity, $\mathsf{C}\,\mathsf{S}\,\mathsf{I}\,x\,y \to \mathsf{S}\,x\,\mathsf{I}\,y \to x\,y\,(\mathsf{I}\,y) \to x\,y\,y$ by the rules of C, S and I. For the second, write $M = \mathsf{B}(\mathsf{BW})(\mathsf{BBC})$. Two uses of the rule of B give $M\,f\,g\,x \to \mathsf{W}\,(\mathsf{BBC}\,f\,g)\,x$, the rule of W gives $\mathsf{BBC}\,f\,g\,x\,x$, the rule of B rewrites the prefix $\mathsf{BBC}\,f$ to $\mathsf{B}\,(\mathsf{C}\,f)$ so that the term becomes $\mathsf{B}\,(\mathsf{C}\,f)\,g\,x\,x$, a further use of B gives $\mathsf{C}\,f\,(g\,x)\,x$, and the rule of C gives $f\,x\,(g\,x)$. □

The proposition says that identifying S with contraction is correct in the presence of exchange and incorrect without it. For that reason the permission structures of Section 4 range over $\{\mathsf{W}, \mathsf{K}, \mathsf{C}\}$ and treat S as a derived combinator whose admissibility is computed from those of its components. The three structural combinators cannot be recovered from the neutral ones, which is what makes gating them meaningful.

Proposition 2.2 (status C)**.** *No term over* $\{\mathsf{I}, \mathsf{B}, \mathsf{K}, \mathsf{C}\}$ *defines* W *or* S*, and no term over* $\{\mathsf{I}, \mathsf{B}, \mathsf{C}, \mathsf{W}, \mathsf{S}\}$ *defines* K.

Proof. Translate each combinator into the lambda calculus in the standard way. The translations of $\mathsf{I}, \mathsf{B}, \mathsf{K}, \mathsf{C}$ are closed affine terms, in which no variable occurs free more than once, and affine reduction strictly decreases size, so by the Church-Rosser theorem every combination of them has a normal form in which each free variable occurs at most once. A term T with $T\,f\,x \twoheadrightarrow f\,x\,x$ would have the normal form $f\,x\,x$, in which x occurs twice, contradicting this; the argument for S is the same with $f\,x\,(g\,x)$. For the second claim, the translations of $\mathsf{I}, \mathsf{B}, \mathsf{C}, \mathsf{W}, \mathsf{S}$ are $\lambda\mathsf{I}$-terms, and β-reduction of $\lambda\mathsf{I}$-terms preserves the set of free variables. A term T with $T\,a\,b \twoheadrightarrow a$ would reduce to a term with free variables $\{a\}$ from one with free variables $\{a, b\}$. The classical facts used are in Barendregt (1984) and, for the $\lambda\mathsf{I}$ case, Church (1941). □

The independence of exchange from the other four is more delicate, because deletion and duplication disturb the order of variable occurrences, and it is not claimed. The logical reading follows the standard correspondence between combinatory bases and implicational logics (Curry and Feys, 1958; Hindley and Seldin, 2008): for a set of structural combinators containing C, a formula is a theorem of the logic with exactly those structural rules if and only if it is the type of a closed combination of I, B and the members of the set. The four commutative cases are the linear, affine, relevant and unrestricted logics, and without C the corresponding logics are noncommutative in the manner of Lambek (1958). Linear logic proper (Girard, 1987) adds exponential modalities, and the resemblance between those modalities and permission-bearing scopes is noted in Section 10.

3 Duplication, Nontermination, and the Dependence on Strategy

The pure fragment terminates because each pop removes a circle. The size measure that proves this fails as soon as the application circle and the contraction rules are admitted, since one rule

deletes circles and another creates arbitrarily many by duplicating an operand. Write $\Delta = \mathsf{S}\,\mathsf{I}\,\mathsf{I}$ and $\Omega = \Delta\,\Delta$. Since I is $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$ and has two circles, Δ has six circles and Ω has thirteen. The behaviour of Ω is the standard example of nontermination, and what matters here is that it depends on the order in which circles are popped, in a way that can be stated exactly.

Lemma 3.1 (Contracting an identity; status E)**.** *For all terms a and b, the term $\mathsf{I}\,a\,b$ reduces in exactly two leftmost-outermost steps to $a\,b$, and $\mathsf{I}\,a$ reduces in two steps to a.*

Proof. In $\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,a\,b$ the head redex is $\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,a$, which contracts to $\mathsf{K}\,a\,(\mathsf{K}\,a)$ applied to b. The leftmost-outermost redex of that term is $\mathsf{K}\,a\,(\mathsf{K}\,a)$, with contractum a. □

Let $\mathsf{I}^n\Delta$ denote I applied n times to Δ, so that $\Omega = \Delta\,(\mathsf{I}^0\Delta)$.

Proposition 3.2 (The leftmost-outermost sequence of Ω; status E)**.** *For every $n \geq 0$ the term $\Delta\,(\mathsf{I}^n\Delta)$ has $13 + 3n$ circles and reduces under leftmost-outermost reduction in exactly $2n + 3$ steps to $\Delta\,(\mathsf{I}^{n+1}\Delta)$. The term Ω therefore has an infinite leftmost-outermost reduction in which the sizes strictly increase, so no term repeats.*

Proof. Write $x = \mathsf{I}^n\Delta$. The term $\Delta\,x = \mathsf{S}\,\mathsf{I}\,\mathsf{I}\,x$ is a redex, and it is leftmost and outermost; contracting it gives $\mathsf{I}\,x\,(\mathsf{I}\,x)$. By Lemma 3.1 this reduces in two further steps to $x\,(\mathsf{I}\,x)$. If $n \geq 1$ then $x = \mathsf{I}\,(\mathsf{I}^{n-1}\Delta)$, so the term is $\mathsf{I}\,(\mathsf{I}^{n-1}\Delta)\,(\mathsf{I}\,x)$ and the lemma again removes one identity at the cost of two steps. After the n outer identities are removed the term is $\Delta\,(\mathsf{I}\,x) = \Delta\,(\mathsf{I}^{n+1}\Delta)$, and the total is $1 + 2 + 2n = 2n + 3$ steps, each at a head redex and therefore leftmost and outermost. Each application of I adds three circles, so $|\mathsf{I}^n\Delta| = 6 + 3n$ and $|\Delta\,(\mathsf{I}^n\Delta)| = 1 + 6 + (6 + 3n) = 13 + 3n$. □

A cycle detector that watches for a repeated term would therefore report no cycle on this sequence and run forever, which is correct behaviour and not a defect of the detector. Other strategies behave differently. Under leftmost-innermost, rightmost-innermost and rightmost-outermost reduction, the engine of the earlier essay finds that Ω returns to itself after five contractions, and reports period five for each of the three. This is a measurement of the engine (status T), and the explicit leftmost-innermost path is recorded in the repository of the earlier essay. The sharpest demonstration of strategy dependence is the term $\mathsf{K}\,a\,\Omega$. Under the two outermost strategies the redex $\mathsf{K}\,a\,\Omega$ is contracted first, discards Ω unexamined, and yields the normal form a in one step. Under the two innermost strategies the argument Ω is reduced first, and cycles, so the same term has a normal form that one strategy finds in a single step and another never finds. In this exact sense the unrestricted application circle is call by name, and the value-restricted discipline of the pure fragment, which pops operands first, becomes unsafe once operands need not be values.

