Local Symbolic Swarm

A Formal Specification of Certified Rewriting
over Immutable Workspace Snapshots

Flyxion
Independent Researcher

Version 0.1 November 2026

Abstract

This document specifies the Local Symbolic Swarm, a system in which many cooperating local processes transform a workspace of symbolic documents by proposing and certifying rewrites, while a single coordinator alone decides what becomes part of the authoritative history. The authoritative state is a chain of immutable, content-addressed snapshots with one current head. Search is deliberately unconstrained: a worker may use any procedure its resource budget permits. Trust is placed instead in a small checker that validates proof objects over a fixed symbolic language of partial rational-valued expressions, and in a coordinator that applies a separate progress rule and serializes publication through an atomic replacement of the head pointer. The specification fixes an abstract state model, a canonical concrete grammar with hash-based object identity, the semantics of expressions and of the proof language, a strictly decreasing progress measure, the filesystem primitives that give publication its atomicity, four coordination algorithms, and a failure model with explicit guarantees. Soundness of the baseline rule catalog and of the proof language is established, a bound on the number of successful commits is derived, and a set of conformance cases is given together with reference identifiers computed for the worked trace. The design separates recognition, execution, verification and publication so that each has an independent meaning and an independent failure mode.

Contents

1 Introduction **3**

2 Conventions and Notation **4**

3 Execution Model **5**
3.1 Roles 5
3.2 The Recruitment Sequence 6
3.3 Deployment and Trust 6

4 Abstract State **6**
4.1 Objects, Snapshots and the Head 6
4.2 Configurations and the Rule Catalog 7
4.3 Jobs, Attempts, Candidates and Certificates 7
4.4 Transactions 8

5 Canonical Grammar and Object Identity **8**
5.1 Lexical Classes . . . 8
5.2 Grammar . . . 8
5.3 Well-Formedness Conditions . . . 9
5.4 Unambiguity, Typing and Identity . . . 9
5.5 Reference Structure . . . 10
5.6 Paths . . . 11

6 Semantics of Expressions, Rules and Proofs **11**
6.1 Denotation . . . 11
6.2 Contexts and Equivalence . . . 12
6.3 The Baseline Rule Catalog . . . 12
6.4 The Proof Language . . . 13
6.5 The Checker . . . 14
6.6 Support Records . . . 15

7 Progress and Rule Safety **15**
7.1 The Progress Measure . . . 15
7.2 Overlap and Order Dependence . . . 16
7.3 A Timing-Independent Commit Policy . . . 16
7.4 The Meaning of No-Match . . . 16

8 Spool and Publication Primitives **16**
8.1 Layout . . . 16
8.2 The Primitive Put . . . 17
8.3 Replacing the Head . . . 17
8.4 Durability Profiles . . . 18
8.5 Claims, Sessions and Fencing . . . 18
8.6 Retention and Cleanup . . . 19
8.7 Announcements . . . 19

9 Algorithms **19**
9.1 Algorithm A: Observe and Recruit . . . 19
9.2 Algorithm B: Attempt a Rewrite and Obtain Support . . . 20
9.3 Algorithm C: Verify and Commit . . . 21
9.4 Algorithm D: Recover and Determine Quiescence . . . 22

10 The Coordinator Transition Model **22**
10.1 State . . . 23
10.2 Transitions . . . 23
10.3 Invariants . . . 23

11 Quiescence and Terminal Classification **25**
11.1 Job Status at a Snapshot . . . 25
11.2 Classification . . . 25

12 Failure Model and Guarantees **26**
12.1 Resource Limits 26
12.2 Crash Analysis 26
12.3 Guarantees 27

13 Conformance **28**

14 Relation to Prior Work **30**

15 Limitations and Extensions **30**

A Reference Vectors for the Worked Trace **31**

1 Introduction

A common pattern in automated symbolic work is that discovery is cheap and unreliable while acceptance must be cheap and reliable. A procedure that searches for a simplification may be heuristic, randomized, resource hungry or simply wrong, whereas the decision to alter a document that other work depends upon must rest on something that can be checked in a bounded and well understood way. The Local Symbolic Swarm applies this asymmetry to a single machine. Workers search, a checker verifies, and a coordinator publishes. No component is asked to be both creative and authoritative, and the protocol that connects them is written so that duplicated, delayed, lost or stale work cannot damage committed state.

The recruitment structure follows the sequence observed in immune responses. A sentinel observes a change and publishes a fact about it. A worker recognizes a task it is able to attempt. A claim reserves one attempt at that task, so that effort is not wastefully multiplied. The worker then crosses into a private staging area and produces a candidate, a step that corresponds to execution. A checker establishes whether the candidate preserves the declared semantics, which corresponds to verification. Finally a coordinator determines whether the verified candidate still applies to the current workspace, which corresponds to publication. These five steps are given independent meanings throughout this specification, and a good deal of the protocol is concerned with keeping them independent. In particular, a candidate that is semantically correct may nonetheless be stale, and a worker that fails operationally has established nothing about the document it was examining.

The specification makes the following contributions. It defines an abstract state consisting of immutable snapshots and a single head, together with the notion of a job, an attempt, a candidate and a certificate (Section 4). It defines a canonical grammar in which every object has exactly one byte representation and is referenced by the cryptographic hash of that representation (Section 5). It gives a denotational semantics of expressions as partial functions, a proof language for nonzero predicates, and a checker whose acceptance is proved sound with respect to that semantics (Section 6). It separates semantic validity from progress, and derives a bound on the number of successful rewrite commits under a fixed configuration (Section 7). It specifies the filesystem primitives by which immutable objects and the head pointer are published atomically, and the claim and fencing discipline that tolerates workers whose attempts have been abandoned (Section 8). It presents the observation, attempt, commit and recovery algorithms together with a transition model of

the coordinator and the invariants that model preserves (Sections 9 and 10). It states the failure model and its guarantees (Section 12) and provides conformance cases and reference vectors (Section 13 and Appendix A).

The scope is intentionally narrow. The baseline deployment is one Linux host, one local filesystem, one coordinator at a time, and workers that belong to the same operating-system account and cooperate. The specification does not address Byzantine workers, networked filesystems, multiple hosts, or distributed consensus. The proof language is deliberately incomplete, so that the failure to derive a predicate is a normal and well-defined outcome rather than an error. The baseline coordinator commits the first valid candidate it processes, so the specification does not promise that the final normal form is independent of worker timing; a separate policy that restores such independence is described in Section 7. Stronger deployments may be built from the same objects and algorithms, but they are outside the claims made here.

The remainder of the document is organized as follows. Section 2 fixes conventions. Section 3 describes the execution model and trust assumptions. Section 4 defines the abstract state. Section 5 gives the canonical grammar and the identity of objects. Section 6 gives the meaning of expressions, rules and proofs, and proves soundness. Section 7 treats progress and rule safety. Section 8 defines the publication primitives. Section 9 gives the algorithms, and Section 10 the coordinator transition model and its invariants. Section 11 defines quiescence and terminal classification. Section 12 states the failure model and guarantees. Section 13 gives conformance cases. Section 14 places the design in relation to prior work, and Section 15 records limitations and extensions.

2 Conventions and Notation

The verbs *must*, *must not*, *should* and *may* are used with the force of the corresponding requirement levels of Bradner [3]. Statements labelled as definitions, requirements and invariants are normative. Remarks, examples, and text describing directory layouts are informative unless the text states otherwise. Where this document gives a procedure in prose and also in pseudocode, the two are intended to agree, and an implementation that finds a discrepancy should treat the prose of the relevant section as controlling.

A byte string is a finite sequence of octets. The symbol H denotes the SHA-256 function [6], and $\mathrm{id}(b)$ denotes the lowercase hexadecimal rendering of $H(b)$, which consists of exactly sixty-four digits. The set of rational numbers is written $\mathbb{Q}$, and $\mathbb{Q}_\perp = \mathbb{Q} \cup \{\perp\}$ is the same set extended with a distinguished element that denotes the absence of a value. The size of an expression e, written $|e|$, is the number of nodes in its syntax tree, counting each operator, variable and rational literal as one node. The size of a snapshot, written $|S|$, is the sum of the sizes of the expressions of its documents. A finite partial map from a set A to a set B is written $A \rightharpoonup_{\mathrm{fin}} B$. For a path p and an expression E, the subterm at p is written $E|_p$ and the expression obtained by replacing that subterm by R is written $E[p := R]$.

The assumption that underlies content addressing is stated once and used thereafter.

Requirement 2.1 (Collision resistance)**.** The implementation assumes that distinct canonical byte strings have distinct identifiers. When publication discovers a destination that exists with different bytes than those being published, the implementation must treat the situation as a collision and halt publication, rather than overwrite or ignore either object.

Symbol	Meaning
$S,\ S'$	snapshots, with revision $n(S)$, parent $\pi(S)$, configuration $\kappa(S)$, origin $o(S)$ and document map $\delta(S)$
Γ	a context, a finite set of predicates, indexed by canonical order
$[\![e]\!]_\sigma$	the value of expression e under assignment σ, an element of $\mathbb{Q}_\perp$
$e \approx_\Gamma e'$	equivalence of e and e' in every assignment admitted by Γ
$\Gamma \vdash_\pi \mathsf{nonzero}(e)$	the proof π establishes that e is nonzero in Γ
$\mathcal{R}$	the rule catalog, a finite set of named, versioned rules
$j = (S, d, r, p)$	a job: snapshot, document name, rule name and path
$\lvert e\rvert,\ \lvert S\rvert$	node count of an expression and of a snapshot

Table 1: Principal notation.

Role	Authority held	Products
Sentinel	none; observes head advancement and spool activity	events, which are facts and carry no authority
Worker	a claim and an attempt token allocated by the coordinator	obligations, candidates, certificates and results, first in a private staging area and then as immutable objects
Support worker	an obligation and a bounded search budget	support records, which are proof objects offered as hints
Checker	the identity recorded in the configuration	verdicts, bound to an exact job, candidate and checker identity
Coordinator	exclusive lease and a fresh session token	receipts, the head pointer, authoritative stale outcomes, terminal classification

Table 2: Roles and the authority attached to their products.

Table 1 summarizes the principal symbols. Identifiers of objects are written in italic for abstract objects and in typewriter font for concrete records.

