Beyond Parentheses:
A Geometric Foundation for Concurrent Computation through the Spherepop Calculus

Evaluation as the Construction of Irreversible Computational Histories

Flyxion

June 2026

Abstract

Traditional programming languages inherit a conception of computation in which scope is represented syntactically through nested parentheses, braces, and textual delimiters. Evaluation proceeds by repeatedly locating the innermost reducible expression and replacing it with its value, while the intermediate computational history is largely discarded. Although this model has served programming language theory for decades, it obscures the geometric structure of dependency, limits explicit reasoning about provenance, and separates operational execution from the persistent record of computation.

This monograph develops the Spherepop Calculus (SPC), a history-oriented computational framework in which scope is represented as nested geometric regions rather than purely textual syntax. Computational abstractions are modeled as embedded spheres within a computational plenum, evaluation becomes the geometric collapse of boundaries through the Pop operator, and execution constructs an irreversible history that remains a first-class mathematical object throughout computation. Replay reconstructs admissible historical context, Refusal prevents incoherent continuations before commitment, and Collapse records terminal computational outcomes while preserving provenance.

The resulting framework unifies operational semantics, denotational semantics, category theory, graph rewriting, compiler construction, concurrent execution, probabilistic computation, differentiable programming, and geometric reasoning within a common historical model. Logic becomes an operator library rather than a fixed execution engine, allowing Boolean, fuzzy, probabilistic, differentiable, and other computational regimes to share a single composition-first architecture. Histories replace mutable state as the primary semantic object, while dependency graphs, hyperplane boundaries, and manifold geometry provide complementary mathematical interpretations of execution.

The later chapters establish the Spherepop Calculus as a formal computational system through type theory, abstract machines, rewrite systems, categorical semantics, algebraic structures, geometric semantics, complexity analysis, and a collection of mathematical conjectures intended to guide future research. Rather than viewing computation as the manipulation of transient symbolic expressions, the Spherepop perspective treats computation as the construction, transformation, and geometric organization of persistent histories.

Contents

1 Introduction

Every student first encounters symbolic computation through arithmetic. An expression such as

$$3(2 + (5 \times 4)) - 7$$

is not evaluated all at once. Instead, one repeatedly identifies the innermost scope, computes its value, substitutes the result, and continues until the entire expression has been reduced.

The algorithm is so familiar that it is rarely regarded as a profound observation about computation itself. Yet nearly every programming language, proof assistant, compiler, symbolic algebra package, and logical calculus ultimately performs some variation of exactly this procedure.

The central claim of Spherepop is that this simple algorithm deserves to be regarded as the primitive operation of computation.

Computation is not fundamentally the manipulation of variables, nor the evaluation of lambda terms, nor the execution of instructions. It is the repeated discovery of an admissible scope, the evaluation of that scope, and the replacement of the scope by its result.

Everything else is structure built around this operation.

2 The Parenthesis Algorithm

Consider

$$((2 + 3) \times (4 + 5)).$$

A conventional interpreter performs

$$((2 + 3) \times (4 + 5)) \to (5 \times 9) \to 45.$$

Notice that the computation always follows the same pattern.

First, locate a scope.

Second, evaluate that scope.

Third, replace the scope by its result.

Repeat until no scopes remain.

Spherepop simply elevates this familiar algorithm into the primary mathematical object.

Instead of viewing parentheses as temporary syntax, it views every scope as an explicit computational object that is born, evaluated, recorded, and finally dissolved.

3 Why "Sphere" and "Pop"?

A pair of parentheses defines a region of computation.

Spherepop interprets this region literally as a sphere of abstraction.

Opening a parenthesis creates a sphere.

Closing the parenthesis seals that sphere.

Evaluation pops the sphere, replacing the entire region by the value produced inside it.

For example,

$$((2+3)\times(4+5))$$

may be pictured as two nested spheres.

The first sphere pops,

$$((2+3)\times(4+5))\to(5\times(4+5)),$$

then the second,

$$(5\times(4+5))\to(5\times 9),$$

and finally the outer sphere,

$$(5\times 9)\to 45.$$

The familiar arithmetic algorithm therefore becomes a sequence of sphere creations and sphere dissolutions.

4 The Missing Object

Traditional programming languages preserve only the current state.

Once

$$2 + 3 = 5$$

has been computed, the computation that produced the value is normally discarded.

Spherepop asks a different question.

What if the computation itself were never forgotten?

Instead of storing only values, suppose every scope, every evaluation, every dependency, and every replacement were preserved as part of an irreversible history.

The primitive object would no longer be the current expression.

It would be the complete record of how the expression came to exist.

5 Arithmetic as the First Programming Language

Long before students encounter programming languages, automata, or formal logic, they learn a remarkably sophisticated computational procedure through elementary arithmetic. Every expression containing parentheses implicitly defines a computation, and every student learns to execute that computation by following a deterministic evaluation strategy.

Consider the familiar expression

$$3\,(2 + (5 \times 4)) - 7.$$

No student attempts to evaluate every operation simultaneously. Instead, one repeatedly locates the innermost parenthesized expression, computes its value, substitutes the result into the surrounding expression, and repeats until no unresolved scopes remain.

The evaluation proceeds as

$$3\,(2 + (5 \times 4)) - 7$$

$$\longrightarrow 3(2 + 20) - 7$$

$$\longrightarrow 3(22) - 7$$

$$\longrightarrow 66 - 7$$

$$\longrightarrow 59.$$

This algorithm is so commonplace that its significance is often overlooked. Yet it contains the essential structure of symbolic computation.

At every stage three operations occur.

First, an admissible scope is identified.

Second, the computation contained within that scope is evaluated.

Third, the scope itself disappears, replaced by the value it produced.

This procedure appears throughout mathematics.

Polynomial evaluation proceeds by reducing nested expressions.

Matrix expressions evaluate inner products before outer compositions.

Logical formulas reduce subformulas before larger propositions.

Programming languages evaluate nested function applications.

Proof assistants normalize expressions by repeatedly reducing local redexes.

Although the syntax differs, the underlying computational strategy remains remarkably consistent.

This observation motivates the central question of this work.

What exactly is the primitive operation of symbolic computation?

Traditional foundations answer this question differently.

The λ-calculus identifies abstraction and application as primitive.

Combinatory logic eliminates variables entirely.

Term rewriting systems emphasize rewrite relations.

Abstract machines describe transitions between machine states.

Operational semantics specifies reduction rules over syntax.

Each perspective captures an important aspect of computation, yet they all inherit a common assumption: computation is fundamentally manipulation of symbolic expressions.

Spherepop proposes a different starting point.

The primitive operation is not the manipulation of syntax itself but the evaluation of a bounded computational region.