Nontermination does not require a cycle. Let $\mathsf{M} = \mathsf{S}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I})\,\mathsf{I}$. For every term x one has $\mathsf{M}\,x \twoheadrightarrow \mathsf{S}\,\mathsf{I}\,\mathsf{I}\,x\,(\mathsf{I}\,x) \twoheadrightarrow x\,x\,x$, so $\mathsf{M}\,\mathsf{M} \twoheadrightarrow \mathsf{M}\,\mathsf{M}\,\mathsf{M}$, and applying the same equation to the leftmost two copies of the left-nested application of k copies of M gives an infinite reduction in which the number of circles is unbounded. The tree grows roughly linearly with the number of steps while the number of distinct nodes grows much more slowly, which is the effect of sharing and is the reason the cost of reduction should be reported both as a step count and as a term size.

3.1 Confluence and the Ability to Find Results

Strategy dependence of termination does not contradict agreement of results. The system with the six rules above is confluent: by the parallel-reduction method of Takahashi (1995), if t

reduces in parallel to u and to v, both reduce in parallel to the complete development of t, and the diamond property of parallel reduction yields confluence of the reduction relation. Two consequences should be kept apart. Confluence implies that a term has at most one normal form, so that whenever two strategies both produce a result the results agree. It does not imply that a given strategy produces a result whenever one exists, as $\mathsf{K}\, a\, \Omega$ shows. The first property belongs to the system and the second to the strategy, and the guarantees of Section 8 use the first only. A strategy is a function of the current term, so a run can be stopped and resumed without changing the outcome, a fact used by the scheduler in Section 7.

3.2 Universality

The unrestricted calculus is computationally universal, and the route that the earlier essay takes through counter machines makes explicit which parts are proved. A counter machine has finitely many registers holding natural numbers and a finite program of increment, decrement-or-branch and halt instructions, and the partial recursive functions are exactly those computed by such machines (Minsky, 1967). The machine is compiled to a lambda term using Scott numerals (Mogensen, 1992), for which the predecessor is a constant-time operation, together with a fixed-point combinator suited to call-by-name evaluation, and the lambda term is translated into circles by bracket abstraction. Each machine step corresponds to a bounded number of weak contractions of the compiled term, which gives the halting direction.

Theorem 3.3 (Halting direction; status E)**.** *If the machine started on given inputs halts with output n in m steps, then the compiled term weakly reduces to a term representing n, and its circle translation reduces to the translation of the numeral $\overline{n}$. Applying that numeral to two fresh variables decodes it by a reduction to the first variable if $n = 0$ and to the second variable applied to a term whose normal form is the translation of $\overline{n-1}$ otherwise.*

The proof is an induction on m using a one-step lemma: a configuration represented by numerals weakly reduces, in a fixed number of contractions per skipped instruction and per register operation, to the representation of the next configuration, and every contraction used is weak because none lies under an abstraction. Uniqueness of normal forms, which follows from confluence, closes the decoding. The converse direction is not proved. A non-halting machine must yield an infinite weak-head evaluation of the compiled term, which an infinite weak reduction alone does not establish, since an argument may reduce forever when a head contraction would discard it. The bracket-abstraction translation must also reflect the absence of a weak-head normal form by the absence of a circle normal form on these compiled terms. Both statements are explicit adequacy hypotheses.

Corollary 3.4 (Undecidability for circle terms, conditional; status P)**.** *Under the two adequacy hypotheses, no algorithm decides normal-form existence for closed circle terms, and every partial recursive function has a circle representation whose defined inputs reduce to the translated output numeral and whose undefined inputs yield no normal form.*

The evidence in the non-halting case is a finite prefix: a program that increments a register forever produced no normal form under a large budget for either the lambda term or the circle term, and a program that halts exactly on even inputs decoded a numeral on the even inputs tried and exhausted its budget on the odd ones. Such evidence cannot be strengthened into a statement about non-termination. The simulation is also inefficient, at roughly a hundred

to five hundred circle contractions per machine step in the programs tried, and no efficiency is claimed. What universality contributes to the present argument is a negative fact with a positive consequence. No static analysis of a circle term can in general decide whether it terminates, so a bound on the resources a term may use has to come from outside the term. In the implementation that bound is fuel, and it is the first of the places where a guarantee does not come from the term.

4 Scoped Permission

The structural rules of a logic say what may be done with assumptions without regard to what the assumptions say: an assumption may be used twice, left unused, or reordered. In combinatory logic these become reduction rules, and in none of the settings above does the permission to copy depend on where in a term the copying occurs. The systems in this section add exactly that dependence. A term may contain bracketed subterms, each bracket carries a label, and each label carries a set of structural combinators that may act inside it.

Definition 4.1 (Permission structure and scoped terms)**.** A *permission structure* consists of a finite set Λ of labels, a root label $\varepsilon \notin \Lambda$, and a map $\pi : \Lambda \cup \{\varepsilon\} \to \mathcal{P}(\{\mathsf{W}, \mathsf{K}, \mathsf{C}\})$. Scoped terms are generated by $t ::= \mathsf{a} \mid c \mid \mathsf{sc}_\ell \mid (t\,t) \mid \langle t \rangle_\ell$ with $c \in \{\mathsf{I}, \mathsf{B}, \mathsf{C}, \mathsf{K}, \mathsf{W}\}$ and $\ell \in \Lambda$. A term without the constants sc_ℓ is a *core* term.