3 Execution Model

3.1 Roles

A swarm transforms a local workspace by proposing and certifying symbolic rewrites. The system has five roles, each defined by what it may observe, what it may produce, and what authority its products carry. Table 2 gives the summary. The distinction between production and authority is the central design device: every role except the coordinator can produce objects, but only the coordinator can cause one of them to become part of committed history.

Bash orchestrates processes, files and scheduling. A parser and a certificate checker operate on typed syntax trees and are not themselves written in shell. A worker may use any search procedure that its local resource budget permits, because nothing a worker produces is trusted until the checker has examined it and the coordinator has published it.

3.2 The Recruitment Sequence

The recruitment sequence has five stages, each with its own meaning and its own way of failing.

Recognition. A sentinel observes that the head has advanced, or that a job has become ready, and publishes that fact. Recognition does not commit anyone to work. A worker that recognizes a task it can attempt has learned only that the task exists. Because announcements may be duplicated, delayed or lost, recognition is advisory, and the authoritative list of work is always reconstructible from the current snapshot.

Claim. A claim reserves one attempt at one job. Its purpose is to prevent wasteful duplication of effort, and it confers no right to publish. Claims are issued by the coordinator, which allocates a fresh attempt token for each attempt and registers the claim before the worker is started.

Execution. The worker crosses into a private staging area and produces a candidate, or an obligation, or nothing. Operational failure at this stage is distinct from a proved absence of a matching rewrite, and the distinction is preserved in the outcome vocabulary.

Verification. A checker establishes whether the candidate preserves the declared semantics in the recorded context. Verification is relative to a snapshot, a context and a checker identity, and says nothing about whether the candidate is still applicable.

Publication. The coordinator determines whether the candidate still applies to the current workspace, applies the progress rule, and, if both hold, atomically replaces the head. Publication is the only step that changes what readers see.

3.3 Deployment and Trust

The baseline deployment is one Linux host, one local filesystem and one coordinator at a time. The coordinator lease uses a process-held exclusive lock on a stable inode, so that the lease is released by the operating system when the holding process terminates for any reason. Workers share trust with the coordinator at the operating-system level, and private staging directories are a protocol convention within a single account rather than a security boundary. Record contents are always data. Executable paths come from a trusted local configuration, and no field of any record is passed to shell evaluation. These assumptions determine what kinds of fault the protocol must tolerate: a worker may be slow, may crash, may be restarted after its claim was fenced, or may produce a wrong answer, but a worker does not forge the coordinator's lock or rewrite the head by hand.

4 Abstract State

4.1 Objects, Snapshots and the Head

The authoritative state of a swarm is a sequence of immutable snapshots with one current head. An *object store* is a finite set of canonical byte strings, each addressed by its identifier. A *snapshot* contains a monotone revision number, its parent snapshot, a configuration identifier, an origin, and a finite map from logical document names to pairs of an expression object identifier and a context object identifier.

Definition 4.1 (Snapshot)**.** A snapshot is a tuple $S = (n, \pi, \kappa, o, \delta)$ where $n \in \mathbb{N}$ is the revision, π is either the symbol root or the identifier of a snapshot, κ is the identifier

of a configuration, o is either the symbol import or the identifier of a candidate, and $\delta\colon \text{Names} \rightharpoonup_{\text{fin}} \text{ExprId} \times \text{CtxId}$ is a finite document map. An *initial* snapshot has $n = 0$, parent root and origin import. Every later snapshot S' has a parent S with $n(S') = n(S) + 1$.

The *head* is one snapshot identifier, held in a single pointer. Readers resolve the head exactly once per operation and read all documents from the snapshot it names. A reader therefore never observes a mixture of two snapshots, and an operation that spans several documents sees a single consistent state.

Proposition 4.2 (Snapshots commit to their history)**.** *Under Requirement 2.1, the identifier of a snapshot determines its parent, revision, configuration, origin and document map, and, by induction on the parent link, determines the entire chain of snapshots up to the initial snapshot.*

Proof. The identifier is the hash of the canonical bytes of the snapshot record, and those bytes contain the revision, parent identifier, configuration identifier, origin and every document entry. Distinct tuples have distinct canonical bytes by the unambiguity of the grammar (Proposition 5.1), hence distinct identifiers. The parent identifier is itself a snapshot identifier, so the argument repeats at each step until the parent is root. □

A consequence that the algorithms rely on is that two snapshots with identical document maps but different ancestry have different identifiers. A document that changes and later returns to the same bytes therefore does not return the workspace to an earlier snapshot, and a proposal checked against an earlier snapshot cannot be mistaken for one checked against a later snapshot whose contents happen to coincide.

4.2 Configurations and the Rule Catalog

A *configuration* names a rule catalog, a checker and the set of enabled rules. The rule catalog defines each rule by a stable name, a version and an implementation identity. The configuration is recorded in each snapshot by identifier, so that every judgment made about a snapshot is made under a definite catalog and a definite checker. A change to assumptions, enabled rules, the checker or document contents is a serialized snapshot transaction. An external editable source can be imported through that interface, but its filesystem path has no authority over committed state: only the snapshot created by the transaction does.

4.3 Jobs, Attempts, Candidates and Certificates

A *job* is a request to attempt one rule at one node under one context and one snapshot. An *attempt* is one execution of that job, identified by a token that is never reused. A *candidate* is one proposed replacement for the expression of one document, accompanied by either a certificate or a request for assistance. A *certificate* is a proof object that binds the job and the proposed expression. A successful attempt may yield no rewrite, and operational failure and a proved lack of a matching rewrite have different meanings.

Definition 4.3 (Outcome vocabulary)**.** The result of an attempt is one of five outcomes. *Proposed* carries a candidate identifier. *Blocked* carries the identifier of an obligation, which names a predicate that the worker could not establish. *No-match* states that the configured rule has no applicable syntactic instance at the selected path. *Failed* carries a diagnostic

name and states only that the attempt did not complete, so that it establishes nothing about applicability. *Stale* states that the head no longer equals the job snapshot, and only the coordinator may emit it as an authoritative outcome.

The separation in Definition 4.3 has a precise consequence for quiescence. Only a confirmed no-match contributes to a fixed point, whereas a failed attempt that exhausts its retries leaves applicability undecided and forces an incomplete classification (Section 11).

4.4 Transactions

Two kinds of transaction advance the head. A *rewrite commit* replaces one document expression by the certified successor expression, keeps that document's context and all other entries, sets the parent to the job snapshot, increments the revision, and records the candidate identifier as origin. An *external transaction* imports a change to assumptions, enabled rules, the checker or document contents, and records the terminal `import` as origin. In both cases the new snapshot and any supporting objects are published as immutable objects before the head pointer is replaced, and both are serialized by the coordinator. Version 0.1 defines receipts only for rewrite commits, because the receipt record names a candidate; an external transaction is witnessed by the origin of the snapshot it creates.

5 Canonical Grammar and Object Identity

5.1 Lexical Classes

Every object consists of exactly one record followed by a line feed. The productions below specify all spaces literally, and quoted terminals denote literal ASCII. Juxtaposition denotes concatenation, and the word `eps` in the displayed grammar denotes the empty string, which is written ε in running text. The lexical classes are fixed as follows. An *identifier* is exactly sixty-four lowercase hexadecimal digits. A *token* is exactly thirty-two lowercase hexadecimal digits. A *name* begins with a lowercase ASCII letter and continues with zero or more lowercase letters, ASCII digits, hyphens or underscores, with at most sixty-four characters in total. A *natural number* is zero, or a nonzero digit followed by zero or more digits. A *positive number* is a natural number other than zero. A *signed integer* is a natural number, or a hyphen followed by a positive number. In the grammar, `<id>`, `<token>` and `<name>` denote the first three classes, `<nat>` and `<pos>` the next two, and `<int>` the last.

5.2 Grammar

The grammar is given in Backus–Naur form. It is the normative definition of concrete syntax.

```
<object> ::= <expr> | <context> | <snapshot> | <head> | <job>
           | <claim> | <candidate> | <obligation> | <support>
           | <certificate> | <event> | <result> | <configuration> | <receipt>

<expr> ::= <rat> | "(var " <name> ")" | "(neg " <expr> ")"
         | "(add " <expr> " " <expr> ")" | "(mul " <expr> " " <expr> ")"
         | "(div " <expr> " " <expr> ")"
```

```
<rat> ::= "(rat " <int> " " <pos> ")"
<pred> ::= "(nonzero " <expr> ")"
<preds> ::= eps | " " <pred> <preds>
<context> ::= "(context" <preds> ")"

<configuration> ::= "(config " <id> " " <id> <rules> ")"
<rules> ::= eps | " " <name> <rules>
<snapshot> ::= "(snapshot " <nat> " " <parent> " " <id> " " <origin> <entries> ")"
<origin> ::= "import" | <id>
<receipt> ::= "(receipt " <id> " " <id> " " <id> ")"
<parent> ::= "root" | <id>
<entries> ::= eps | " " <entry> <entries>
<entry> ::= "(doc " <name> " " <id> " " <id> ")"
<head> ::= "(head " <id> ")"

<path> ::= "(path" <indices> ")"
<indices> ::= eps | " " <index> <indices>
<index> ::= "0" | "1"
<job> ::= "(job " <id> " " <name> " " <name> " " <path> ")"
<claim> ::= "(claim " <id> " " <token> " " <token> ")"
<candidate> ::= "(candidate " <id> " " <token> " " <id> " " <id> ")"
<obligation> ::= "(obligation " <id> " " <token> " " <pred> ")"
<support> ::= "(support " <id> " " <proof> ")"
<certificate> ::= "(cert " <id> " " <id> " " <derivation> ")"
<derivation> ::= "(self-div " <proof> ")" | "(add-zero-right)"
               | "(mul-one-right)" | "(rat-fold)"
<proof> ::= "(assumed " <nat> ")" | "(literal " <rat> ")"
          | "(product " <proof> " " <proof> ")"

<event> ::= "(head-advanced " <id> ")"
          | "(job-ready " <id> ")" | "(attempt-returned " <id> ")"
          | "(proof-required " <id> ")" | "(proof-available " <id> ")"
<result> ::= "(result " <id> " " <token> " " <outcome> ")"
<outcome> ::= "(proposed " <id> ")" | "(blocked " <id> ")"
            | "(no-match)" | "(failed " <name> ")" | "(stale)"
```

The nonterminals `<path>`, `<pred>`, `<proof>`, `<derivation>` and `<outcome>` occur only as components of other records and are not objects in their own right.