Parentheses are therefore not merely punctuation. They identify regions of computation that temporarily isolate local structure from the surrounding expression. Evaluation consists of selecting one of these regions, resolving the compu-

tation contained within it, and replacing the entire region by its result.

This shift may initially appear cosmetic. After all, every interpreter already behaves in this manner.

However, a closer inspection reveals that conventional formalisms treat these regions as temporary implementation details. Once evaluation completes, the scope disappears completely. Only the resulting value survives.

Spherepop instead asks whether the scope itself should be regarded as a first-class mathematical object.

Rather than viewing an expression as a sequence of symbols decorated by parentheses, Spherepop views it as a collection of nested computational regions whose creation, evolution, interaction, and dissolution constitute the computation itself.

In this interpretation, evaluation is no longer simply the replacement of one term by another.

It is the controlled collapse of a bounded computational region.

The history of these collapses, together with the dependencies they create, forms the persistent computational object.

The remainder of this paper develops this idea systematically.

We begin not with advanced category theory or dependent type systems, but with the elementary notion of scope itself. By progressively enriching the mathematics of nested computational regions, we show that abstraction, application, concurrency, probabilistic choice, typing, provenance, and dependent computation can all be understood as natural extensions of the same elementary process first encountered in arithmetic: locating an admissible scope, evaluating it, and continuing until no unresolved computational regions remain.

6 Parentheses, Scope, and Evaluation

Having recognized that elementary arithmetic already embodies a complete evaluation strategy, we now examine the mathematical role played by parentheses themselves. Surprisingly, parentheses contribute nothing to the numerical result of a computation. They exist solely to delimit regions within which particular computational rules are temporarily privileged.

For example, the expressions

$$2 + 3 \times 4$$

and

$$(2 + 3) \times 4$$

contain precisely the same symbols. The only difference is the placement of a single pair of parentheses, yet this alteration fundamentally changes the evaluation order and therefore the resulting computation.

The parenthesis does not perform arithmetic.

It performs organization.

It defines a temporary computational environment whose contents must be resolved before the surrounding computation may continue.

One may therefore think of every pair of parentheses as creating a local workspace. Variables introduced inside this workspace remain invisible outside it until the workspace has been completely evaluated. Intermediate values may be created, transformed, or discarded entirely within its boundary. Only the final result escapes.

This observation extends far beyond elementary arithmetic.

In the λ-calculus,

$$(\lambda x.t)\ u$$

defines a computational region that cannot be reduced until the function and its argument have both become sufficiently explicit. Likewise, in modern programming languages, every function body, block statement, closure, module, or namespace defines a computational scope that regulates visibility, evaluation, and dependency.

Although these constructs appear different, they all perform the same fundamental task.

They establish a boundary.

The traditional presentation of these ideas is entirely syntactic. Compilers typically represent expressions as abstract syntax trees whose internal nodes correspond to operators and whose leaves correspond to atomic values. Evaluation proceeds by recursively traversing these trees according to their grammatical structure.

This representation is both elegant and efficient, but it encourages an important misconception.

It suggests that computation is fundamentally about symbols.

Spherepop instead argues that syntax is merely one possible encoding of a more primitive object.

The primitive object is the bounded computational region itself.

Parentheses are therefore not essential because they are symbols.

They are essential because they identify where one computation temporarily separates from another.

To emphasize this distinction, consider the expression

$$((1 + 2) + 3) + 4.$$

Conventionally, this expression is viewed as a nested sequence of parentheses.

Spherepop proposes a different interpretation.

Rather than seeing four textual delimiters, we see four nested computational regions.

The innermost region contains

$$1 + 2.$$

Once evaluated, that region ceases to exist as an unresolved computation and is replaced by its result.

The surrounding region now becomes the innermost unresolved computation.

Its evaluation proceeds in exactly the same manner.

The process repeats until every computational region has been resolved.

This viewpoint naturally suggests a geometric interpretation.

Instead of imagining parentheses as punctuation marks written on a page, we may regard each scope as a bounded region occupying its own local space. Larger regions contain smaller ones, and evaluation always proceeds from the most deeply nested admissible region outward.

This geometric perspective has several immediate advantages.

First, containment becomes an intrinsic property rather than a syntactic artifact. A computational region literally contains its local variables, temporary constructions, and intermediate states.

Second, locality becomes explicit. Every computation occurs within a bounded environment whose effects are initially confined to that region.

Third, evaluation acquires a natural direction. Rather than recursively descending through textual syntax, computation becomes the progressive resolu-

tion of nested regions from the interior toward the exterior.

Most importantly, this interpretation prepares the transition from symbolic syntax to computational history.

In conventional semantics, once an inner scope has been evaluated, it simply disappears. The computation proceeds as though the intermediate region had never existed.

Spherepop takes the opposite position.

The disappearance of a computational region is itself a computational event.

The opening of a scope, the transformations occurring within it, and its final resolution together form an indivisible episode in the history of the computation.

Rather than treating scope as temporary punctuation, Spherepop elevates it to the primary structural object from which evaluation, dependency, provenance, and ultimately computation itself are constructed.

The next section formalizes this intuition by replacing purely syntactic scope with an explicit geometric object: the *Sphere*. Rather than representing scope by punctuation, the Sphere represents computation as the creation of a bounded region whose eventual resolution becomes the fundamental operation of the calculus.

7 From Parentheses to Spheres

The transition from conventional symbolic notation to the Spherepop Calculus begins with a simple observation. Parentheses themselves are not the objects being computed. They are merely symbols used to indicate where computation is temporarily localized.

If two mathematicians evaluate the same expression written in different notations, the underlying computation remains unchanged. Whether one writes

$$((2 + 3) \times (4 + 5))$$

or draws a syntax tree, or uses indentation, or constructs an abstract syntax graph, the numerical result is identical. The notation has changed, but the computational structure has not.

This suggests that parentheses are not fundamental mathematical objects. Instead, they are one possible representation of a more primitive notion: bounded computational scope.

Spherepop therefore replaces the symbolic parenthesis with an explicit geometric object called a *Sphere*.

A Sphere is not intended merely as a visual metaphor. It represents a computational region whose interior possesses temporary autonomy. Every local definition, intermediate value, and dependency exists within the boundary of the Sphere until evaluation is complete.

Rather than writing

$$(a + b)$$

one may think of computation as creating a bounded region containing the operations

$$a + b.$$

The boundary itself records that this computation is temporarily isolated from its surroundings. External computations cannot directly interfere with the interior of the Sphere until the boundary has been resolved.

This interpretation separates two concepts that are often conflated in traditional programming languages.

The first is the computation itself.

The second is the boundary within which that computation occurs.