For an occurrence p in t, the *chain* of p is the sequence of labels of the brackets that contain p properly, read from the outside, and its *effective permission* is $\pi^*_t(p) = \pi(\varepsilon) \cap \pi(\ell_1) \cap \cdots \cap \pi(\ell_n)$. A scope can therefore restrict what its ancestors allow but never extend it. A *redex* is an occurrence of a constant of arity n together with n arguments attached by a spine of n consecutive application nodes with no bracket on the spine; this is the only place where brackets are opaque, since a bracket between a head and its arguments prevents the redex from forming. A step with head $c \in \{\mathsf{W}, \mathsf{K}, \mathsf{C}\}$ at occurrence p is *admissible* when $c \in \pi^*_t(p)$, and steps with head I or B are always admissible. The side condition depends on the position of the redex and not on the instantiation of its variables, which distinguishes it from the conditions of conditional rewriting (Meseguer, 1992). Two boundary rules move terms across brackets and are always admissible:

$$\text{(Abs)} \quad \langle u \rangle_\ell\, v \;\to\; \langle u\, v \rangle_\ell, \qquad\qquad \text{(Sc)} \quad \mathsf{sc}_\ell\, u \;\to\; \langle u \rangle_\ell.$$

Abs lets an argument enter a scope from outside, and Sc lets a term be placed in a scope by an application. The core relation is written $\to_\pi$ and the relation with the boundary rules $\to^+_\pi$.

Proposition 4.2 (Availability versus admissibility; status E)**.** *A rule may be available in the system and fail to be admissible at a redex. If $\mathsf{K} \in \pi(\varepsilon)$ and $\mathsf{K} \notin \pi(\ell)$, then $\langle \mathsf{K}\,\mathsf{b}\,\mathsf{c} \rangle_\ell$ contains an occurrence of the available combinator K at the head of a redex whose step is not admissible, and the term is normal.*

Lemma 4.3 (Monotonicity and chain extension; status E)**.** *If $\pi \le \pi'$ pointwise then $\to_\pi \subseteq \to_{\pi'}$, so every term strongly normalizing under π' is strongly normalizing under π. If a chain γ' extends a chain γ then $\pi^*(\gamma') \subseteq \pi^*(\gamma)$, so a term with no permitted redex at γ has none at γ'.*

Proof. Effective permissions are monotone in π and antitone in the chain, and admissibility is a membership test in the effective permission. □

Moving a term into a bracket thus always moves it in the favorable direction for normality. The lemma gives no control in the other direction: it does not say that a term remains insensitive, or that a pending boundary step inside it remains eligible. Confluence is the subject of the next subsection.

4.1 Confluence and the Boundary Rules

Theorem 4.4 (Confluence of the core system; status C)**.** *For every permission structure, $\to_\pi$ is confluent on core terms.*

Proof. Label each constant occurrence with its effective permission, introducing symbols c^Q for each constant and each $Q \subseteq \{\mathsf{W}, \mathsf{K}, \mathsf{C}\}$, and call a labelled term well labelled when every c^Q sits at a position with effective permission Q. Well-labelled terms correspond bijectively to core terms. The rewriting system has the rules for I^Q and B^Q for all Q, and the rule for c^Q with $c \in \{\mathsf{W}, \mathsf{K}, \mathsf{C}\}$ only when $c \in Q$. Each left-hand side is an application spine headed by one constant with distinct variables as arguments, so the system is left-linear, and two distinct left-hand sides have different head constants at the end of their spines, so there is no critical pair. A left-linear system with no critical pairs is orthogonal and confluent (Huet, 1980; Terese, 2003). A core step never changes the chain of any surviving constant, so the labelling is preserved and confluence transfers back. □

The proof shows what makes the core system tame: a step never changes the chain of any surviving constant. The boundary rules do change it, and the effect is decisive.

Theorem 4.5 (Boundary rules; status C)**.** *The system with Abs is confluent if and only if $\pi(\varepsilon) \subseteq \pi(\ell)$ for every $\ell \in \Lambda$, that is, if and only if no scope restricts the root permission. The same holds for the system with Sc. When the condition fails, some term has two distinct normal forms.*

Proof. If $\pi(\varepsilon) \subseteq \pi(\ell)$ for all ℓ, every effective permission equals $\pi(\varepsilon)$ and the system is a left-linear rewriting system over a signature with a bracket symbol for each label, with the core rules for the combinators in $\pi(\varepsilon)$ together with Abs and Sc for every label. The left-hand sides are a constant spine, a bracket applied to an argument, and a scope constant applied to an argument. Their heads differ and no proper non-variable subterm of one unifies with another, so the system is orthogonal. Conversely, suppose some ℓ and some $R \in \pi(\varepsilon) \setminus \pi(\ell)$. Take a redex u built from R and distinct atoms, for example $\mathsf{K}\,\mathsf{b}\,\mathsf{c}$ with contractum $u' = \mathsf{b}$, and let $t = \langle \mathsf{a} \rangle_\ell\, u$. By Abs, $t \to \langle \mathsf{a}\, u \rangle_\ell$, which is normal because u now lies in the chain ℓ where R is not admissible. On the other hand $u \to u'$ is admissible at the root, so $t \to \langle \mathsf{a} \rangle_\ell\, u' \to \langle \mathsf{a}\, u' \rangle_\ell$, which is also normal and differs from the first. A confluent system has at most one normal form per term. The argument for W, C and Sc is the same with the corresponding redexes. □

The theorem concerns confluence and uniqueness of normal forms, not determinism. What fails when a scope restricts the root permission is joinability: the same term reaches two normal forms according to whether an argument is reduced before or after it is absorbed. The permission governing a redex is a function of position, and a boundary rule can move a redex from a position where it is permitted to one where it is not. A theory that keeps both restrictive scopes and boundary crossing therefore has to compare sets of outcomes or restrict the boundary rules. Restriction is the second course, and the earlier essay carries it out for an abstract base relation.

4.2 Restricted Absorption and the Settled Scope Constructor

Absorbing only arguments that are normal for the base relation is not enough. For an abstract relation of a stated spine-based form, an explicit counterexample has a pending absorption inside the argument that creates a redex, which is then evaluated under the wrong permissions. Two restrictions do work. Absorbing only arguments that are normal for the combined relation, and absorbing only normal arguments that contain no permission-sensitive constant, each preserve confluence (status C), and these are the modes of absorption that the implementation names settled and insensitive. The same idea applies to the scope constructor, and it is here that the implementation extended the earlier results.

Definition 4.6 (Settled)**.** A term v is *settled at* a permission m if it contains no permitted contraction at its own effective permission, which is m at the top and the intersection with $\pi(\ell)$ inside a bracket labelled ℓ, and no application whose function child is a bracket or a scope constructor.