5.3 Well-Formedness Conditions

Derivability in the grammar is necessary but not sufficient for an object to be canonical. Four further conditions are part of the definition of a canonical object, and a parser must reject any record that violates them. First, a rational literal must have coprime numerator and denominator, and zero must be written `(rat 0 1)`. Second, the predicates of a context must be unique and sorted by their canonical bytes. Third, the enabled rules of a configuration must be unique and sorted by rule name. Fourth, the document entries of a snapshot must be unique and sorted by document name. Sorting is bytewise on the canonical encoding.

5.4 Unambiguity, Typing and Identity

Proposition 5.1 (Unambiguity)**.** *Every canonical byte string has exactly one parse tree, and every parse tree has exactly one canonical byte string. The type of an object is determined*

by its leading literal.

Proof. In each production the alternatives begin with pairwise distinct literal prefixes, or are distinguished by a single character that cannot begin the competing alternative. The top-level alternatives begin with the literals visible in the grammar, which are pairwise distinct; in particular a head record begins with `(head ` followed by a space, whereas the corresponding announcement begins with `(head-advanced` . Lists are introduced by a space and terminated by a closing parenthesis, so the next character decides between continuation and termination. The terminal `import` begins with a letter that is not a hexadecimal digit, and `root` likewise, so `<origin>` and `<parent>` are decided by one character. The lexical classes have fixed width or an unambiguous terminator, since spaces and closing parentheses do not occur inside a name, an identifier or a number, and a natural number cannot begin with zero unless it is zero. A deterministic one-character lookahead therefore selects a unique production at every step, and the parse is unique. Because spaces are specified literally and no production admits optional whitespace, the printing of a parse tree is also unique. □

Definition 5.2 (Identifier of an object)**.** The identifier of a canonical object o with byte string $b(o)$ is $\mathrm{id}(b(o))$, where $b(o)$ includes the terminating line feed. An object reference is an identifier together with an expected object type. On lookup, an implementation must verify that the hash of the retrieved bytes equals the identifier, that the bytes are canonical, and that the type determined by the leading literal equals the expected type.

5.5 Reference Structure

Table 3 lists, for each record type, the meaning of its fields in order. The role of each identifier is fixed by position, so that no field carries a type tag of its own. The two identifiers of a configuration are, in order, the identity of the rule catalog and the identity of the checker. These denote trusted local artifacts and need not themselves be objects in the store; they are included in the configuration so that a change of either alters the configuration identifier and therefore the snapshots that cite it.

A job lists the snapshot, document name, rule name and path. A claim lists the job identifier, the attempt token and the coordinator session token. A candidate lists the job, the attempt token, the full replacement expression and the certificate, and a certificate binds the job and the same full expression. An obligation names the job and the attempt and carries the required predicate, and a support record binds an obligation identifier to a proof. An attempt-returned event points to a worker result. Only the coordinator emits authoritative stale outcomes.

Definition 5.3 (Valid record graph)**.** A proposed result is part of a valid record graph when every referenced object exists, has the expected type and verifies its hash, and the following agreements hold. The job identifiers of the result, the candidate and the certificate are equal. The attempt token of the result equals that of the candidate. The candidate identifier named by the `proposed` outcome is the identifier of the candidate. The expression identifier of the candidate equals that of the certificate.

Record	Fields in order
`config`	catalog identity, checker identity, enabled rule names
`snapshot`	revision, parent snapshot or `root`, configuration, origin, document entries
`doc`	document name, expression identifier, context identifier
`job`	snapshot identifier, document name, rule name, path
`claim`	job identifier, attempt token, coordinator session token
`candidate`	job identifier, attempt token, full replacement expression identifier, certificate identifier
`cert`	job identifier, expression identifier (equal to the candidate's), derivation
`obligation`	job identifier, attempt token, required predicate
`support`	obligation identifier, proof
`result`	job identifier, attempt token, outcome
`receipt`	candidate identifier, previous snapshot identifier, accepted successor identifier
`head`	snapshot identifier
events	the single identifier of the object that the event concerns

Table 3: Fields of each record type.

5.6 Paths

Paths count children from zero. A unary node has child zero. Binary operators have left child zero and right child one. Variable and rational nodes have no expression children. The empty path selects the entire document, and invalid paths are rejected.

Definition 5.4 (Subterm and preorder enumeration)**.** For an expression E and path p, the subterm $E|_p$ is defined by $E|_\varepsilon = E$ and $f(E_0, \ldots, E_{k-1})|_{i\,p'} = E_i|_{p'}$ when $0 \le i < k$, and is undefined otherwise. The preorder enumeration $\mathrm{paths}(E)$ is the sequence that begins with ε and continues, for each child index i in increasing order, with the paths $i\,p'$ for p' in $\mathrm{paths}(E_i)$. Its length equals $|E|$, since every node of the syntax tree has exactly one path.

6 Semantics of Expressions, Rules and Proofs

6.1 Denotation

Expressions denote partial rational-valued functions of rational assignments to variables. An *assignment* is a function $\sigma\colon \mathrm{Var} \to \mathbb{Q}$ from variable names to rational numbers. Each operator first requires all operands to be defined, and division additionally requires a nonzero denominator. Writing $\bot$ for the undefined value, the denotation $[\![e]\!]_\sigma \in \mathbb{Q}_\bot$ is given by the following equations, in which each clause applies only when every operand is different from $\bot$, and yields $\bot$ otherwise.

$$
\begin{aligned}
[\![(\mathsf{rat}\ n\ d)]\!]_\sigma &= n/d, & [\![(\mathsf{var}\ x)]\!]_\sigma &= \sigma(x),\\
[\![(\mathsf{neg}\ a)]\!]_\sigma &= -[\![a]\!]_\sigma, & [\![(\mathsf{add}\ a\ b)]\!]_\sigma &= [\![a]\!]_\sigma + [\![b]\!]_\sigma,\\
[\![(\mathsf{mul}\ a\ b)]\!]_\sigma &= [\![a]\!]_\sigma \cdot [\![b]\!]_\sigma, & [\![(\mathsf{div}\ a\ b)]\!]_\sigma &= [\![a]\!]_\sigma \,/\, [\![b]\!]_\sigma \quad (\text{if } [\![b]\!]_\sigma \neq 0).
\end{aligned}
$$

6.2 Contexts and Equivalence

A predicate $\mathsf{nonzero}(e)$ holds under σ exactly when $[\![e]\!]_\sigma \in \mathbb{Q} \setminus \{0\}$, which is to say when e is defined and has a nonzero value. A context Γ is a finite set of predicates, and it *admits* those assignments that satisfy every predicate it contains. Contexts are supplied by the user or imported as explicit assumptions, and every accepted judgment is relative to its recorded context. The set of assignments admitted by Γ is written $\mathrm{Adm}(\Gamma)$.

Definition 6.1 (Equivalence under a context)**.** Expressions e and e' are *equivalent under* Γ, written $e \approx_\Gamma e'$, when $[\![e]\!]_\sigma = [\![e']\!]_\sigma$ in $\mathbb{Q}_\perp$ for every $\sigma \in \mathrm{Adm}(\Gamma)$. Equality in $\mathbb{Q}_\perp$ means that the two expressions have equal definedness and, whenever defined, equal value.

The relation $\approx_\Gamma$ is an equivalence relation, because it is the kernel of a function from expressions to functions on $\mathrm{Adm}(\Gamma)$. This is the property that makes a sequence of checked rewrites preserve meaning.

Lemma 6.2 (Replacement)**.** *Let E be an expression, p a path valid in E, and R an expression with $E|_p \approx_\Gamma R$. Then $E \approx_\Gamma E[p := R]$.*

Proof. By induction on the length of p. When p is empty the claim is the hypothesis. Otherwise $E = f(E_0, \ldots, E_{k-1})$ and $p = i\,p'$. For every $\sigma \in \mathrm{Adm}(\Gamma)$, the value $[\![E]\!]_\sigma$ is determined by the values $[\![E_j]\!]_\sigma$ through the clause for f, and the replacement leaves E_j unchanged for $j \neq i$. By the induction hypothesis $[\![E_i[p' := R]]\!]_\sigma = [\![E_i]\!]_\sigma$, so the arguments to the clause for f are unchanged and so is the result. □

Remark 6.3. There are no binding constructs in the language, so no variable capture can arise, and the context of a subterm is the context of the whole document. The strictness of the operators is not needed for Lemma 6.2, but it matters for which rules are sound. A rule that discarded an operand whose definedness is not known, such as rewriting $(\mathsf{mul}\ e\ (\mathsf{rat}\ 0\ 1))$ to $(\mathsf{rat}\ 0\ 1)$, would change an undefined expression into a defined one, and it is accordingly absent from the baseline catalog.

6.3 The Baseline Rule Catalog

For a document expression E, an occurrence p and a proposed replacement T, a valid certificate proves that E and T have equal definedness and equal value whenever defined, in every assignment admitted by the context. The checker reads E and the context from the job snapshot, retrieves the selected subterm $S = E|_p$, computes a replacement subterm R from the named derivation, constructs $E[p := R]$, and compares canonical bytes against T. This keeps the scope of the candidate within the certified occurrence.

Table 4 defines the four baseline rules by pattern, replacement and proof obligation. For the rule $\mathsf{self\text{-}div}$ the two occurrences of e must have identical canonical syntax, and the checker discharges the obligation $\mathsf{nonzero}(e)$ with the proof carried in the derivation. The rules for right addition of zero and right multiplication by one return their left operand, and both preserve undefinedness because the discarded operand is a defined literal. The rule $\mathsf{rat\text{-}fold}$ evaluates a unary or binary operator whose immediate operands are rational literals, using exact rational arithmetic, and it writes the result as a canonical literal. A division with a zero denominator yields no fold. An enabled rule name must agree with the derivation tag under the recorded configuration, and the derivation tags have the same spelling as the rule names.