Conventional syntax represents both simultaneously using parentheses.

Spherepop treats them as distinct mathematical entities.

The computational content consists of operators, values, and dependencies.

The Sphere consists solely of the boundary that contains those objects.

Once this distinction is made, many familiar programming constructs become instances of the same geometric idea.

A function body defines a Sphere.

A lexical block defines a Sphere.

A namespace defines a Sphere.

A module defines a Sphere.

Even a process executing concurrently may be viewed as occupying its own bounded computational region.

The apparent diversity of programming constructs begins to collapse into a single geometric abstraction.

The significance of this reinterpretation becomes clearer when considering

nested computation.

Suppose we evaluate

$$((2 + 3) \times (4 + 5)).$$

Traditional notation presents this as nested parentheses.

Spherepop instead views the expression as three nested computational regions.

The outer Sphere contains two interior Spheres.

Each interior Sphere contains a local computation.

Neither interior computation depends upon the other.

Consequently, each may evolve independently until both have produced values.

Only after both interior regions have been resolved does the enclosing Sphere become admissible for evaluation.

This observation immediately exposes concurrency hidden inside ordinary arithmetic.

The additions

$$2 + 3$$

and

$$4 + 5$$

are independent.

Nothing prevents their simultaneous execution.

The enclosing multiplication merely waits until both interior Spheres have completed.

Thus sequential notation often conceals naturally parallel computational structure.

The geometry makes this independence explicit.

More generally, every Sphere possesses three distinguishable phases during its lifetime.

Initially, the Sphere is opened.

This establishes a new computational boundary together with its local environment.

During execution, the Sphere evolves internally. Intermediate values appear, disappear, and interact while remaining confined to the local region.

Finally, the Sphere is resolved.

Its boundary disappears, its interior computation is replaced by its resulting value, and its influence propagates into the surrounding computational environment.

Spherepop gives this final event a dedicated mathematical status.

Rather than treating evaluation as an anonymous rewrite rule, the calculus identifies the dissolution of a computational boundary as a primitive operation.

This operation is called a *Pop*.

The terminology reflects an intuitive geometric picture. A Sphere temporarily encloses a computation. Evaluation removes the enclosing boundary, exposing the value produced inside. The surrounding computation then continues using this newly available result.

Importantly, the Pop is not merely another name for β-reduction.

It represents a broader computational event.

Function application, arithmetic evaluation, module initialization, concurrent task completion, logical proof normalization, and probabilistic commitment may all be understood as particular instances of the same underlying operation: the dissolution of an admissible computational boundary.

By elevating bounded scope to a first-class mathematical object, Spherepop moves the emphasis of computation away from symbolic manipulation and toward the evolution of computational regions.

Expressions are no longer viewed simply as trees of symbols.

They become evolving collections of nested Spheres whose successive openings and Pops define the observable history of computation.

The next section formalizes the Pop operation itself and demonstrates that evaluation is most naturally understood as the controlled collapse of an admissible computational region rather than merely the replacement of one symbolic expression by another.

8 The Pop Primitive

Having replaced symbolic parentheses with explicit computational regions, we must now identify the fundamental operation that advances a computation. In traditional operational semantics this role is played by reduction. A redex is lo-

cated, a rewrite rule is applied, and the resulting expression replaces the original. While mathematically elegant, this description emphasizes the transformation of syntax rather than the evolution of computation itself.

Spherepop adopts a different viewpoint. The primitive event is not the rewrite of an expression but the dissolution of an admissible computational region. This event is called a *Pop*.

Intuitively, every Sphere represents a temporary suspension of commitment. Within its boundary multiple intermediate constructions may exist simultaneously. Variables may be introduced, functions applied, constraints evaluated, and dependencies accumulated. None of these internal activities are visible outside the Sphere until the boundary itself is removed. The Pop is precisely the event that removes this boundary.

Unlike a simple rewrite rule, a Pop has both local and global significance. Locally, it resolves the computation contained within a Sphere. Globally, it changes the surrounding computational landscape by making new information available to enclosing regions. Every Pop therefore represents both an evaluation and a propagation.

Formally, let

$$S = (B, I)$$

denote a Sphere consisting of a boundary B enclosing an interior computation I. If the interior has reached an admissible state, the Pop operation

$$\mathrm{Pop}(S)$$

replaces the entire Sphere by the value computed within it,

$$\mathrm{Pop}(S) \longrightarrow v.$$

Unlike conventional reduction, however, the Pop is understood as a transition between two computational histories rather than merely two symbolic expressions. The computation has not simply changed form. One bounded region of the computational world has ceased to exist, and a new history has been created.

This distinction becomes clearer when examining nested evaluation.

Consider once more

$$((2 + 3) \times (4 + 5)).$$

Initially three computational regions exist.

The two inner Spheres are immediately admissible because their computations contain no unresolved subregions.

The enclosing Sphere is not yet admissible because its operands remain unresolved.

The first Pop produces

$$(5 \times (4 + 5)).$$

The second produces

$$(5 \times 9).$$

Only now does the outer Sphere become admissible.

Its Pop finally produces

$$45.$$

Nothing about this process depends specifically upon arithmetic.

Exactly the same sequence occurs during function application,

$$(\lambda x.t)\ u,$$

during expression simplification,

during logical normalization,

during proof reduction,

and during compiler optimization.

Each case consists of identifying a computational region whose internal requirements have been satisfied, dissolving its boundary, and exposing its result to the surrounding environment.

From this perspective, evaluation is naturally viewed as a sequence of admissible Pops.

The scheduler of a Spherepop runtime therefore has a remarkably simple task. Rather than recursively traversing syntax trees looking for rewrite rules, it searches for Spheres whose internal dependencies have all been satisfied.

A Sphere is said to be *ready* whenever every prerequisite computation contained within it has already completed.

Evaluation then becomes

$$\text{Locate Ready Sphere} \longrightarrow \text{Pop} \longrightarrow \text{Update History} \longrightarrow \text{Repeat}.$$

This formulation deliberately separates readiness from execution.

Conventional interpreters often intertwine dependency analysis with reduction. Spherepop instead treats dependency analysis as the determination of which computational regions are currently admissible for Pop.

This distinction becomes particularly valuable once concurrency is introduced.

If two Spheres are independent, neither blocks the other. Both are admissible simultaneously.

Consequently,

$$((2 + 3) \times (4 + 5))$$

contains two Pops that may occur in parallel without altering the final result.

The computation therefore possesses an intrinsic concurrency that is obscured by purely textual syntax but immediately visible once computation is viewed as the evolution of nested computational regions.