A settled term is frozen. If v is settled at m and $m' \subseteq m$, then v is settled at m', because permission is a membership test and masks only shrink and the second clause of the definition does not mention the mask. Consequently no redex of any rule occurs anywhere in v at m or at any smaller permission. A step elsewhere in the term touches disjoint positions, and a step may copy, discard, or carry a settled argument across a bracket without altering it. The only material that ever changes chain is material that the side conditions have already required to be settled. Let Abs_s and $\mathrm{Sc}_{\mathrm{set}}$ denote the boundary rules restricted to fire only when the argument is settled at the effective permission of the redex.

Theorem 4.7 (Settled scope; status E, paper proof, unreviewed)**.** *For every permission structure and every gated set, the system consisting of the contractions,* Abs_s *and* $\mathrm{Sc}_{\mathrm{set}}$ *has the diamond property for parallel reduction, and its reduction relation is therefore confluent. This holds for terms that do not terminate.*

Proof sketch. Define parallel reduction in the usual way, with every side condition evaluated on the term before the step, and let t° be the complete development of t. A settled argument develops to itself because it is frozen. The Takahashi property, that $t \Rightarrow u$ implies $u \Rightarrow t^\circ$, is proved by induction on the derivation, by cases on whether the root of t is a redex and, if so, whether u arises by reducing children only or by contracting the root. In the first case the redex is reproduced in u with the same head, the same arity and the same chain, and its argument, if settled, is unchanged; in the second case the contractum is the same as the one in t° after the induction hypothesis is applied to the arguments, using the fact that each right-hand side is built from its arguments by application alone. If $t \Rightarrow u$ and $t \Rightarrow v$ then both reduce to t°, and since $\to \subseteq \Rightarrow \subseteq \to^*$ the strip lemma gives confluence of $\to$ (Takahashi, 1995). □

The argument is that of an orthogonal system with one addition: the left-hand sides have different heads, so no two overlap at a root, and the only interaction between rules is through the side conditions. Those side conditions inspect the argument and nothing else, and when the argument is frozen every step either leaves the condition unchanged or removes the redex that carries it. The status of the theorem needs stating plainly. The proof is a paper proof by one author. It has not been machine-checked and has not been reviewed by anyone else. A computational record agrees with it (status T): the diamond held for every pair of parallel successors of every term in each of three sets, namely all 410,960 terms with at most four

leaves and two brackets and one label (6,664,432 pairs), all 1,366,969 terms with at most five leaves and one bracket (31,696,480 pairs), and all 368,430 terms with at most three leaves and three brackets and two labels (28,639,888 pairs), over the symbols $\mathsf{a}, \mathsf{K}, \mathsf{C}, \mathsf{W}, \mathsf{I}, \mathsf{sc}_x$ and, with two labels, sc_y, under every assignment of subsets of $\{\mathsf{W}, \mathsf{K}, \mathsf{C}\}$ to the labels with full root permission (Flyxion, 2026). Non-terminating terms such as $\mathsf{W}\,\mathsf{W}\,\mathsf{W}$ are included and no run was cut off by a cap. The check was given two positive controls and one mutation: it fails for the earlier essay's unrestricted Sc at $\mathsf{sc}_x(\mathsf{K}\,\mathsf{a}\,\mathsf{a})$, it fails for the safe forms at identified patterns, and when the settled condition of Sc was weakened by hand to plain normality it reported 37,440 failing pairs at four leaves and two brackets. The search is bounded by term size, symbol set and label count, and the proof does not depend on it.

The cost of the restriction is real. Because $\mathsf{sc}_\ell\, u$ waits until every boundary step inside u has run, a scope whose argument holds a boundary step that never fires stays inert, which matches what a blocked Abs already does.

Proposition 4.8 (The safe form; status P)**.** *Whether the scope constructor in its safe form, which fires when its argument is normal and either settled or insensitive, is confluent is open, and so is absorption in the insensitive mode in combination with Sc.*

The proof route of Theorem 4.7 fails for the safe form at an identified pattern. In $\mathsf{sc}_x\,(\langle \mathsf{I} \rangle_x\, \mathsf{a})$ with I ungated, the argument is normal and insensitive, so safe Sc may fire, but the argument also contains a pending Abs whose firing creates the contraction redex $\mathsf{I}\,\mathsf{a}$ and blocks the outer Sc until that redex has run, so the two orders need a different number of parallel rounds and the diamond fails. This is a failure of one proof method and not a counterexample. Local confluence of the full relation, with both kinds of step on both legs and joins searched to depth eight, held for every peak in the recorded runs, for insensitive absorption with safe scope, including 81,616 peaks between contractions, 309,024 between boundary steps and 433,944 mixed peaks at four leaves and two brackets with one label, and an exhaustive search of normal forms found no term with two of them across about 53.8 million explorations (status T). An exploration is a term together with one assignment of permissions to its labels, and 2,208 explorations in each of the two safe modes with absorption hit the cap of 600 states or 40 circles and were counted as capped and not as passes; they contain W loops, so the unexplored region is the non-terminating one, which is also where the proof route is silent. Local confluence does not imply confluence for a system that does not terminate, since Newman's lemma needs termination (Newman, 1942). The evidence is evidence of absence of a counterexample in a bounded region and nothing more.

4.3 Descent of Admissibility

Proposition 4.9 (Descent; status E)**.** *Along any reduction with core steps and the boundary rules, the effective permission of every surviving constant occurrence can only shrink. Consequently the scope structure of the initial term bounds the structural fragment that is ever realized without determining it.*

Proof. A core step rearranges its arguments at the chain of its own root, so every surviving constant keeps its chain. Abs carries an argument into a bracket and Sc carries its argument into one, and in both the chain is extended by the label. By Lemma 4.3 an extended chain has a smaller or equal effective permission. □

The proposition is what makes a scope usable as a guarantee. A term that begins inside a scope that withholds contraction cannot acquire the ability to duplicate by any sequence of

steps, whatever the program does. The implementation relies on this fact in its design but has no conformance vector for it, because tracking the identity of a constant through duplication needs a term-graph representation that is not yet implemented.

4.4 Copying, Sharing and Erasure

Two further results of the earlier essay concern representations and are stated here without proof. In scoped term graphs the equivalence of copying and sharing holds under a coherence condition on the permissions seen by shared nodes; the condition is sufficient and not necessary, and once it fails the tree system and the two graph policies, one admitting a step when some access path permits it and the other when every access path does, produce three different sets of normal forms (status E). The coherence condition is the reason the cost accounting of Section 3, which separates steps from distinct nodes, cannot be adopted by an implementation without checking what a shared node sees. In a more general account of boundary-governed transition systems, erasing boundaries preserves every admissible reduction, fails to reflect admissibility exactly when some scope restricts the root permission, and fails to lift reductions even when no permission is restricted, because erasure can expose redexes that the boundaries interrupt. Loss of permission and loss of exposure are independent obstructions. These results matter for the thesis in one respect: they show that a scope is not an annotation that can be stripped before execution, since the scoped system and its erasure are different systems.