Rule and derivation	Pattern S	Replacement R	Obligation
`self-div` π	(div e e)	(rat 1 1)	nonzero(e)
`add-zero-right`	(add e (rat 0 1))	e	none
`mul-one-right`	(mul e (rat 1 1))	e	none
`rat-fold`	$f(\ell_1)$ or $f(\ell_1, \ell_2)$ with ℓ_i literals, $f \in \{\mathsf{neg}, \mathsf{add}, \mathsf{mul}, \mathsf{div}\}$, and ℓ_2 not zero when $f = \mathsf{div}$	the exact value as a canonical literal	none

Table 4: The baseline rule catalog.

Definition 6.4 (Match test)**.** For each rule r the catalog supplies a decidable predicate $M_r(E, p)$ that holds exactly when $E|_p$ is a syntactic instance of the pattern of r in Table 4, including the side conditions stated there. A rule *matches* at p when $M_r(E, p)$ holds, and *has no applicable instance* otherwise. For rat-fold the match test includes the requirement that a divisor literal be nonzero, so that division by a zero literal is a non-match and not a match without a certificate.

Lemma 6.5 (Rule soundness)**.** *Let r be a baseline rule with $M_r(E, p)$, let $S = E|_p$, let R be the replacement that Table 4 associates with S, and suppose the obligation of r, if any, holds at every assignment in* $\mathrm{Adm}(\Gamma)$*. Then $S \approx_\Gamma R$.*

Proof. Fix $\sigma \in \mathrm{Adm}(\Gamma)$. For self-div the obligation gives $[\![e]\!]_\sigma \in \mathbb{Q} \setminus \{0\}$, so $[\![(\mathsf{div}\ e\ e)]\!]_\sigma = 1 = [\![(\mathsf{rat}\ 1\ 1)]\!]_\sigma$, and both sides are defined. For add-zero-right the literal (rat 0 1) is always defined with value 0, so the sum is defined exactly when e is defined and then equals $[\![e]\!]_\sigma$. The case of mul-one-right is identical with the literal 1. For rat-fold the operands are literals and therefore defined, the match test excludes division by zero, and exact rational arithmetic yields a defined value that the canonical literal denotes. □

6.4 The Proof Language

A proof establishes that an expression is nonzero in a context. The proof language has three constructors. For (assumed i) the demanded predicate must equal the predicate at zero-based index i in the canonical context. For (literal r) the demanded predicate must be nonzero(r) and the numerator of r must be nonzero. For (product A B) the demanded predicate must be nonzero((mul a b)), and the checker recursively checks A against nonzero(a) and B against nonzero(b). These requirements are summarized by the following derivation rules, in which Γ_i denotes the ith predicate of Γ in canonical order.

$$\frac{\Gamma_i = \mathsf{nonzero}(e)}{\Gamma \vdash_{(\mathsf{assumed}\ i)} \mathsf{nonzero}(e)} \qquad \frac{n \neq 0}{\Gamma \vdash_{(\mathsf{literal}\ (\mathsf{rat}\ n\ d))} \mathsf{nonzero}((\mathsf{rat}\ n\ d))}$$

$$\frac{\Gamma \vdash_A \mathsf{nonzero}(a) \qquad \Gamma \vdash_B \mathsf{nonzero}(b)}{\Gamma \vdash_{(\mathsf{product}\ A\ B)} \mathsf{nonzero}((\mathsf{mul}\ a\ b))}$$

Lemma 6.6 (Proof soundness)**.** *If $\Gamma \vdash_\pi \mathsf{nonzero}(e)$ then $[\![e]\!]_\sigma \in \mathbb{Q} \setminus \{0\}$ for every $\sigma \in \mathrm{Adm}(\Gamma)$.*

Proof. By induction on π. For (assumed i) the predicate nonzero(e) is a member of Γ, and every admitted assignment satisfies every member. For (literal r) the value is n/d with $n \neq 0$ and $d > 0$, hence nonzero. For (product A B) the induction hypothesis gives that a and b are defined and nonzero, so their product is defined and is nonzero because $\mathbb{Q}$ has no zero divisors. □

The proof language is deliberately incomplete. Failing to derive a predicate leaves it unresolved, and an unresolved predicate yields a blocked outcome and not a rewrite. Incompleteness is easily exhibited. In the context $\Gamma = (\mathsf{nonzero}(x))$ the predicate nonzero((mul x x)) is derived by (product (assumed 0) (assumed 0)). The predicate nonzero((neg x)) is semantically valid in every admitted assignment, but it has no derivation, because the language has no constructor that mentions negation, and the predicate nonzero((div x x)) is likewise valid but underivable. A richer proof language can be introduced by a new configuration and a new checker identity, and Lemma 6.6 is the property that any such extension must preserve.

6.5 The Checker

The checker accepts or rejects a triple consisting of a job $j = (S, d, r, p)$, a candidate c and a certificate χ, and it proceeds in four steps.

First, it resolves the job snapshot and its configuration, verifies that the rule r is enabled in the configuration, and verifies that the graph of c and χ is valid in the sense of Definition 5.3 with respect to j. Second, it reads the expression E and the context Γ of document d from S, and obtains $S_p = E|_p$. A path invalid in E causes rejection. Third, it dispatches on the derivation tag of χ, which must agree with r. For self-div the subterm must have the form (div a b) with a and b byte-identical, the proof carried by the derivation must satisfy $\Gamma \vdash_\pi \mathsf{nonzero}(a)$ under the checking rules above, and $R = (\mathsf{rat}\ 1\ 1)$. For add-zero-right and mul-one-right the subterm must match the pattern of the rule and R is its left operand. For rat-fold the subterm must match the pattern and R is computed by exact arithmetic. Fourth, it constructs $E[p := R]$ and compares its canonical bytes with those of the expression named by the candidate, accepting exactly when they are equal.

Theorem 6.7 (Soundness of the checker)**.** *If the checker accepts a job j, candidate c and certificate χ, and T is the expression named by c, then $E \approx_\Gamma T$, where E and Γ are the expression and context of document d in the snapshot of j.*

Proof. Acceptance entails $T = E[p{:=}R]$ for the replacement R computed from the derivation. By Lemma 6.6, where an obligation is present it holds at every admitted assignment, and so by Lemma 6.5 the equivalence $E|_p \approx_\Gamma R$ holds. Lemma 6.2 then gives $E \approx_\Gamma E[p{:=}R] = T$. □

Remark 6.8 (Syntax and semantics are separate checks). The grammar can encode expressions on which a semantically invalid rewrite looks syntactically plausible. The document (mul (div (rat 1 1) (rat 0 1)) (rat 0 1)) is undefined at every assignment, while the literal (rat 0 1) is defined and denotes zero. Both are well-formed objects, yet the second cannot be certified as a replacement for the first, because no derivation of the baseline catalog produces it from the first. This is the reason a certificate is checked against semantics and not merely parsed.

6.6 Support Records

A support record may guide certificate construction, and its publication confers no authority to change a document or add an assumption. A support worker that completes a proof publishes a support record, and a resumed attempt of the same job may consume it as an input hint. The checker nevertheless verifies the proof again in the exact job context. An accepted certificate establishes equivalence only, and the coordinator then applies a separate progress rule, which is the subject of the next section.

7 Progress and Rule Safety

7.1 The Progress Measure

Semantic validity does not by itself guarantee that a sequence of rewrites ends. The baseline normalizer therefore accepts only rewrites that strictly decrease the number of expression nodes in the affected document, counting each operator, variable and rational literal as one node. The check is applied after semantic verification and before publication.

Definition 7.1 (Progress rule)**.** A verified candidate with replacement expression T for a document with current expression E satisfies the progress rule when $|T| < |E|$.

Lemma 7.2 (Baseline rules decrease size)**.** *For every baseline rule r and every match, the replacement R satisfies $|R| < |S|$, and consequently $|E[p := R]| < |E|$.*

Proof. For self-div the sizes are $|S| = 2|e| + 1 \geq 3$ and $|R| = 1$. For add-zero-right and mul-one-right the sizes are $|S| = |e| + 2$ and $|R| = |e|$. For rat-fold the subterm has two or three nodes and the result has one. Replacing a subterm by a smaller one reduces the size of the whole by exactly the difference of the sizes of the subterms. □

Theorem 7.3 (Bound on successful commits)**.** *Fix a configuration whose rules all satisfy the conclusion of Lemma 7.2, and suppose that no external transaction occurs. Let S be a snapshot with m documents and total size $N = |S|$. Then the number of successful rewrite commits that can follow S is at most $N - m$.*

Proof. A rewrite commit changes exactly one document expression, leaves the document set unchanged, and by the progress rule decreases the total size by at least one. Every document expression has at least one node, so the total size never falls below m. A decreasing sequence of integers bounded below by m and beginning at N has at most $N - m$ strict steps. □

The bound concerns commits and not attempts. If each job is permitted at most b attempts, including attempts resumed after support, then the number of attempts at a snapshot of size N' with ρ enabled rules is at most $\rho N' b$, since every node of every document has one path and each path is paired with each enabled rule. Because snapshot size is non-increasing along rewrite commits, the total number of attempts over a run is at most $(N - m + 1)\,\rho\,N\,b$. This bound is far from tight and gives no estimate of the work a particular run requires.

7.2 Overlap and Order Dependence

Rules may overlap, and normal forms may depend on the order of accepted candidates. The baseline coordinator may commit the first valid candidate it processes. Exact replay uses the committed snapshot chain and its recorded receipts, with no need to repeat a scheduling decision. The size measure makes every rewriting sequence finite, so the rewriting relation is terminating. A terminating relation that is also locally confluent has unique normal forms by Newman's lemma [13, 1], but local confluence is a property of a catalog to be established separately, and this specification makes no confluence claim, either for the baseline catalog or in general. Where it fails, the final snapshot depends on commit order.

7.3 A Timing-Independent Commit Policy

A run whose outcome must be independent of worker timing needs a separate policy. Let $\prec$ be the total order on the jobs of a snapshot given by document name, then the position of the path in the preorder enumeration, then the position of the rule in the sorted list of enabled rules; this is the order in which Algorithm A creates jobs. Under the *ordered commit policy*, the coordinator commits the $\prec$-least job whose result is a verified proposal, and only after every earlier job has a definitive result, meaning a confirmed no-match, an exhausted blocked obligation, an exhausted operational failure, or a rejected candidate. The sequence of commits then depends only on the definitive result of each job. It is independent of timing provided each of those results is itself independent of timing, which holds for the baseline rules when workers construct proofs deterministically and are given budgets that do not depend on the clock. The ordered policy trades throughput for reproducibility, since a late result for an early job delays every later commit.