The Pop primitive also provides a natural location for recording provenance. Instead of remembering only the value produced by a computation, the runtime may record when the Sphere was opened, which dependencies entered it, which computations occurred internally, and when the boundary was dissolved. Evaluation becomes an observable historical event rather than an invisible rewrite.

This historical interpretation does not alter the computed result. Arithmetic still produces the same numbers, functions still return the same values, and proofs still normalize to the same canonical forms. What changes is the object of study. The primary mathematical object is no longer the expression itself but the evolving sequence of bounded computational regions whose creation and resolution constitute the computation.

The next step is therefore to ask what should happen after a Pop has occurred. Traditional semantics simply discard the vanished scope. Spherepop instead treats every Pop as an irreversible extension of computational history. It is this transition from isolated evaluation to persistent history that gives the calculus its distinctive semantic character.

9 Histories as Persistent Scope Evolution

The Pop primitive completes the evaluation of a computational region, but it raises an important question. What becomes of the region that has been evaluated?

Traditional operational semantics answers this question by simply removing the reduced expression. Once

$$(2 + 3)$$

has become

$$5,$$

the computation that produced the value is no longer represented within the state of the program. Evaluation is therefore fundamentally ephemeral. Intermediate computational structure disappears as soon as it has served its purpose.

This design has proved extraordinarily successful for efficient execution. Nevertheless, it reflects a particular philosophical commitment: that the current computational state is more fundamental than the process that produced it.

Spherepop rejects this assumption.

Instead of treating computation as a sequence of transient states, Spherepop treats it as the construction of an irreversible history. Every Pop extends that history, and every computational object derives its meaning from the sequence of scope transformations that produced it.

The fundamental mathematical object therefore becomes a history

$$H = (e_1, e_2, \ldots, e_n),$$

where each event

$$e_i$$

records a meaningful computational transition.

Rather than representing only values, a history records the evolution of computational boundaries themselves.

A typical history may contain events such as

$$\text{Open}(S),$$

$$\mathrm{Bind}(x, v),$$

$$\mathrm{Pop}(S),$$

$$\mathrm{Collapse}(v),$$

or

$$\mathrm{Refuse}(r),$$

depending upon the operational semantics being executed.

The important observation is that these events are ordered.

Unlike an ordinary typing context, whose primary role is to record assumptions, a history records the chronology of construction. Every later event depends upon the existence of earlier events.

Consequently, histories possess an intrinsic direction.

Computation no longer moves between anonymous states.

It grows.

Every admissible computation extends an existing history,

$$H \longrightarrow H',$$

where

$$H' = H \parallel e$$

denotes the extension of the existing history by a newly admitted event e.

This monotonicity property immediately distinguishes histories from mutable machine states.

A mutable state may overwrite information.

A history cannot.

The only admissible operation is extension.

Nothing already admitted may be removed from the historical record.

This does not imply that previous values remain operationally active.

Many historical events cease to influence future computation directly. Intermediate values may become unreachable. Entire computational regions may no longer participate in subsequent evaluation.

Operational irrelevance, however, is distinct from historical nonexistence.

A value may cease to matter computationally while remaining part of the construction that produced the current program.

This distinction mirrors ordinary mathematical practice.

A proof consists of many intermediate lemmas.

Once the theorem has been established, those intermediate constructions may no longer be consulted during routine use.

Nevertheless, they remain part of the proof.

Likewise, a compiler may eliminate temporary variables after optimization.

The optimized program no longer contains them.

Yet they remain part of the developmental history of the program that was compiled.

Spherepop elevates this familiar distinction into a semantic principle.

Programs are not merely expressions.

They are histories of computational construction.

This perspective naturally introduces provenance.

Every value possesses not only an identity but also an origin.

Suppose a computation produces

$$v.$$

Traditional semantics associates v only with its current value.

Spherepop instead associates v with the history responsible for its construction,

$$\mathrm{Prov}(v) = H_v.$$

The provenance of a value therefore consists of the sequence of computational events whose successful execution admitted that value into the evolving history.

As computations become increasingly concurrent, this provenance becomes particularly valuable.

Suppose two independent Spheres execute simultaneously.

Each develops its own local history.

When both complete, the enclosing computation observes not merely two values, but two independently constructed historical trajectories that now become available for composition.

Concurrency therefore becomes the composition of histories rather than the

interleaving of machine instructions.

This interpretation provides a remarkably uniform description of sequential and parallel computation.

Sequential evaluation extends a single history.

Parallel evaluation develops multiple compatible histories whose eventual composition forms a larger historical structure.

The operational semantics therefore shifts emphasis away from instantaneous program states and toward persistent computational evolution.

Every successful computation answers two questions simultaneously.

First, what value was produced?

Second, how did that value come to exist?

Traditional semantics primarily answers the first question.

Spherepop treats the second as equally fundamental.

This historical viewpoint will become increasingly important throughout the remainder of the calculus.

Replay, dependent computation, provenance tracking, repair, admissibility, parallel scheduling, and compilation all rely not merely upon the values that exist at a given moment, but upon the histories that brought those values into existence.

The next section develops this idea further by replacing traditional syntactic substitution with the more general operation of replay, demonstrating that computation is better understood as the continuation of historical construction than as the manipulation of isolated symbolic expressions.

10 Replay as Historical Continuation

Traditional symbolic computation is founded upon one of the most successful ideas in the history of logic: substitution. Whenever a function is applied, an argument replaces a formal parameter, and computation proceeds as though the parameter had always denoted the supplied value. In the λ-calculus, this operation appears as the familiar β-reduction

$$(\lambda x.t)\ u \longrightarrow t[u/x].$$

Substitution is elegant because it reduces function application to a purely syntactic transformation. The variable disappears, the argument is copied into its place, and evaluation continues.

From the perspective developed in the preceding sections, however, substitution conceals an important aspect of computation.

Nothing in the resulting expression records how the argument arrived there.

The transformed term is mathematically equivalent to the original, yet the history that justified the transformation has vanished.

Spherepop therefore replaces substitution with a more primitive operation called *Replay*.

Replay does not merely replace one symbol with another.

Instead, it reconstructs the continuation of a computation by replaying the historical events that produced the supplied argument.

Suppose a function expects a value

$$x : A,$$

and suppose that a computation has produced a value

$$v : A$$

through the history

$$H_v.$$

Traditional substitution inserts only the value

$$v.$$

Replay instead introduces the pair

$$(v, H_v),$$

thereby preserving not only what the argument is, but how it came to exist.

Consequently, function application becomes the extension of an existing history rather than the replacement of a variable.

Formally, if

$$H_f$$

denotes the history producing the function and

$$H_v$$

denotes the history producing the argument, then function application extends both histories into a larger construction

$$H' = \text{Replay}(H_f, H_v).$$

The resulting computation therefore inherits the complete provenance of both participants.