5 The Reference Implementation

The formal core is accompanied by a reference implementation in Go, the SpherePOP kernel (Flyxion, 2026), whose purpose is to make the claims above executable and to produce records that can be checked. Its architecture enforces a separation by imports and not by convention: a sphere is a structural object, a pop is a rule-governed reduction, and a host, whether a game entity, a test or a command line, is only a place where the computation can run. The package `term` holds immutable, content-addressed circle terms with a parser, a printer and positions, and imports only the standard library. The package `reduce` holds the rule table, the permission structure, the strategies and bounded evaluation, and imports `term`. The package `trace` holds trace documents, the hash chain and the independent checker, and imports `term` and `reduce`. The package `sched` holds a deterministic, fuel-bounded scheduler and its log checker. The package `effects` holds the host boundary, and the packages `encode`, `lambda`, `driver` and `arena` hold data encodings, a front end from lambda expressions, a loop for persistent programs, and a headless example world. The reducer runs with no game engine, no graphics and no network gateway, and nothing in `reduce` depends on the packages that use it.

Evaluation is bounded by fuel, a count of steps granted by the caller, and a run that exhausts its fuel says so in its record. Four strategies are provided, leftmost-outermost, leftmost-innermost, rightmost-innermost and rightmost-outermost, each a function of the current term. The permission structure assigns subsets of $\{\mathsf{W}, \mathsf{K}, \mathsf{C}\}$ to labels as in Section 4, and the settled forms of absorption and scope of Theorem 4.7 are selectable modes. Two defaults must be kept apart. The reducer's own default is the rule of the earlier essay, with absorption in the insensitive mode and an unrestricted scope constructor, and that scope constructor is not confluent in general by Theorem 4.5. The default for hosts, in the scheduler's host configuration and in the simulation and arena commands, is settled absorption with settled scope, the combination of

Theorem 4.7. The choice was made by the owner of the language, and the tests do not make it.

A crude measure of size gives some feeling for the scale of what must be trusted. At the commit from which the figures in this essay were taken, the non-test Go source comes to 708 lines in `term`, 985 in `reduce`, 312 in `trace`, 559 in `sched` and 748 in `effects`, and an independent verifier written in Python from the specification of the evidence formats comes to 1112 lines. The corresponding test code is larger than the code it tests in several packages, 1942 lines against 985 in `reduce`. Line counts are an indication of scale and nothing more: they do not say which functions the checker reaches, and Section 9 supplies a count of that kind under a stated rule.

6 Evidence: Computation, Record and Acceptance

The architectural principle of the implementation is that three things stay apart: the computation, the evidence that it took place, and the decision to accept it. A worker reduces a term. Its record is a trace, a JSON document containing the format identifier, the configuration in canonical form, the start term and its digest, the strategy and fuel the worker claims, for each step the rule, the position and the digest of the whole term after the step, the claimed final term and its digest, whether fuel was exhausted, and the head of a hash chain that links each step to its predecessor. A checker parses the start term and the configuration afresh, verifies the digest, and replays each step with the rule table. It rejects a step whose rule is not in the table, whose rule does not match the subterm at the cited position, whose rule is not admissible at that position under the effective permission recorded in the configuration, or whose result digest differs from the claim. It then verifies the head of the chain, the final term, the termination status, and that the step count does not exceed the stated fuel. A claim of exhaustion must use the whole budget. Unknown fields are rejected.

Proposition 6.1 (What a successful check establishes; status T)**.** *If the checker accepts a trace, then every step is an instance of a rule in the table at a position where that rule is permitted under the recorded configuration, and the recorded final term is what that sequence produces from the recorded start term.*

The statement is conditional on the correctness of the shared code, and its status is tested for that reason: it is established by construction, and checked by a corpus described below, and not proved. What a successful check does not establish is as important. Matching final digests alone prove nothing, and the checker does not rely on them. A checked trace does not show that the result is the unique normal form, since confluence is a property of the system and the configuration and not of a trace, which is why the restricted modes of Section 4 exist. Without the strategy requirement the trace does not show which order the worker used, because agreement on a result is different from agreement on how it was found. The hash chain gives integrity and order within the document, but a forger can recompute it, so tamper evidence holds only relative to an anchor held outside the document, such as a head digest published or signed elsewhere. Because a worker states its own fuel, a truncated run can be made self-consistent by restating the fuel, so a host that has agreed a budget passes it to the checker, which then requires the document to state exactly that. The agreed budget is held by the host and not carried by anything in the record. The checker shares the rule-application routine with the reducer, so an error in the definition of a rule would be shared by both, and a second implementation written from the specification is the way to remove that dependence.

Finally, the trace does not say that anyone should accept or commit the result. Acceptance is a separate decision made by whatever policy the host holds, and the reducer deliberately has no notion of commitment, so that a verifier gains no authority from being able to replay.

The second implementation exists, and its limits are stated. The specification of the evidence formats is one document, and two interpreters of it, the Go packages and the Python verifier, are run over a conformance corpus of 455 records, of which 222 are trace documents, 109 are scheduler logs and 124 are effect logs. The invalid records are built by single mutations. A record that is tampered with and not resealed tests the chain. The same tampering resealed, so that every hash verifies, tests the semantic check behind the chain. Several scheduler and effect cases are built so that only one check can reject them, because a check that is also covered by a later one cannot be told from a missing check. A random differential test generates reduction problems, has the Go engine write a trace that the Python verifier checks, has the Python reducer write a trace that the Go checker checks, and compares the two traces step for step, since a strategy is a function and the traces must be identical. A run of 300 generated cases with seed 1 on a fresh checkout of the repository found no disagreement (status T). Building the corpus exposed defects that no existing test had caught: a scheduler log with no entities skipped the final-state check, a label with an empty permission was written as a null and then rejected by the reader that had just been made strict, the hash of an effect log depended on how Go escapes strings, JSON decoding accepted case-variant names, missing members and duplicates, and the specification contradicted the code on the scope-constructor clause of settledness and on the default scope mode. Both verifiers and the specification have one author, and the specification was drafted from the behaviour of the Go code, so an error shared by all three would pass. A second human implementer reading only the specification is the missing test.

7 Scheduling and Effects

Two further components extend the same separation to concurrency and to the world. Neither adds a rule to the calculus.