7.4 The Meaning of No-Match

The result `(no-match)` means that the configured rule has no applicable syntactic instance at the selected path. A rule that matches but lacks a certificate yields either a blocked result with a named predicate or an operational failure. The coordinator confirms definitive structural no-match outcomes using the catalog's match test M_r, and does not rely on the word of a worker. A failed worker cannot establish that a path is irreducible, and a worker-reported no-match that the match test contradicts is treated as an operational failure of the attempt.

8 Spool and Publication Primitives

8.1 Layout

Immutable objects, the head file, claim directories, attempt staging directories, announcements and receipts are kept below one workspace root. Paths are derived by the implementation from validated hashes, tokens and fixed directory names, so that no field of any record can select a location outside the workspace. An implementation must validate the length and alphabet of every identifier and token before using it in a path. The layout of Table 5 is informative, and any layout that satisfies the requirements of this section is permitted.

Location below the root	Content
`objects/`	immutable objects, one file per identifier
`head`	the single pointer record `(head <id>)`
`coordinator.lock`	the stable inode on which the exclusive lease is held
`claims/<session>/<job>/`	one directory per job and session, holding immutable claim records
`attempts/<token>/`	private staging area of one attempt, never reused
`spool/`	announcements, which are events published as immutable objects
`receipts/<successor>`	receipt of the commit that produced the named snapshot

Table 5: An informative workspace layout.

Algorithm 1 The primitives Put and ReplaceHead.

function Put(record o)
    validate o against the grammar and the well-formedness conditions
    $b \leftarrow$ canonical bytes of o; $i \leftarrow \mathrm{id}(b)$; $\delta \leftarrow$ destination derived from i
    write b to a temporary file θ beside δ
    durable profile: flush θ
    publish θ at δ by an atomic no-replace operation
    if δ already exists **then**
        compare the bytes at δ with b; if they differ, halt publication; remove θ
    end if
    durable profile: flush the directory that contains δ
    return i
end function

function ReplaceHead(identifier i) ▷ coordinator only
    write the record (`head` i) to a temporary file beside the head pointer
    durable profile: flush the temporary file
    rename the temporary file over the head pointer
    durable profile: flush the directory that contains the head pointer
end function

8.2 The Primitive Put

The primitive Put validates and hashes a record, writes its bytes to a temporary file beside the object destination, and publishes the complete file by an atomic no-replace operation, such as creating a hard link at the destination, which fails if the destination exists, or a rename that refuses to replace. If the destination exists, the bytes are compared. A byte difference is a collision and halts publication (Requirement 2.1). Because identifiers are derived from content, Put is idempotent, and two processes that publish the same record concurrently both succeed and leave one file.

8.3 Replacing the Head

ReplaceHead writes a temporary pointer beside the head and atomically renames it over the old pointer. Only the coordinator calls it. The rename is the visibility boundary: before it, readers resolve the old head, and after it, they resolve the new one, and no reader observes an intermediate state. A Bash implementation uses a small filesystem helper for

these operations, because the shell has no portable means of flushing files and directories or of an atomic no-replace publication.

8.4 Durability Profiles

For machine-crash durability, an implementation must flush object bytes and their containing directories before it publishes references to them, and it must flush the replacement pointer and its containing directory in the appropriate order around the rename. The difficulty of obtaining crash-consistent updates from the interfaces that file systems actually provide is well documented [14], which is why the ordering is stated as a requirement and not left to convention. A process-crash-only implementation may omit durability flushing and must identify that profile when it starts. The guarantees of Section 12 are stated separately for the two profiles.

8.5 Claims, Sessions and Fencing

Claims are exclusive per job and coordinator session. A claim is reserved by atomic directory creation, which fails if the directory exists, and its immutable claim record is then published into that directory. The coordinator allocates attempt tokens and registers the claim in its active-attempt table before it spawns the worker. A coordinator session begins with the acquisition of the lease and the allocation of a fresh session token, and it ends when the coordinator process ends. Old-session claim directories cannot prevent recruitment in a fresh session, because the session token is part of the path.

Only the coordinator reclaims an attempt. On timeout it first *fences* the token in its active-attempt table, which means that it removes the token from the set of tokens whose results may be accepted, then requests termination of the worker, and may allocate a fresh token within the retry budget. A late result from the fenced token cannot be accepted, whatever the physical state of the process that produced it. This is the familiar discipline of fencing tokens [9], applied here with the coordinator as the only party that checks the token. Attempt tokens are never reused.

Proposition 8.1 (At most one eligible token per job)**.** *Within one session, at every moment, at most one attempt token for each job is eligible, meaning that it is in the active-attempt table and not fenced.*

Proof. The claim directory for a job and session is created at most once, so only one reservation succeeds. The coordinator publishes a further claim record into that directory only after it has fenced the previous token for the job, and it allocates tokens itself, so there is no concurrent allocation. Hence the set of eligible tokens for a job has at most one element throughout. □

A fenced worker may still be running. The protocol tolerates this, because everything such a worker can do is either confined to its private staging directory or consists of publishing immutable, content-addressed objects, and neither affects committed state. Acceptance is serial, and the protocol tolerates duplicate computation.

8.6 Retention and Cleanup

While work is running, every object reachable from the head history or from a live attempt must be retained. The first implementation can retain all immutable objects. Deletion of a staging directory must wait for process reaping, or must use session isolation, so that late activity of a worker cannot write into a reused directory. Because tokens are never reused and the staging directory is named by token, a directory is never reused in any case.

8.7 Announcements

Announcements are events published as immutable objects in the spool. They may be duplicated, delayed or lost. On coordinator restart, pending work is rebuilt from the current snapshot, optionally reusing verified immutable objects. The coordinator treats an event as a hint that something may have changed, never as a record of what has changed.

Proposition 8.2 (Announcements are idempotent hints)**.** *Processing any announcement any number of times, in any order, and the loss of any announcement, leave the committed chain unchanged.*

Proof. An announcement can cause only three kinds of action. A head announcement causes a rescan of the current head, which registers each job once and is stateless with respect to the chain. A result announcement causes consumption of a result, which requires an active token for the job; the first consumption retires the token, so any later consumption finds none and discards the result. A proof announcement registers a support request once per current job and predicate, and cannot reopen completed work. None of these actions can change the head except through a commit that has passed every check of Algorithm C. A lost announcement is recovered by the polling that Algorithm D performs, which depends only on the spool and on the active workers. □

9 Algorithms

9.1 Algorithm A: Observe and Recruit

On startup, and on every advancement of the head, the coordinator reads the head pointer once, validates the snapshot that it names, fences or cancels every active attempt whose job snapshot differs from it, and enumerates the jobs of the snapshot. Documents are visited in canonical name order, nodes in preorder, and rules in canonical order. Each job is published and registered once in the state for the snapshot. A job at which the catalog confirms a structural mismatch is marked as confirmed no-match, and every other job is placed in the ready queue. While a worker slot is available and the queue holds a job for the current snapshot, the coordinator allocates a fresh attempt token, reserves the claim, registers the active attempt, and launches the rule worker with the job identifier and the token.

Lemma 9.1 (Complete enumeration)**.** *When Algorithm A completes without interruption at a snapshot S with ρ enabled rules, it has registered exactly $\rho\,|S|$ jobs, one for each triple of document, node path and enabled rule.*

Proof. Each document contributes $|E|$ paths by the definition of the preorder enumeration, and each path is paired with each enabled rule exactly once. Registration is once per job,

Algorithm 2 Observe and recruit.

```
1: procedure RECRUIT                                   ▷ on startup or head advancement
2:     S ← the snapshot named by one read of the head pointer, validated
3:     fence every active attempt whose job snapshot differs from S
4:     for all documents d of S in canonical name order do
5:         parse the expression E of d
6:         for all p ∈ paths(E) do
7:             for all enabled rules r in canonical order do
8:                 J ← PUT(job(S, d, r, p)) and register J once in the state for S
9:                 if the catalog confirms ¬M_r(E, p) then
10:                    mark J as confirmed no-match
11:                else
12:                    append J to the ready queue
13:                end if
14:            end for
15:        end for
16:    end for
17:    while a worker slot is free and the ready queue holds a job for S do
18:        take the next ready job J and allocate a fresh attempt token t
19:        reserve the claim for J and register (t, J) as an active attempt
20:        launch the rule worker with the identifier of J and with t
21:    end while
22: end procedure
```

since the job identifier is a function of the quadruple of snapshot, document, rule and path. □

A sentinel may avoid rescanning when it sees a duplicate notification for the same snapshot. The baseline deliberately rebuilds jobs on head change, even for documents whose bytes remain identical. A later incremental scheduler may reuse results only with evidence that all their relevant inputs are identical, and the identifiers of expressions, contexts and configurations are the natural form of such evidence.

9.2 Algorithm B: Attempt a Rewrite and Obtain Support

A worker resolves and checks the job snapshot and its configuration, reads the document expression and context, and locates the selected subterm. It then applies the match test of the configured rule. If there is no structural match it publishes a no-match result. Otherwise, if the rule requires a predicate, it tries to construct a proof under the job context. If a proof is unavailable it publishes the obligation and a blocked result, and if a proof is found it publishes the candidate, the certificate and a proposed result. A rule that requires no predicate proceeds directly to the candidate.

The coordinator can dispatch blocked obligations to local support workers. In the baseline, a support worker searches the recorded context and the finite proof tree induced by the demanded term. A completed proof can resume the same job with a fresh attempt, which consumes it as an input hint. Each support request is registered once per current job and predicate. Exhausted proof searches are recorded, and finite attempt and support budgets are enforced, so that a duplicate proof announcement cannot reopen completed

Algorithm 3 Attempt a rewrite.

procedure ATTEMPT(job $J = (S, d, r, p)$, token t)
    resolve and check S and its configuration
    read the expression E and the context Γ of d; locate the subterm $E|_p$
    if $\neg M_r(E, p)$ **then**
        publish a result `(no-match)`
    else if r requires a predicate φ **then**
        try to construct a proof of φ under Γ, consulting any support record for J
        if no proof is found **then**
            publish the obligation for φ and a result `(blocked ...)`
        else
            publish the candidate, the certificate and a result `(proposed ...)`
        end if
    else
        publish the candidate, the certificate and a result `(proposed ...)`
    end if
end procedure

work.