This interpretation naturally explains why histories were introduced before Replay.

Replay is not an isolated primitive.

It is an operation on histories.

Without persistent histories there would be nothing to replay.

The distinction between substitution and replay becomes clearer when considering repeated evaluation.

Suppose a function computes

$$f(x) = x + x.$$

Traditional substitution transforms

$$f(3)$$

into

$$3 + 3.$$

From the viewpoint of symbolic evaluation this is entirely sufficient.

Spherepop observes, however, that the two occurrences of 3 are not merely identical values.

They are two references to the same historical construction.

Replay therefore duplicates the admissible continuation rather than erasing its origin.

The computational graph remains connected to the events that produced the argument.

This distinction becomes increasingly important once computation extends beyond deterministic arithmetic.

Suppose a value arises from probabilistic sampling, external input, or a distributed computation.

Two identical numerical values may possess entirely different historical origins.

Traditional substitution cannot distinguish between them because both reduce to the same symbol.

Replay preserves this distinction automatically.

The identity of a computational object therefore consists of both its value and its provenance.

Replay also provides a natural interpretation of incremental computation.

Suppose a large computation has already produced an extensive history

$$H.$$

A small modification to an earlier input should not require the entire computation to begin again.

Instead, Replay identifies precisely those historical continuations whose dependencies have changed and reconstructs only those portions of the computation.

The unaffected regions remain valid because their histories remain unchanged.

In this sense, Replay resembles the execution strategy employed by modern incremental build systems and reactive programming environments, yet it is expressed here as a primitive semantic operation rather than an implementation optimization.

Another consequence concerns explanation.

One of the persistent difficulties in contemporary software systems is the production of meaningful explanations for computed results.

Traditional execution traces often consist of low-level machine operations bearing little resemblance to the conceptual structure of the original program.

Replay naturally generates explanations at the level of computational history.

To explain why a value exists is simply to replay the sequence of admissible events that constructed it.

Explanation therefore becomes an intrinsic property of execution rather than an auxiliary debugging facility.

It is important to emphasize that Replay is not intended as a replacement for the efficiency of substitution in existing implementations.

An optimizing compiler may still perform ordinary substitutions whenever they preserve the required historical semantics.

Replay is instead the semantic primitive from which substitution may be derived as a history-erasing optimization under appropriate conditions.

This reverses the traditional perspective.

Rather than beginning with substitution and attempting to recover provenance afterwards, Spherepop begins with complete historical continuity and permits substitution only when the discarded history is known to be observationally irrelevant.

Replay therefore represents the first major consequence of adopting histories as the primary computational object.

Programs no longer evolve through isolated symbolic rewrites.

They evolve through the continual extension and reconstruction of admissible histories whose provenance remains available throughout the lifetime of the computation.

The next section introduces the complementary operation of *Refusal*. Where Replay governs the admissible continuation of histories, Refusal governs their admissible termination, preventing incoherent computational extensions before they become irreversible commitments.

11 Refusal and Admissibility

Replay governs the admissible continuation of computation. Every successful evaluation extends an existing history by replaying the computational events necessary to produce new constructions. The complementary question is equally important.

What should happen when no admissible continuation exists?

Traditional programming languages answer this question in numerous ways. Execution may halt with an exception, terminate with an error code, produce an undefined value, invoke recovery mechanisms, or simply exhibit undefined behavior. Although these mechanisms differ operationally, they share a common assumption: failure is primarily an event that occurs *after* an inadmissible computation has already been attempted.

Spherepop adopts a different perspective.

Instead of treating failure as the consequence of executing an invalid computation, the calculus introduces a primitive operation that prevents such computations from becoming historical commitments in the first place.

This primitive is called *Refusal*.

Refusal is not an exception.

It is not an error message.

It is not a distinguished return value.

Rather, Refusal is a computational boundary that prevents an inadmissible history from being extended.

Suppose a computation has produced the history

$$H,$$

and suppose that an operation proposes to extend this history by an event

$$e.$$

Traditional execution attempts the extension and determines afterwards whether the resulting state is acceptable.

Spherepop instead evaluates the admissibility of the extension itself.

If

$$\mathrm{Adm}(H, e)$$

holds, then the computation proceeds,

$$H \longrightarrow H \parallel e.$$

Otherwise,

$$H \longrightarrow \mathrm{Refuse}(r),$$

where r records the reason that the continuation was rejected.

The history itself remains coherent.

Nothing inadmissible has been admitted into it.

This distinction is subtle but fundamental.

Errors describe computations that have already occurred.

Refusals prevent certain computations from becoming part of the historical record at all.

From this viewpoint, admissibility precedes evaluation.

Evaluation is not merely the execution of available operations.

It is the execution of operations that satisfy the continuation conditions defined by the surrounding history.

The role traditionally played by types now acquires a new interpretation.

Rather than viewing a type as the collection of permissible values, Spherepop views a type as a structured collection of admissible continuations.

A typing judgment

$$\Gamma \vdash t : A$$

may therefore be read historically.

It asserts not merely that the term t possesses type A, but that the history represented by Γ may be extended by the construction t without violating the continuation discipline embodied by A.

Types become operational boundaries rather than descriptive labels.

This interpretation also clarifies why contexts are naturally understood as histories.

In conventional type theory,

$$\Gamma = x_1 : A_1,\, x_2 : A_2,\, \ldots,\, x_n : A_n$$

is often presented as an environment containing assumptions.

Within Spherepop, the same context is interpreted as the ordered record of admitted computational events that make the present continuation possible.

Each declaration represents a previous commitment.

Future computation inherits these commitments but cannot invalidate them.

The ordering therefore reflects historical construction rather than merely syntactic bookkeeping.

Refusal also provides a natural semantic foundation for program verification.

Suppose a resource may only be consumed once.

Instead of proving globally that every execution satisfies this constraint, each attempted continuation is locally tested for admissibility.

The first consumption extends the history normally.

Subsequent attempts encounter a Refusal because no admissible continuation exists.

Likewise, access-control policies, capability systems, protocol verification, and dependent resource management may all be interpreted as different forms of historical admissibility.

Rather than adding special-purpose runtime mechanisms for each domain, Spherepop treats them uniformly as boundary conditions governing historical

extension.

An important consequence follows immediately.

Refusal is fundamentally different from Collapse.

A Refusal preserves openness.

It declares that the proposed continuation cannot presently be admitted while leaving the surrounding computation historically coherent.

Collapse, by contrast, represents commitment.

Once a Sphere has been successfully popped and its value admitted into the history, that commitment becomes part of every future continuation.

The distinction resembles ordinary scientific investigation.