7.1 The Scheduler

The scheduler runs several terms under a deterministic, fuel-bounded schedule, and its authority is exactly a strategy and a fixed budget. It hands every reduction to the reducer and cannot alter the rule table or the configuration after a world is built. Entities run in byte order of their identifiers, whatever order they were supplied in. In each tick every unfinished entity receives the policy's budget and reduces from wherever it stopped, the continuation being simply its current term. An entity in normal form is skipped and spends nothing, so a non-terminating entity cannot starve the others. The log separates scheduler decisions, which are events, from reduction steps, which appear only inside the slice document attached to a run event, itself an ordinary trace. The log checker verifies three things independently: that the decisions follow the policy, that every slice passes the trace checker with the strategy enforced and the fuel fixed at the budget, and that the slices chain, each starting where the entity's previous slice ended.

Proposition 7.1 (Slicing invariance; status T)**.** *Holding the total fuel of an entity constant at 60 and varying the per-tick budget over 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30 and 60 gives identical final states, equal to the unsliced reduction.*

The reason is that a strategy is a function of the current term, so slicing a run and resuming it reproduces the unsliced run, and this is by design as well as tested. Scheduling alone therefore cannot change a result, entity order does not change any entity's outcome, and the same world under two different strategies on a non-confluent configuration does disagree. A related tested statement concerns suspension. An entity that suspends at a root marker and resumes reaches the normal form of the uninterrupted term with the markers removed: on 600 generated terms, each run under four strategies and four budgets, which is 9,600 scheduler runs of which 640 resumed at least once, the final term always equalled the stripped uninterrupted normal form, under the settled host configuration only. The mathematical statement is an unwritten corollary of confluence and the observation that a root marker stays a marker under every step that can be taken from it, and is open as a theorem. Two runs with the same result can have different suspension histories, which is shown by example: for $\mathsf{I}\,(\mathsf{yield}\,(\mathsf{K}\,\mathsf{a}\,\mathsf{b}))$ with a budget of ten, leftmost-outermost suspends after one step and spends one further step after resuming, while leftmost-innermost suspends after two steps and spends none after resuming, and both end at `a`. The history is not invariant under strategy, and no claim is made that it is invariant under anything.

7.2 The Effects Boundary

A program can describe an effect and cannot perform one. Three relations are kept in different places. Reduction, $t \to^* n$, belongs to the reducer. Extraction of proposals from a normal form belongs to the effects package. The transition of the world together with its recorded dispositions belongs to the host. The host reads one designated position and only after reduction has reached a normal form: the normal form must be an application whose head is the atom `effects`, with one request per argument, in order. A normal form of any other shape is an output with nothing to consider. A reduction that ran out of fuel is never read, even when its partial term happens to look like an output, because a partial term is not a result. Reading a normal form and not an intermediate term is what makes effects independent of strategy: an effect inside a subterm that K later discards is never proposed, and one that W duplicates is proposed twice, whichever strategy got there, and tests exercise both.

A request has the form `effect (kind id) verb arg...`, checked against a schema supplied by the host that lists, for each target kind, the type of its identifier and the verbs it accepts. A request that is well formed as a term but not an instance of the schema is malformed, and the decoder never reduces and never panics. Each request is decoded, authorized and committed on its own, in the order the program listed them. The authorizer is an interface asked after decoding and before the world is touched, and the supplied policy fails closed: only explicit grants of a subject to a verb on a kind are allowed. Every outcome, whether committed, denied, failed, malformed, absent, exhausted or truncated, is appended to a hash-chained log that can be verified against an external anchor and against the schema. A cap on the number of requests per output stops a program from handing the host an unbounded list.

The experiment that justifies the design is small. Two entities propose conflicting changes to one door. Their programs reach the same normal form under all four strategies and under every subdivision of the same total fuel tested, so the proposals are identical in every run. The committed state of the door then depends only on the order in which the host processes the two entities, while the recorded dispositions are the same in both orders. In this setting a difference in committed world state can be attributed to host ordering and not to the calculus, the strategy

or the frame subdivision. A verified effect log establishes that the chain is intact, that the entries are consistent with the schema and, against an anchor, that nothing was removed from the end. It does not establish that the authorizer decided correctly, since checking that would need the policy and the inputs replayed, and it does not establish that a committed request was applied faithfully by the world. A program may also group requests as one atomic unit, which the host logs as a single entry and applies in full or not at all if its world supports it, and fails otherwise; the group is a host convention over inert data, so reduction is unchanged.

8 What the Term Carries and What a Supervisor Keeps

The three questions of the introduction can now be answered separately for this system, and the answers are not the same.

The licence to transform is carried by the term in a qualified sense. The term supplies the labels and their nesting, and the configuration, which the host agrees and which is written into every record, supplies the table from labels to permissions. A step is admissible only if the permissions of the enclosing scopes include the structural combinator at its head, the permission is a function of position and nothing else, and by Proposition 4.9 it can only shrink along a reduction. A term that starts inside a scope withholding contraction cannot acquire the ability to duplicate, whatever the program does and whatever order its steps are taken in, and no process watches for this because the condition is part of the rule that performs the step. Under the settled configuration of Theorem 4.7 the system is also confluent, so the order in which admissible steps are taken cannot change the normal form when one is reached. The scheduler therefore needs no authority over results: it chooses an order, and every order is equivalent where it matters. This is the content of the claim that scope does the work that a result-preserving supervisor would otherwise do, and it is the part of the thesis that survives scrutiny without qualification beyond those already stated, namely that the proof of Theorem 4.7 is unreviewed, that the guarantee covers W, K and C and not other resources, that gating S itself is an option outside the theorems, and that a scope whose argument holds a boundary step that never fires stays inert.

The guarantee has a limit that deserves to be said directly, because it is easy to attribute to scope more than scope does. A supervising process is expensive in part because the actions it must mediate are varied: files, sockets, memory, clocks. The calculus has none of these. A program in it can only rearrange terms, and it can only describe an effect by producing a term of a certain shape at a designated position after a normal form is reached. Much of the economy of the design therefore comes from the absence of ambient authority in the language and not from the scope permissions, which address only the structural operations that the language does have. The constraint-first character of the design is real, but the constraint that does most of the work is the narrowness of what a term can do, and scope permissions refine it at the one place where the language is expressive enough to be dangerous, which is duplication and discarding.

The licence to accept is not carried by the term and is not meant to be. A replayed trace establishes that each step was an instance of an admissible rule and that the claimed result follows. It does not establish that the result is the unique normal form, that the worker followed any particular strategy unless that is demanded, or that anyone should commit the result. The reducer has no notion of commitment, so that being able to replay confers no authority. The same holds for effects. They are read from a normal form, which makes them independent of

strategy, and they are passed one at a time to an authorizer held by the host that fails closed. The authorizer's decisions are recorded but cannot be checked from the record, since checking would need the policy and its inputs.