Remark 9.2. For the proof language of Section 6 the proof tree induced by a demanded term is finite. A demanded predicate $\mathsf{nonzero}(e)$ can be established by an assumption equal to it, by a literal when e is a literal, or, when $e = (\mathsf{mul}\ a\ b)$, by a product of proofs of $\mathsf{nonzero}(a)$ and $\mathsf{nonzero}(b)$. The search therefore terminates after exploring at most one product split per multiplication node of e together with a scan of the context at each node.

9.3 Algorithm C: Verify and Commit

The coordinator consumes a proposed result by validating the record graph and all object identities, and by requiring a current active attempt for its job and token and agreement among the result, candidate and certificate links. It then compares the head with the job snapshot, and if they differ it retires the attempt as stale and stops. Otherwise it runs the certificate checker against the job snapshot, requires strict decrease of the expression node count, and checks that the attempt remains active and that the head still equals the job snapshot. The new snapshot replaces one document expression, keeps its context and all other entries, sets the parent to the job snapshot, sets the revision to that of the parent plus one, and records the candidate as origin. The immutable snapshot and a receipt keyed by its identifier are published, and the head is replaced with the new snapshot identifier. Finally the accepted attempt is retired and the new head is announced.

The receipt names the candidate, the previous snapshot and the accepted successor. A prepared receipt counts as committed only when its successor is reachable from the head. Receipts are written by immutable publication before the head is replaced. A failed publication leaves a candidate or receipt available for inspection, and the head continues to determine committed history. The authority of the single coordinator covers the final comparison and replacement. A parallel checker can do preliminary verification, provided that the coordinator binds that verdict to the exact immutable job, candidate and checker identity and performs the final checks itself.

Remark 9.3 (No revival of earlier proposals). The snapshot comparison includes revision

Algorithm 4 Verify and commit.

procedure COMMIT(result r for job $J = (S, d, r, p)$ with token t, candidate c, certificate χ)
  validate the record graph of r and every object identity
  if t is not an active token for J **then**
    return
  end if
  if head $\neq S$ **then**
    retire the attempt as stale; **return**
  end if
  run the checker on (J, c, χ) against S; if it rejects, record an operational failure and **return**
  require $|T| < |E|$, where T is the expression of c and E that of d in S
  begin indivisible coordinator step
    require t still active and head $= S$
    $S' \leftarrow (n(S) + 1,\ S,\ \kappa(S),\ c,\ \delta(S)[d \mapsto (\mathrm{id}(T), \Gamma_d)])$
    PUT(S'); PUT(receipt(c, S, S'))
    REPLACEHEAD(id(S'))
  end indivisible coordinator step
  retire the accepted attempt and announce the new head
end procedure

and ancestry through its identifier. An expression that changes and later returns to the same bytes therefore cannot revive a proposal from an earlier revision, as Proposition 4.2 shows.

Remark 9.4 (Rejected candidates). Version 0.1 does not name a distinct outcome for a candidate that the checker or the progress rule rejects. This specification treats the rejection as an operational failure of the attempt, with the same retry treatment as any other failure, because a rejection establishes nothing about whether another candidate for the same job would be accepted.

9.4 Algorithm D: Recover and Determine Quiescence

On startup the coordinator acquires the lock, allocates a fresh session token, validates the head and its referenced state, and reconstructs the committed receipt view from the head ancestry. It ignores claims and worker results from earlier sessions, rebuilds the jobs for the current snapshot, and polls active workers and the spool until no work can advance. It then rechecks the current head and all job classifications under coordinator control, and reports the current snapshot identifier and the terminal classification defined in Section 11.

10 The Coordinator Transition Model

The algorithms of the previous section describe the coordinator operationally. This section restates them as a transition system over an abstract coordinator state, so that the safety properties can be stated and argued once, independently of scheduling.

Algorithm 5 Recover and determine quiescence.

```
1: procedure RECOVER
2:     acquire the coordinator lock and allocate a fresh session token
3:     validate the head and its referenced state
4:     reconstruct the committed receipt view from the ancestry of the head
5:     ignore claims and worker results from earlier sessions
6:     run RECRUIT for the current snapshot
7:     poll active workers and the spool until no work can advance
8:     recheck the current head and every job classification under coordinator control
9:     report the identifier of the current snapshot and its terminal classification
10: end procedure
```

10.1 State

The abstract coordinator state consists of the following components. The session token ς is allocated at the start of the session. The head h is the snapshot identifier read from the head pointer. The registry Reg maps each job of h to one of the statuses ready, active, nomatch, blocked and exhausted. The table Act is a partial map from attempt tokens to jobs and contains exactly the eligible tokens. The set Fen holds fenced tokens and the set Used holds every token allocated in the session. The counter $\mathrm{Cnt}(J)$ is the number of attempts made for job J, and b is the attempt budget per job. The status exhausted covers a job whose attempts or support budget are spent without a definitive result.

10.2 Transitions

Table 6 gives the transitions. Each transition is one indivisible step of the coordinator, and the coordinator takes one step at a time, so that the compare-and-replace of the head in COMMIT cannot interleave with another change of h.

10.3 Invariants

Invariant 10.1 (Published head)**.** Every object reachable from the head by reference is published, verified and immutable.

Invariant 10.2 (Linear history)**.** Whenever the head changes from h to h', the snapshot h' has parent h and revision $n(h) + 1$.

Invariant 10.3 (Provenance)**.** Every snapshot with a candidate as origin was produced from a job whose snapshot is its parent, by a candidate that the checker accepted against that parent and that satisfied the progress rule.

Invariant 10.4 (Fencing and uniqueness)**.** The sets Act and Fen are disjoint, no token is allocated twice, and at most one token of Act maps to any job.

Theorem 10.5 (Safety)**.** *Invariants 10.1 to 10.4 hold in every reachable state of the transition system.*

Proof. The initial state is the state after recovery, in which the head has been validated, Act and Fen are empty, and no token has been allocated. Each transition is checked in turn. Recruit and Stale only move tokens from Act to Fen or retire them, which preserves

Transition	Guard	Effect
Recruit	startup, or h differs from the head pointer	$h \leftarrow$ head pointer; Reg $\leftarrow$ enumeration of h (Algorithm A); every token of Act whose job snapshot differs from h moves to Fen
Launch	$\mathrm{Reg}(J) =$ ready, no token of Act maps to J, $\mathrm{Cnt}(J) < b$	choose $t \notin$ Used; reserve the claim; $\mathrm{Act}(t) \leftarrow J$; Used $\leftarrow$ Used $\cup \{t\}$; increment $\mathrm{Cnt}(J)$; $\mathrm{Reg}(J) \leftarrow$ active
NoMatch	$t \in$ Act with job J; result is `(no-match)`; the match test confirms $\neg M_r$	retire t; $\mathrm{Reg}(J) \leftarrow$ nomatch
Block	$t \in$ Act with job J; result is `(blocked ...)` with a valid obligation	retire t; $\mathrm{Reg}(J) \leftarrow$ blocked until a support record resumes it or the support budget is spent
Commit	$t \in$ Act with job J at snapshot h; valid record graph; checker accepts; progress rule holds	publish S' and the receipt; replace the head; retire t; then Recruit
Stale	$t \in$ Act with job snapshot different from h	retire t; record `(stale)`
Fence	$t \in$ Act and the attempt exceeds its duration limit	Fen $\leftarrow$ Fen $\cup \{t\}$; remove t from Act; request termination; set $\mathrm{Reg}(J)$ to ready if $\mathrm{Cnt}(J) < b$, else exhausted
Import	an external transaction is requested	publish the new snapshot with origin `import`; replace the head; then Recruit

Table 6: Transitions of the coordinator.

disjointness. Launch chooses a token outside Used, so no token is allocated twice, and its guard requires that no token of Act already maps to the job, which preserves uniqueness (compare Proposition 8.1). Fence moves a token from Act to Fen, and any later token for the job is allocated by Launch only after this move. For Commit, the guard and the indivisible step together require that the head still equals the job snapshot, that the token is still in Act, and that the checker and the progress rule have accepted the candidate against that snapshot, so Invariant 10.3 holds for the new snapshot. The new snapshot has the job snapshot, which equals the head, as parent and one more than its revision, and this establishes Invariant 10.2. Both the snapshot and the receipt are published by Put before ReplaceHead is applied, and every object they reference already existed or was published earlier, so Invariant 10.1 holds after the rename and was not disturbed before it. Import has the same shape as Commit with the import origin and no checker step. The lease ensures that no second coordinator takes any of these steps concurrently, since the operating system releases the lock only when the holding process ends. □

Theorem 10.6 (Preservation of meaning along rewrite histories)**.** *Let S_a be an ancestor of S_b in the committed chain, let every snapshot strictly after S_a up to S_b have a candidate as origin, and let d be a document of S_a with context Γ. Then the expression of d in S_a is equivalent under Γ to its expression in S_b.*

Proof. By induction on the number of commits between S_a and S_b. In the step, the snapshot has a candidate as origin, so by Invariant 10.3 the checker accepted the candidate against its parent. If the rewritten document is d, then Theorem 6.7 gives equivalence of the two

expressions under the context of d in the parent, which is Γ because a rewrite commit keeps the context of the document. If another document was rewritten, the expression of d is unchanged. Since $\approx_\Gamma$ is an equivalence relation, transitivity composes the steps. The configuration is the same throughout because a rewrite commit keeps the configuration identifier of the parent. □

An assumption change starts a new context for subsequent judgments, and Theorem 10.6 does not cross an import. A consequence of Invariant 10.3 and of content addressing is that any party holding the object store can audit a committed chain by replaying the checker over each non-import snapshot and its candidate, certificate and job, without access to the workers that produced them.

11 Quiescence and Terminal Classification

11.1 Job Status at a Snapshot

Fix a snapshot S and configuration κ. Each job of S has, at the moment the coordinator decides that no work can advance, one of the following statuses. A job is *confirmed no-match* when the catalog's match test establishes that the rule has no applicable instance at the path. It is *blocked* when an attempt returned an obligation and the support search for it was exhausted. It is *exhausted* when its attempts or support budget were spent without a definitive result, as after repeated worker crashes, timeouts, rejected candidates or resource cutoffs. A job that is ready, active, or the subject of a pending support request is *unfinished*.