Rejecting an experimental hypothesis does not invalidate the history of the investigation.

It merely prevents one possible continuation from becoming accepted.

Publishing a confirmed result, however, extends the scientific record in a way that subsequent work inherits.

Spherepop generalizes this pattern into a computational principle.

Programs evolve not merely by producing values, but by selectively admitting or refusing historical continuations.

Computation therefore consists of alternating phases of openness and commitment.

Replay proposes new continuations.

Refusal filters them according to admissibility.

Pop resolves bounded computational regions.

Collapse records the resulting commitments.

Together these four primitives describe the evolution of computational history without requiring mutation, global state replacement, or irreversible syntactic rewrites as primitive notions.

Having established how histories may be extended or refused, we next examine how multiple histories evolve simultaneously. This leads naturally to the introduction of concurrent composition through the *Merge* operator, where independent computational regions may progress in parallel while preserving coherent historical structure.

12 Merge and Concurrent Composition

The historical interpretation of computation developed thus far naturally raises another question. If computation is the evolution of histories rather than merely the reduction of expressions, how should independent computations interact?

Traditional programming languages typically introduce concurrency as an additional execution model layered upon an otherwise sequential semantics. Threads, processes, actors, futures, channels, and asynchronous procedures are often presented as independent mechanisms requiring specialized runtime support. Sequential computation is regarded as fundamental, while concurrency appears as an extension.

Spherepop adopts the opposite viewpoint.

Independence is fundamental.

Sequential execution is merely one particular ordering among histories whose dependencies require serialization.

The primitive responsible for expressing independence is the *Merge* operator.

Rather than denoting temporal interleaving, Merge denotes the coexistence of independent computational regions whose histories may evolve simultaneously.

If

$$H_1$$

and

$$H_2$$

are histories whose continuations possess no unresolved dependencies upon one another, then they may be combined as

$$H_1 \otimes H_2.$$

Here the tensor symbol emphasizes that Merge is not ordinary sequencing.

Neither history is considered primary.

Both evolve concurrently until subsequent computation requires their joint results.

This distinction becomes apparent even in elementary arithmetic.

Consider

$$((2+3)\times(4+5)).$$

Textually, the additions appear one after another.

Operationally, however, nothing requires them to execute sequentially.

The left Sphere

$$2+3$$

shares no dependencies with the right Sphere

$$4+5.$$

Both become admissible immediately after the enclosing multiplication has been constructed.

Consequently,

$$\text{Merge}(2+3,\ 4+5)$$

captures the true computational structure more faithfully than any particular sequential evaluation order.

Only after both histories have completed does the enclosing multiplication become admissible for Pop.

This observation generalizes well beyond arithmetic.

Independent function calls,

parallel database queries,

concurrent simulations,

distributed computations,

and asynchronous network requests

all share the same mathematical structure.

Each consists of computational regions that may evolve independently until their results are eventually combined.

Merge therefore expresses logical independence rather than processor parallelism.

Whether two merged histories execute on separate cores, different machines, or simply alternate on a single processor is an implementation decision rather than a semantic one.

The semantics require only that the histories remain independent until an operation explicitly introduces a dependency between them.

The Merge operator possesses several important algebraic properties.

Associativity follows immediately from the absence of privileged ordering,

$$(H_1 \otimes H_2) \otimes H_3 \cong H_1 \otimes (H_2 \otimes H_3).$$

Likewise, if the histories are genuinely independent, commutativity holds,

$$H_1 \otimes H_2 \cong H_2 \otimes H_1.$$

These identities imply that large Merge expressions need not be interpreted as binary trees.

Instead, they may be flattened into unordered collections of independent computational regions,

$$H_1 \otimes H_2 \otimes \cdots \otimes H_n,$$

whose evaluation order is determined entirely by dependency analysis rather than syntactic nesting.

This flattening has significant practical consequences.

Many runtime systems spend considerable effort discovering opportunities for parallel execution after programs have already been written.

Spherepop exposes these opportunities directly through the structure of the history.

Concurrency is therefore not inferred.

It is represented explicitly.

The runtime scheduler acquires a correspondingly simple role.

At every stage of execution it maintains a collection of currently admissible Spheres.

Whenever multiple independent Spheres are simultaneously ready, they may all be popped without affecting the correctness of the resulting history.

The scheduler therefore performs three elementary operations.

First, it identifies all admissible computational regions.

Second, it partitions these regions according to dependency.

Finally, it evaluates every independent partition concurrently.

Sequential execution appears only when dependency relations reduce a partition to a single admissible Sphere.

Consequently, sequentiality is no longer assumed.

It emerges naturally from the dependency structure of the computation itself.

The historical interpretation also resolves a long-standing ambiguity in concurrent execution.

Traditional execution traces often record the accidental interleaving produced by a particular scheduler.

Spherepop instead records only the partial ordering imposed by computational dependencies.

Two independent Pops possess no intrinsic temporal ordering.

Any scheduler respecting the dependency graph constructs histories that are equivalent up to permutation of independent events.

The semantics therefore distinguish essential causality from accidental execution order.

This distinction greatly simplifies reasoning about distributed computation.

Different processors may execute independent histories in different orders while still producing equivalent historical structures.

Only dependency relations contribute to the semantic identity of the computation.

Merge also prepares the introduction of uncertainty.

Once multiple independent computational regions may evolve simultaneously, it becomes natural to ask what happens when several admissible continuations are possible but only one should ultimately be realized.

This situation is not one of concurrency but of controlled uncertainty.

The corresponding primitive is the *Choice* operator, which introduces probabilistic and nondeterministic continuation while preserving the same historical semantics developed throughout the preceding sections.

13 Choice and Probabilistic Evaluation

The Merge operator describes the evolution of independent computational histories. Every merged Sphere may proceed concurrently because no unresolved dependencies constrain their execution. Concurrency, however, is only one form of computational branching. Another arises whenever the computation itself contains uncertainty.

Traditional programming languages typically represent uncertainty by treating it as an external concern. Random number generators, probabilistic libraries, Monte Carlo methods, Bayesian inference engines, and reinforcement learning

algorithms are introduced as specialized software layered upon an otherwise deterministic computational core.

Spherepop adopts a different philosophy.

Uncertainty is not an external feature of computation.

It is another form of admissible continuation.

Just as Merge represents the simultaneous evolution of independent histories, the *Choice* operator represents the coexistence of multiple admissible future continuations before commitment.

Suppose a computation has reached the history

$$H.$$

Rather than admitting a single continuation, the computation may expose several possible futures,

$$H \rightsquigarrow \{H_1, H_2, \ldots, H_n\}.$$

These histories are not executed simultaneously in the same sense as Merge. Instead, they represent alternative continuations awaiting eventual commitment.