Three things that a supervisor usually supplies are not supplied by the term, and the first of them is forced by the mathematics. The bound on resources has to come from outside, because the unrestricted calculus is universal and no static analysis decides termination, so fuel is a quantity agreed by the host and demanded of the record by the checker; a term cannot state its own bound honestly, and a worker that states its own fuel can make a truncated run self-consistent. Memory is the second. The reducer memoises on term digest and effective permission, the tables grow with the number of distinct subterms visited, and they are bounded only by fuel, which bounds steps and not size, as the sharing example of Section 3 shows; this was not audited. The third is the anchor. A hash chain gives integrity and order within a document and nothing against a forger who recomputes it, so evidence that must survive an adversary needs a head digest held elsewhere, and the scheduler log has no agreed external anchor yet. To these should be added the choice of configuration, since the reducer's own default is not confluent and a host that accepts it gets none of the guarantees of Theorem 4.7.

Put as a division of labour, the term and its record carry the licence to transform and the means of checking that it was exercised, and the host keeps the budget, the anchor, the memory bound, the choice of configuration, the decision to accept and the decision to act. That is a smaller claim than the one with which the project began, which was that scope could replace the supervisor. What it can replace is the part of the supervisor's work that consists of watching a worker rearrange terms, and it is possible to say exactly which part that is.

9 The Supervised Baseline

The comparison of the preceding section was structural until a baseline was built to measure it. The baseline sits beside the scoped design in the kernel repository as a package `supervised` and a command `supexp`, and it modifies no existing file. It reuses the rule-application routine of the reducer under a flat configuration that permits everything, so that nothing in the term or in the configuration can stop a step. A worker proposes, in the order of the claimed strategy, every step whose shape matches a rule, and it looks at structure alone. A supervisor rules on each proposal. The supervisor is a second implementation of the admissibility conditions, written from the specification, covering normality, settledness, insensitivity and the absorption and scope modes, and it shares no code with the reducer other than the routine that applies a step. Every decision is appended to a hash-chained audit log, refusals included, recording the rule, the path, the verdict, the effective permission at the position and the digest of the term afterwards. Fuel counts accepted steps, and once the budget is spent a silent probe decides whether a permitted step remained. The log has two checkers. One verifies the chain and the log's own bookkeeping and replays nothing. The other replays: it re-decides every entry against the policy, applies the permitted steps, compares digests and status, and, when asked, demands that the entries are exactly the proposals the claimed strategy would make.

The corpus is fixed by construction and was run with a harness that is part of the patch. It has 45 configurations, formed from three permission structures (one in which the labels restrict the root permission, one in which the root permission lies inside every label's permission, and one in which S is gated as well), five absorption modes and three scope modes. It has 1027 terms, of which 27 were chosen by hand to include the cases of Sections 3 and 4 and 1000 are

random terms of depth five from a fixed seed. Each term was run under all four strategies with a fuel of 60, giving 184,860 runs with 1,078,204 accepted steps and 573,094 refused proposals. No run differs between the designs in outcome or in the sequence of accepted steps. At each of the 1,263,064 states visited, the steps the scoped reducer offers and the steps the supervisor permits are the same list in the same order, and every honest record is accepted by every checker. The agreement is what independent implementations of one specification should show, and it establishes interchangeability on this corpus and not correctness of either. As a positive control, a supervisor deliberately made to read the absorption mode s as nf disagrees with the reducer in 806 of 36,972 runs, so the comparison is capable of failing.

The trusted code was counted by a rule fixed before the count: the function and method declarations reachable by name from the function that accepts an outcome, within its own package and the module packages it imports, counting non-blank non-comment lines and excluding tests. The scoped acceptance path, rooted at the trace checker, reaches 896 lines. The replaying checker of the supervised design reaches 1195, and the checker that verifies only chain and bookkeeping reaches 496 but, as the next paragraph shows, cannot detect a wrong decision. Both designs depend on the same 322-line closure of the rule-application routine, so beyond it the figures are 574 and 873. The first of the outcomes that would have counted against the thesis, that the supervised design needs no more trusted code once its permissions are data checked by replay, therefore did not occur. The reading of the difference matters, however. The extra lines are a second implementation of admissibility and a worker that enumerates proposals, and they buy independence: an error in the reducer's admissibility is shared with the scoped checker, while the supervisor would not share it. The scoped design can buy the same independence by writing a second implementation, as the Python verifier does for the evidence formats, at a similar cost. The count shows how many times the admissibility conditions are written down on the acceptance path, and not where the permission is held.

The tampering matrix was applied to twelve base runs, one for each of twelve configurations spread over the corpus, each the first run with at least three accepted steps and a refusal under a fuel of 14. The harness reproduces the honest records byte for byte and every checker accepts them. Tampering that leaves the hash chain broken is rejected by all three checkers in all nine variants. The informative rows are those in which the tamperer recomputes the chain, as the format permits, and those that concern the audit log alone, summarised in Table 1.

Three findings follow from the table. The first is that replay of the audit log detects every mutation that the scoped trace checker detects, and nothing is caught by the supervisor's watching that replay of a record does not catch, so the second outcome that would have counted against the thesis did not occur either. The checker that does not replay is the place where the chain-recomputing tamperer succeeds, and the cases in which it rejects sealed mutations (4 of 12 for deletion and duplication, 8 of 12 for the exhaustion flag) are the base runs that exhausted their fuel, where bookkeeping alone betrays the change. The second finding is that the supervised design has a surface the scoped design lacks. A refusal entry can be deleted without detection unless the replay is told the claimed strategy, because the log is then no longer the list of everything the worker proposed, and a worker that omits its refused attempts leaves a log that every other check accepts. The scoped design has no refusal entries to omit because it never offers the forbidden step. That is also a loss, which should be stated plainly: the supervisor's log records what the worker tried, and the scoped record does not, so a host that wants evidence of attempted violations obtains it only from the supervised design. The third finding concerns the last substantive row of the upper block. An honest run under a

Mutation, chain recomputed	Scoped check	Log only	Replay
rule changed	12/12	0/12	12/12
path made invalid	12/12	0/12	12/12
result digest wrong	12/12	0/12	12/12
accepted step deleted	12/12	4/12	12/12
accepted step duplicated	12/12	4/12	12/12
two accepted steps swapped	12/12	0/12	12/12
exhaustion flag flipped	12/12	8/12	12/12
forbidden step forged as accepted	12/12	0/12	12/12
steps not those of the claimed strategy	8/8	0/12	12/12
honest run under a loosened configuration	0/12	0/12	0/12
Mutation of the audit log only			
refusal entry deleted	-	0/12	0/12
refusal entry deleted, strategy required	-	0/12	12/12
acceptance recorded as a refusal	-	0/12	12/12
mask altered	-	0/12	12/12
refusal inserted for a permitted step	-	0/12	12/12