Definition 11.1 (Complete enumeration)**.** The enumeration of S is *complete* when Algorithm A has registered all $\rho\,|S|$ jobs of S without interruption by a crash, a limit, or an unavailable checker.

11.2 Classification

Definition 11.2 (Terminal classification)**.** Let the coordinator be quiescent at snapshot S, with no ready job, no live attempt and no pending support request. The terminal classification of S under κ is defined by the first of the following cases that applies. The classification is *incomplete* when the enumeration of S is not complete, or some job is exhausted, or the checker is unavailable, or a resource cutoff leaves the applicability of some rule undecided, and an incomplete report may also list the blocked obligations that are known. Otherwise it is *blocked* when the enumeration is complete, no job is exhausted, and at least one job is blocked, so that all unfinished alternatives are known and include unsupported proof obligations. Otherwise the enumeration is complete and every job is confirmed no-match, and the classification is *fixed point*.

For an empty workspace or an empty enabled rule set, complete enumeration registers no jobs, and the classification is a vacuous fixed point. These classifications describe one snapshot under one configuration. A subsequent import or configuration transaction creates a new snapshot and therefore a new question.

Proposition 11.3 (Totality and exclusivity)**.** *At a quiescent snapshot exactly one classification applies.*

Crash point	Objects present after restart	Authoritative head
Before any publication of the commit	none of the commit	S
After the successor is published, before the receipt	successor, unreachable	S
After the receipt, before the rename	successor and receipt, unreachable	S
During the rename	old pointer or new pointer, never a mixture	S or S'
After the rename, before the announcement	successor and receipt, reachable	S'
After the announcement	as above	S'

Table 7: Effect of a crash at each point of a commit from S to S'.

Proof. At quiescence every job is confirmed no-match, blocked, or exhausted, because no job is unfinished. The cases of Definition 11.2 are ordered, so at most one applies. If the enumeration is incomplete or some job is exhausted, the first case applies. Otherwise every job is confirmed no-match or blocked, and the second case applies if some job is blocked and the third if none is. □

The order of the cases carries the conservative principle of the design: an undecided applicability can never be reported as irreducibility. A fixed point is a positive claim that every rule has been shown not to apply at every node, and it can be established only from confirmed outcomes. A failed worker cannot supply one.

12 Failure Model and Guarantees

12.1 Resource Limits

The coordinator enforces finite limits on record size, parse depth, arithmetic digit count, live worker count, attempt duration and total retries. A limit breach produces an incomplete outcome if it leaves applicable work undecided. Arithmetic is exact, either with checked input bounds or with a dedicated arbitrary-precision evaluator. Bash's machine integer arithmetic is suitable for bounded orchestration counters and for nothing else. Worker errors preserve their diagnostics in staging, and the authoritative outcome is expressed through the record protocol, so that a diagnostic that exists only in a log cannot influence a classification.

12.2 Crash Analysis

A commit consists of a fixed sequence of publication steps, and the effect of a crash depends only on where in the sequence it falls. Table 7 gives the state that a restarted coordinator finds. In every row the previous head is authoritative until the rename completes, and the successor is authoritative afterwards, regardless of which auxiliary objects exist and regardless of whether an announcement was sent.

Proposition 12.1 (Head authority)**.** *After any crash the head pointer names either S or S'. A fully prepared successor can exist without publication, and therefore the head cannot be reconstructed by selecting the snapshot of largest revision on disk.*

Proof. The pointer is changed only by an atomic rename of a complete temporary file, so a restarted reader finds the old pointer or the new one. In the second and third rows of Table 7 the successor snapshot has revision $n(S) + 1$ and is present on disk, yet it is not the head. A rule that chose the largest revision would name the unpublished successor and would thereby publish a commit that the coordinator never completed, and that a competing proposal from the same snapshot might have preempted. □

In the durable profile, a commit that has been acknowledged, which is to say one whose pointer file and containing directory have been flushed after the rename, survives a machine crash. A machine crash before the acknowledgement leaves one of the rows of Table 7, because the flush ordering of Section 8 ensures that no reachable reference precedes the bytes it names. In the process-crash-only profile the table holds for process crashes, and a machine crash may lose or reorder unflushed writes, which is why the profile must be declared at startup.

12.3 Guarantees

Each visible head references fully published immutable state, by Invariant 10.1. The parent link serializes successful mutations, by Invariant 10.2. For rewrite-only history at a fixed context and configuration, transitivity of checked equivalence preserves the partial rational function of every document across the sequence, by Theorem 10.6. A claim guarantees at most one live eligible token per job within a session, by Proposition 8.1, though a fenced worker may still be physically running. The protocol tolerates duplicate computation while making acceptance serial. A spool notification can be dropped and reconstructed, by Proposition 8.2, whereas the head cannot be reconstructed from the snapshots on disk, by Proposition 12.1.

If the filesystem becomes unreadable, or if an authoritative object fails an identity or grammar check, the coordinator must stop and surface the invalid state, and must not attempt to continue by repair. An implementation may offer a separate recovery tool, but a tool of that kind is outside the protocol and has no authority to alter the head.

Theorem 12.2 (Termination of a session)**.** *Assume that expressions and the rule catalog are finite, that external input is fixed during the session, that the attempt budget b, the support budget and the retry budget are finite, and that every attempt and every support search has bounded duration. Then a coordinator session performs a finite number of attempts and a finite number of accepted rewrites, and it reaches a quiescent state and reports a terminal classification.*

Proof. By Theorem 7.3 the number of accepted rewrites is at most $N - m$. At each of the at most $N - m + 1$ snapshots, Lemma 9.1 bounds the number of jobs by ρN, and each job receives at most b attempts, each of bounded duration, so that the number of attempts at a snapshot, and with it the time spent at that snapshot, is finite. Support requests are registered once per job and predicate and are bounded by the support budget. When no ready job, live attempt or pending support request remains, Proposition 11.3 gives a classification. □

These assumptions give a finite number of attempts per version and a finite number of accepted rewrites. A run can terminate with unresolved obligations or operational failures, and the classification then reports blocked or incomplete and not fixed point.

13 Conformance

An implementation conforms to version 0.1 of this specification if it satisfies every requirement stated in the normative text, declares at startup whether it provides the durable profile or the process-crash-only profile, and exhibits the behaviour described by each of the following conformance cases under the baseline catalog. The cases are stated as observable behaviour at the level of snapshots, heads, receipts and classifications, so that they can be exercised by a test harness that injects crashes, delays and duplicate announcements. Reference identifiers for the objects of the first case are given in Appendix A.

Conformance Case 1 (Obligation, import and fixed point)**.** Let the workspace contain the document `sample` with expression `(div (var x) (var x))` and the empty context. Recruitment registers twelve jobs, three paths with four rules each, of which eleven are confirmed no-match and one, the rule `self-div` at the empty path, is ready. The worker recognizes the form and demands $\mathsf{nonzero}(x)$. The support search exhausts the available proofs, the head remains at revision zero, and the coordinator reports the classification blocked together with the obligation. Importing the context `(context (nonzero (var x)))` creates a snapshot of revision one with origin `import`, and a new set of twelve jobs whose identifiers differ from the earlier ones. The worker now builds the derivation `(self-div (assumed 0))`. The checker accepts the replacement `(rat 1 1)`, the coordinator verifies that the size decreases from three to one, and the head moves to a snapshot of revision two whose origin is the candidate. Enumeration at that head registers four jobs, all confirmed no-match, and the classification is fixed point.

Conformance Case 2 (Competing rewrites)**.** Let the document be `(add (mul (var x) (rat 1 1)) (rat 0 1))`, of size five, so that recruitment registers twenty jobs. Exactly two match: `mul-one-right` at the path to the multiplication, and `add-zero-right` at the empty path. Two workers can propose eliminating the multiplication or the addition from the same snapshot. After one succeeds, the other result is stale and is retired without effect. Recruitment under the next head finishes the remaining simplification, and the final expression is `(var x)` after two commits, within the bound of four given by Theorem 7.3. Each receipt names the snapshot from which its accepted candidate was checked, so that the first receipt names revision zero and the second names revision one.

Conformance Case 3 (Undefinedness)**.** Let the document be `(mul (div (rat 1 1) (rat 0 1)) (rat 0 1))`. The expression is undefined, and the baseline catalog admits no rewrite of it: all twenty jobs are confirmed no-match, including `rat-fold` at the division, whose divisor is a zero literal, and the classification is fixed point. A test harness that submits to the checker the triple consisting of the root job, a certificate with derivation `(rat-fold)`, and the replacement `(rat 0 1)` must observe rejection, because the subterm does not match the pattern of the rule, and a rewrite to zero would in any case change an undefined value into a defined one.

Conformance Case 4 (Scope confinement)**.** Let the document be `(add (mul (var x) (rat 1 1)) (var y))` and let the job be `mul-one-right` at the path to the multiplication.

A candidate whose expression is `(add (var x) (var y))` is accepted. A candidate that carries the same certificate but the expression `(add (var x) (var z))` has the same size and satisfies the progress rule, and is nevertheless rejected, because its canonical bytes differ from the replacement computed by the checker outside the certified occurrence.

Conformance Case 5 (Crash before head replacement)**.** Terminate the coordinator after the successor snapshot and the receipt of a commit have been published and before the head pointer is replaced. On restart the head names the previous snapshot, the successor and the receipt are present but unreachable, the coordinator acquires a fresh session, ignores earlier claims and results, and rebuilds jobs for the previous snapshot. An implementation that selects the snapshot of largest revision on disk is nonconforming.

Conformance Case 6 (Crash after head replacement)**.** Terminate the coordinator after the head pointer has been replaced and before the new head has been announced. On restart the head names the successor, recruitment proceeds from it, and no announcement is needed.

Conformance Case 7 (Delayed result through a fenced token)**.** Let a worker hold token t for a job and exceed its attempt duration. The coordinator fences t, requests termination, and allocates a fresh token t'. Suppose the worker for t' succeeds and its commit is accepted, and the worker for t, which is still running, then publishes a proposed result. The result through t is not accepted, the committed chain contains exactly one commit for the job, and the receipt of that commit cites t' through its candidate.