The simplest binary Choice may be written

$$\text{Choice}(p, A, B),$$

where

$$0 \leq p \leq 1.$$

Operationally, this exposes two admissible continuations.

The first continues toward

$$A$$

with probability

$$p,$$

while the second continues toward

$$B$$

with probability

$$1 - p.$$

Unlike conventional conditional execution, neither branch is considered fundamental.

Both are legitimate continuations of the current history until commitment occurs.

This interpretation deliberately separates uncertainty from execution.

The computational graph itself remains unchanged.

Only the interpretation of the Choice operator determines how continuation is selected.

Consequently, the same computational structure may support multiple semantic regimes.

In deterministic computation,

$$\text{Choice}(1, A, B)$$

always continues toward

$$A.$$

In probabilistic computation,

the branch is selected according to a probability distribution.

In symbolic reasoning,

both continuations may remain available simultaneously.

In planning systems,

Choice represents alternative futures that are evaluated before commitment.

The surrounding computational graph therefore does not need to know what kind of logic it is executing.

Only the local operator changes.

This observation has important architectural consequences.

Most programming languages distinguish Boolean computation, probabilistic reasoning, fuzzy inference, and differentiable optimization by constructing entirely different execution engines.

Spherepop instead separates the composition graph from the operator algebra.

The graph merely records how computational regions depend upon one another.

The operators determine how information propagates through that graph.

Changing the operators changes the computational interpretation without changing the underlying history.

Boolean logic therefore becomes one particular operator library.

Fuzzy logic becomes another.

Probabilistic inference becomes another.

Differentiable computation becomes yet another.

The computational substrate remains identical.

This separation between composition and interpretation significantly simplifies language design.

Rather than constructing specialized runtimes for every reasoning paradigm, the Universal Operator Machine introduced later executes a single historical composition graph while allowing the operator library to vary.

Choice also admits two complementary semantic interpretations.

In the first, the operator returns an immediate value,

$$a : A,$$

chosen according to the specified probabilities.

This corresponds to operational sampling and is appropriate whenever execution requires a concrete result.

In the second interpretation, the operator returns an explicit probability distribution,

$$d : \mathrm{Dist}(A),$$

without committing to any particular sample.

This monadic interpretation preserves uncertainty as a first-class computational object.

Subsequent operators may therefore transform entire distributions rather than individual values.

Both interpretations share the same historical semantics.

The difference lies solely in when commitment occurs.

Immediate sampling performs an early Collapse.

Distribution-valued Choice delays Collapse until later stages of computation.

This distinction reflects a recurring principle throughout the Spherepop Calculus.

Wherever possible, commitment should occur only after sufficient information has become available.

Early commitment reduces future flexibility.

Delayed commitment preserves admissible continuations.

Choice therefore complements the Pop primitive.

Pop dissolves computational boundaries.

Choice delays which boundary will ultimately be dissolved.

Together they provide two orthogonal mechanisms governing computational evolution.

Pop determines *when* a computational region becomes ready for commitment.

Choice determines *which* admissible continuation will eventually be committed.

This perspective also clarifies the relationship between probabilistic computation and machine learning.

Backpropagation, Bayesian inference, reinforcement learning, Monte Carlo simulation, stochastic optimization, and probabilistic programming all become different ways of assigning semantics to Choice operators distributed throughout a common historical composition graph.

The execution engine itself need not distinguish among these disciplines.

It merely maintains histories, evaluates admissible Spheres, propagates dependencies, and records commitments.

The interpretation of uncertainty belongs to the operator algebra rather than to the execution model.

Having established the four primitive computational operations—Sphere, Pop, Merge, and Choice—we now possess a sufficiently expressive operational calculus to reconsider one of the oldest structures in logic itself: the context. Rather than treating contexts as static collections of assumptions, the next section reconstructs them as evolving histories of admissible computational construction.

14 Contexts as Histories

The notion of a context occupies a central position in modern logic and type theory. Every typing judgment, proof construction, and program derivation is performed relative to a context

$$\Gamma,$$

which records the assumptions currently available during construction. Conventionally, a context is presented as an ordered sequence of declarations,

$$\Gamma = x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n.$$

Although the ordering of declarations is important operationally, the context is generally interpreted as a static environment—a collection of assumptions against which terms are checked. Structural rules such as weakening, contraction, and exchange describe how this environment may be manipulated without changing the meaning of derivations.

Spherepop proposes a different interpretation.

A context is not fundamentally an environment.

It is a history.

Each declaration records an event that has already occurred in the evolution of the computation. The meaning of a declaration is therefore inseparable from its position within that history.

Rather than viewing

$$x : A$$

as a passive assumption, Spherepop interprets it as the successful admission of a computational event into the historical record. The context therefore grows by historical extension rather than by arbitrary insertion.

The empty context,

$$\emptyset,$$

represents the unique history containing no admitted computational events.

Every subsequent context is obtained by extending a previous history,

$$\Gamma \longrightarrow \Gamma, x : A,$$

provided that the proposed extension satisfies the admissibility conditions of the calculus.

This seemingly modest reinterpretation has profound consequences.

Traditional contexts describe what is currently available.

Historical contexts describe how the current situation came to exist.

The distinction becomes apparent when considering dependency.

Suppose the declarations

$$x : A$$

and

$$y : B(x)$$

appear in a context.

Within conventional presentations this simply indicates that the type of y depends upon the value of x.

Within Spherepop it also records a temporal relationship.

The event introducing x necessarily precedes the event introducing y.

The dependency is therefore simultaneously logical and historical.

Every declaration inherits the complete provenance of the declarations upon which it depends.

Consequently, a context is naturally viewed as a directed acyclic graph of construction rather than merely a linear sequence of assumptions.

The linear notation survives only because every directed acyclic graph admits a topological ordering compatible with its dependency relations.

The graph, however, is the more fundamental object.

Sequential notation is merely one readable representation.

This interpretation also changes the role of structural rules.

Weakening no longer means that arbitrary assumptions may be inserted into an environment.

Instead, weakening represents the monotonic extension of an existing history by an event whose future influence may ultimately prove irrelevant.

The computation grows, even if the newly admitted event is never referenced again.

Contraction likewise acquires a historical interpretation.

Rather than duplicating assumptions syntactically, contraction recognizes that multiple computational continuations may legitimately replay the same historical construction.

Nothing in the history itself is duplicated.

Only multiple references to the same historical event are created.

Exchange similarly changes character.

Traditional type theory often allows adjacent assumptions to be reordered provided that dependency constraints are respected.

Spherepop interprets this as the existence of multiple equivalent historical presentations of the same dependency graph.

Independent events possess no intrinsic temporal ordering.