Table 1: Records rejected out of records to which the mutation applied, for twelve base runs. The scoped check requires the claimed strategy in the strategy row, as does the replay of the audit log in the strategy row and in the row marked strategy required. Final text, final digest and fuel mutations are rejected by all three checkers in every case and are omitted.

configuration looser than the one agreed is accepted by all checkers of both designs, because each checks against the configuration the record states, and it is rejected in all twelve cases by a comparison of the recorded configuration with the agreed one, which is the same few lines in either design. Both designs therefore keep the permission table in the configuration, and the host must hold it. This is the reason for the qualification made in Section 8: the scoped design does not place the permission in the term, it places the labels there and leaves the meaning of the labels to a table that the host agrees, exactly as the supervised design leaves the map from positions to permissions to a policy that the host agrees.

Cost per accepted step was measured on three workloads, as the median of nine repetitions on one core. For a nonterminating loop with no refusals the scoped run, including construction of its record, took 1.47 microseconds a step and the supervised run 1.74. Beside a forbidden K that is refused at every step, the supervised run took 3.76 against 1.92, because the worker proposes the forbidden step and the supervisor logs the refusal at every state while the reducer discards the subterm at no cost after the first inspection. For a term that grows without a cycle the two took 21.0 and 22.3. The cost of checking a record was equal within noise for the scoped checker and the replaying checker, at 1.01 against 1.05, 1.17 against 1.99 and 16.5 against 16.5, the second workload again being the one in which refusals dominate, while the checker that does not replay was cheaper throughout and detected the least. The supervised run therefore costs between 1.06 and 2.0 times as much, and the larger factor arises when refusals are frequent. These timings come from a single machine and should be read as an ordering and not as constants.

The experiment has limits that bear on how far it can be pushed. One author wrote both implementations of admissibility from the same specification, so the independence is of code and not of understanding. The worker is naive, in that it enumerates every proposal in order with only a structural memo, and a cleverer worker would narrow the cost gap. The rule-application

routine is shared, and a fault in it would pass through both designs unnoticed. The line counts are by name and over-approximate where a method name is shared, and they count functions only. The twelve base runs and the small terms of the corpus say little about large or adversarial programs, and nothing in the experiment bears on the duties that the preceding section assigns to the host in either design: the budget, the anchor, the memory bound, the choice of configuration and the decisions to accept and to act.

Taken together, the results reduce the thesis to a narrower form than the one it had before they were available. The claim that scope replaces a watching supervisor in detecting illegitimate steps is not supported, because a replaying checker over an audit log detects the same mutations, and the claim that the scoped design places the permission in the term must be qualified as above. What the measurements support is smaller. The scoped design has a shorter acceptance path for the same detection power, it is cheaper per step when forbidden steps are frequent, and its record has fewer places in which tampering can hide, and it obtains these properties by never offering the forbidden step, at the price of not recording attempts. The third outcome that would count against the thesis, a counterexample to the confluence of the settled constructor or an error in its proof, remains untested by this experiment and remains open.

10 Open Obligations and Conclusion

The open obligations are of three kinds. The mathematical ones are the confluence of the safe scope constructor and of insensitive absorption in combination with Sc, for which the diamond route fails at an identified pattern and bounded searches find no counterexample; the review or machine checking of the proof of Theorem 4.7, which has a single author; the discharge of the two adequacy hypotheses that stand between Theorem 3.3 and Corollary 3.4; and the independence of exchange from the other structural combinators, which is not claimed. The resemblance between permission-bearing scopes and the exponential modalities of linear logic (Girard, 1987), and between the structural combinators and the stack words of concatenative languages that copy, discard and exchange, is noted and unexplored. The implementation obligations are the term-graph representation with sharing and the coherence check of Section 4, without which descent has no conformance vector and heavily shared terms will strain a tree walk with memoisation, a bound on the size of logs, an external anchor for the scheduler log, and the audit of memory growth. The evidential obligation is the one that the conformance corpus names for itself: a second human implementer reading only the specification of the evidence formats.

The conclusion is deliberately narrower than the introduction promised. A combinatory calculus with scope-conditional structural rules gives the host a licence to transform that lives in the term, shrinks monotonically, and, in the settled configuration, leaves the result independent of order. A replayable record lets anyone check that the licence was exercised, and a fuel bound, which cannot come from the term, limits what the exercise costs. The licence to accept a result and the licence to act on it stay with the host by design, and the guarantees that depend on a budget, an anchor, a memory bound and a choice of configuration stay with the host by necessity. Measured against a supervised baseline built from the same reducer, the scoped design detects no more than a replaying supervisor does, and it differs from it in being shorter on the acceptance path, cheaper per step when forbidden steps are frequent, and free of refusal entries that can be omitted, at the price of not recording what the worker attempted. Whether that difference is large enough to matter depends on how much the host values a small

acceptance path and a compact record against evidence of attempted violations, and on the open questions of Section 10, which the experiment does not touch.

References

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*, revised edition. North-Holland, Amsterdam.

Church, A. (1941). *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton.

Curry, H. B. and Feys, R. (1958). *Combinatory Logic*, Volume I. North-Holland, Amsterdam.

Girard, J.-Y. (1987). Linear Logic. *Theoretical Computer Science*, 50(1):1–102.

Hindley, J. R. and Seldin, J. P. (2008). *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, Cambridge.

Huet, G. (1980). Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27(4):797–821.

Flyxion (2026). The SpherePOP Kernel: A Reducer, Trace Checker, Scheduler and Effects Boundary in Go. Software repository, github.com/standardgalactic/spherepop-kernel, commit 511f63e, 28 October 2026.

Lambek, J. (1958). The Mathematics of Sentence Structure. *American Mathematical Monthly*, 65(3):154–170.

Meseguer, J. (1992). Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155.

Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs.

Mogensen, T. (1992). Efficient Self-Interpretation in Lambda Calculus. *Journal of Functional Programming*, 2(3):345–364.

Newman, M. H. A. (1942). On Theories with a Combinatorial Definition of Equivalence. *Annals of Mathematics*, 43(2):223–243.

Takahashi, M. (1995). Parallel Reductions in Lambda-Calculus. *Information and Computation*, 118(1):120–127.

Terese (2003). *Term Rewriting Systems*. Cambridge University Press, Cambridge.