Conformance Case 8 (Duplicate and reordered announcements)**.** Deliver each announcement three times and in reverse order of publication, and compare the committed chain, meaning the sequence of heads and the receipts, with that of a run in which each announcement is delivered once. The two chains are identical.

Conformance Case 9 (Exhausted retries)**.** Let the document be `(add (var x) (rat 0 1))`, for which eleven of twelve jobs are confirmed no-match and one, `add-zero-right` at the empty path, matches. Terminate the worker on every attempt until the attempt budget is spent. The coordinator reports incomplete, and does not report fixed point, although every other job has returned no-match.

Conformance Case 10 (Return to identical bytes)**.** Let a snapshot S_a contain a document with expression e, and let two successive imports first replace e by another expression and then restore e with the same context. The resulting snapshot has the same document map as S_a and a different identifier and revision. A proposal checked against S_a and delivered after the second import is retired as stale.

Conformance Case 11 (Restart with live claims from an earlier session)**.** Terminate the coordinator while claim directories exist and workers are running. On restart the coordinator allocates a new session token and launches new attempts for the ready jobs of the current snapshot despite the presence of the old claim directories, and results published by the earlier workers are ignored.

Conformance Case 12 (Identifier collision)**.** Place at the destination of an identifier i a file whose bytes differ from the record whose identifier is i, and then publish that record. Publication halts, the condition is surfaced, and the head is unchanged.

Conformance Case 13 (Arithmetic bound)**.** Let the document contain a `rat-fold` instance whose literals have more digits than the configured arithmetic limit. The implementation performs no unchecked fixed-width arithmetic, produces no fold, and after the attempt budget reports incomplete.

14 Relation to Prior Work

The rewriting relation of the baseline catalog is an ordinary terminating term rewriting system, enriched by the use of a context to discharge side conditions, and the questions of termination and confluence treated in Section 7 are those of the standard theory [5, 1, 13]. The division of labour between a search procedure of unknown reliability and a small checker of proof objects has the structure of proof-carrying code [12], and it reflects the long-standing practice in computer mathematics of trusting only a kernel small enough to be examined [2]. The recruitment structure, in which workers react to facts that have been left in a shared place and coordinate through their effects on it, is an instance of the coordination through a common medium that Grassé called stigmergy [8] and that tuple-space systems implement computationally [7]. The hash-linked chain of snapshots belongs to the family of authenticated data structures that descends from Merkle's hash trees [11], although here the hash chain provides identity and auditability and not a signature scheme. The replacement of the head conditional on the job snapshot is optimistic concurrency control with validation at commit [10], with the coordinator as the single validator. The lease taken on a stable inode is a local analogue of the coarse-grained locks of distributed lock services [4], and the fencing of tokens addresses the hazard, familiar from those services, that a party whose lease has lapsed may continue to act [9].

15 Limitations and Extensions

The proof language is deliberately small, and Section 6 exhibits true predicates that it cannot derive. Enlarging it, for instance with constructors for negation and division, is a matter of defining a new checker identity and proving the analogue of Lemma 6.6 for the new constructors. Enlarging the rule catalog requires, for each new rule, a soundness argument of the kind given in Lemma 6.5, and a progress argument. A rule that does not decrease the node count, such as one that distributes multiplication over addition, needs a different measure, and Theorem 7.3 then does not apply without a new bound.

The baseline commits the first valid candidate and promises no confluence. The ordered commit policy of Section 7 restores timing independence at the price of throughput, and a catalog for which local confluence has been established would make the order irrelevant to the final snapshot. The baseline rebuilds all jobs at every head advancement, and an incremental scheduler would reduce the cost of rescanning, provided that it reuses a result only on evidence that all of its inputs are identical. Version 0.1 defines receipts only for rewrite commits, and a later version may define a receipt for external transactions so that imports can be audited in the same manner as rewrites. A rejected candidate is treated as an operational failure, and a later version may introduce a distinct outcome for it so that the diagnosis of a faulty worker is separate from that of a crashed one.

The trust model is that of cooperating processes in one account on one machine. Extending the design to several hosts would require a replacement for the process-held lock,

a replicated or consensus-backed head, and a rethinking of the atomicity assumptions on which Section 8 rests, and none of the guarantees of Section 12 should be assumed for such a deployment. Finally, every classification is relative to one snapshot under one configuration, so that the report of a fixed point is a statement about a particular question and is not invalidated, nor preserved, by a later import.

A Reference Vectors for the Worked Trace

This appendix gives the canonical objects of Conformance Case 1, with identifiers computed by the procedure of Section 5. The rule catalog identity and the checker identity, which the grammar leaves to the implementation, are taken for the purpose of these vectors to be the SHA-256 digests of the ASCII strings `baseline-catalog-0.1` and `baseline-checker-0.1`, without terminating line feed. They are illustrative and have no normative status. The attempt tokens are T_1, which consists of thirty-two digits `1`, and T_2, which consists of thirty-two digits `2`. In the displayed text of each object, an identifier is abbreviated to a number sign followed by its first eight digits, and a token is written T_1 or T_2 in typewriter font. The full identifier of the object is given beneath its text. Every identifier is the SHA-256 digest of the canonical bytes of the object including the terminating line feed, and any entry can be checked by supplying the unabbreviated record, with its line feed, to a standard hash utility.

The objects appear in order of creation. The expression e_0 and the contexts Γ_0 and Γ_1 are the document expression and the two contexts of the trace, and κ is the configuration. The snapshot S_0 is initial, and j_1 is the job at the empty path for `self-div`, which yields the obligation ω_1. The import of Γ_1 creates S_1, and the job j_2 is the same quadruple at the new snapshot. The expression e_1 is the literal one, χ is the certificate, c the candidate, r the result, S_2 the snapshot produced by the commit, ρ the receipt, and h the head record that the coordinator installs.

e_0 `(div (var x) (var x))`

`1bb3127aef6da25b220c31db911b5c1677b06b13868a8dafb1d6fdd45c42e5a9`

Γ_0 `(context)`

`9dce0b2a3e8d5a0496858e403c5b0d967c49d3557b9ed3367faec342fa99b59e`

Γ_1 `(context (nonzero (var x)))`

`375c602e55104f5bc760a3c5053779ff17810467aa06ccefb601475d7d79be8c`

κ `(config #d123c37d #b0341c22 add-zero-right mul-one-right rat-fold self-div)`

`8f4d955f48389e4ad8b69857745ffd5788d20ab37d18b0288b6807824f05d581`

S_0 `(snapshot 0 root #8f4d955f import (doc sample #1bb3127a #9dce0b2a))`

`0dcd6da0cf7e67686fc29caf9e7b1dbc98c04b72a4fc5929529dd640524e805a`

j_1 `(job #0dcd6da0 sample self-div (path))`

`33367d513e6673b1674962fef866d289a456a67029ee33a53220a66aa8dabba2`

ω_1 `(obligation #33367d51 T1 (nonzero (var x)))`

`a729d7326ea6ab6efbe16b0e561d2f89e6a9721abc481330d246a12075e0eee6`

S_1 `(snapshot 1 #0dcd6da0 #8f4d955f import (doc sample #1bb3127a #375c602e))`

`24276d0fabd4ed612372b09bd8793aac5b30f3ce94a6ab74bc895f7d21e7030b`

j_2 `(job #24276d0f sample self-div (path))`

`f47e889a3170ff1777f41c593c9bf8cde79ae819d9813373fb73bfdbe8f4b2d1`

e_1 `(rat 1 1)`

`99ddb0389cab5a9543ca3fb89d2102bbbde8758a20c33128d0d9c24d262253cc`

χ `(cert #f47e889a #99ddb038 (self-div (assumed 0)))`

`733914812b66213ba2295c5ba20203e781eb95d431bfd988def55882717b477a`

c `(candidate #f47e889a T2 #99ddb038 #73391481)`

`636be26345a16eb3046466de60f43ba791d4e42e98d33dd2a44b21f86643847b`

r `(result #f47e889a T2 (proposed #636be263))`

`fba45974b6985186f07afd2280a9a7a5b206e334f9606b3cb7d98337e0c42ccf`

S_2 `(snapshot 2 #24276d0f #8f4d955f #636be263 (doc sample #99ddb038 #375c602e))`

`9b6f1bd1153035296a7020120b1ba844b86a02935a19eac26b4f88c91a3bc200`

ρ `(receipt #636be263 #24276d0f #9b6f1bd1)`

`3f77ea22c99fa130b8808a12bb20e5552d5a35e9e5da93f85ea705f3d2965b6b`

h `(head #9b6f1bd1)`

`1a744d425ab06b3a513f58d7965950e8622c9863e5473df13c5d0bb71e1898ae`

References

[1] F. Baader and T. Nipkow, *Term Rewriting and All That*, Cambridge University Press, 1998.

[2] H. Barendregt and F. Wiedijk, The Challenge of Computer Mathematics, *Philosophical Transactions of the Royal Society A* 363(1835), 2351–2375, 2005.

[3] S. Bradner, Key Words for Use in RFCs to Indicate Requirement Levels, RFC 2119 (BCP 14), Internet Engineering Task Force, 1997.

[4] M. Burrows, The Chubby Lock Service for Loosely-Coupled Distributed Systems, in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 335–350, 2006.

[5] N. Dershowitz and J.-P. Jouannaud, Rewrite Systems, in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier, 243–320, 1990.

[6] National Institute of Standards and Technology, Secure Hash Standard (SHS), Federal Information Processing Standards Publication 180-4, 2015.

[7] D. Gelernter, Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems* 7(1), 80–112, 1985.

[8] P.-P. Grassé, La Reconstruction du Nid et les Coordinations Interindividuelles chez Bellicositermes Natalensis et Cubitermes sp., *Insectes Sociaux* 6(1), 41–80, 1959.

[9] M. Kleppmann, *Designing Data-Intensive Applications*, O'Reilly Media, 2017.

[10] H. T. Kung and J. T. Robinson, On Optimistic Methods for Concurrency Control, *ACM Transactions on Database Systems* 6(2), 213–226, 1981.

[11] R. C. Merkle, A Digital Signature Based on a Conventional Encryption Function, in *Advances in Cryptology, CRYPTO '87*, Lecture Notes in Computer Science 293, Springer, 369–378, 1988.

[12] G. C. Necula, Proof-Carrying Code, in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 106–119, 1997.

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

[14] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications, in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 433–448, 2014.