Only dependency determines historical precedence.

Thus two declarations

$$x : A$$

and

$$y : B$$

that are entirely independent may appear in either order without altering the underlying history.

The apparent linear sequence is therefore understood as one of many valid topological orderings of the same historical construction.

Viewing contexts as histories also provides a natural foundation for incremental verification.

Suppose a computation modifies an early declaration.

Traditional systems frequently require large portions of the derivation to be rechecked because the environment itself has changed.

Within Spherepop, the dependency graph identifies precisely which subsequent historical events depend upon the modified construction.

Only those continuations require replay.

All independent regions of the history remain valid.

This interpretation aligns naturally with modern incremental compilation, proof replay, and reactive programming while expressing them as consequences of the historical semantics rather than as implementation techniques.

Perhaps most importantly, historical contexts eliminate the artificial distinction between execution and verification.

The same object serves simultaneously as an execution trace, a dependency graph, a typing environment, and a provenance record.

Evaluation extends the history.

Typing constrains admissible extensions.

Replay reconstructs dependent continuations.

Refusal blocks inadmissible ones.

These are no longer separate mechanisms operating on different data structures. They become different perspectives on the same evolving historical object.

The reinterpretation of contexts therefore prepares the next conceptual step. If contexts themselves are histories of admitted computational events, then types cannot merely classify completed values. They must regulate which historical continuations are permitted to exist.

The following section develops this idea by reinterpreting type theory itself as a calculus of admissible continuation, where types function not primarily as classifications, but as structured boundaries governing the evolution of computational history.

15 Types as Refusal Structures

Having reinterpreted contexts as histories, we may now reconsider the role of types themselves. In conventional programming languages and logical systems, types are generally understood as classifications. A value either belongs to a type or it does not. Type checking therefore answers a descriptive question: "What kind of object is this?"

While enormously successful, this interpretation places types after construction. The computation first produces an object, and the type system then verifies that the object possesses the desired properties.

Spherepop adopts the opposite perspective.

A type is not primarily a classification of completed objects.

It is a discipline governing admissible continuation.

Rather than asking

"What type does this value have?"

the calculus asks

"May this history continue?"

The distinction is subtle but fundamental.

Types therefore become operational boundaries rather than descriptive labels.

Every computational history approaches a sequence of decision points. At

each point the type determines whether the proposed continuation remains coherent with everything that has already been admitted into the history.

Consequently, typing becomes a form of structured refusal.

A judgment

$$\Gamma \vdash t : A$$

should therefore be read operationally.

It does not merely assert that the term t belongs to the collection of objects described by A.

Instead it states that extending the history represented by Γ by the construction t remains admissible under the continuation discipline represented by A.

The emphasis shifts from membership to permission.

This interpretation naturally aligns with the historical semantics developed throughout the preceding chapters.

Replay determines how histories continue.

Refusal determines which continuations are blocked.

Types specify the admissibility conditions that distinguish one from the other.

Every type therefore acts as a computational boundary.

One may visualize a type as surrounding a computational region.

A proposed continuation approaches the boundary.

If the continuation satisfies the admissibility conditions, it passes through and extends the history.

Otherwise the boundary refuses the continuation before commitment occurs.

Unlike conventional runtime errors, this refusal does not represent a failed state.

It represents the successful preservation of historical coherence.

Nothing incoherent has entered the computational record.

The history remains internally consistent.

This viewpoint also clarifies the relationship between static and dynamic checking.

Static type systems attempt to establish admissibility before execution.

Dynamic systems postpone certain admissibility decisions until runtime.

Spherepop treats these not as fundamentally different mechanisms but as different points along the same historical process.

In both cases the essential operation is identical.

A continuation is proposed.

The surrounding refusal structure evaluates its admissibility.

The history either extends or remains unchanged.

Consequently, compile-time verification and runtime verification become different scheduling strategies for the same semantic operation.

This interpretation extends naturally beyond ordinary type checking.

Ownership systems refuse histories in which multiple incompatible mutations occur simultaneously.

Capability systems refuse histories lacking the required authority.

Session types refuse communication histories violating agreed protocols.

Linear logic refuses histories that duplicate unique resources.

Dependent type systems refuse histories whose logical evidence is insufficient.

Rather than requiring separate theoretical foundations for each discipline, Spherepop views them as particular instances of historical admissibility.

The specific refusal conditions differ.

The underlying mechanism does not.

This uniformity becomes increasingly valuable as software systems grow in complexity.

Modern compilers routinely combine lexical scope, ownership analysis, effect systems, borrow checking, protocol verification, resource accounting, and proof obligations.

Each introduces its own collection of rules.

Spherepop instead treats each as defining a different family of refusal boundaries operating upon the same historical object.

The complexity therefore shifts away from the execution engine and into the admissibility conditions themselves.

The execution model remains remarkably simple.

Histories evolve.

Replay proposes continuations.

Types determine admissibility.

Refusal blocks incoherent extensions.

Pop commits admissible computational regions.

This separation between continuation and commitment also illuminates an important philosophical distinction.

A computation is not correct because it eventually reaches a desirable state.

Rather, correctness is preserved because every intermediate continuation has been admitted through coherent historical extension.

Global correctness emerges from local admissibility.

This mirrors the structure of mathematical proof.

One does not establish the validity of a theorem solely by inspecting the final line.

Each inference step must itself be admissible.

The theorem is trustworthy because every extension of the proof history satisfied the governing rules of inference.

Programs may therefore be viewed in precisely the same manner.

Execution is the gradual construction of a computational proof whose admissibility is maintained continuously rather than verified only after the fact.

This reinterpretation prepares the transition to dependent type theory.

If types regulate historical continuation, then it becomes natural for the admissibility of future continuations to depend explicitly upon earlier historical events.

Dependent types therefore emerge not as an independent extension of type theory, but as the natural consequence of allowing refusal structures themselves to evolve along with the histories they govern.

16 Dependent Types as Historical Construction

The reinterpretation of types as refusal structures naturally transforms our understanding of dependent type theory. In conventional presentations, dependent types generalize ordinary types by allowing later types to depend upon earlier values. While this dependence is mathematically elegant, it is usually expressed syntactically as substitution into families of types.

Spherepop proposes a different interpretation.

Dependence is not fundamentally substitution.

Dependence is historical continuation.

Every newly admitted computational event extends the history available to future constructions. Consequently, the admissibility of later computations may depend not merely upon previously computed values, but upon the complete historical process that produced those values.

This distinction becomes apparent even in elementary examples.

Suppose a computation constructs a vector

$$v : \text{Vector}(A, n).$$

Traditional dependent type theory records only the resulting natural number n. Future computations depend upon that numerical value through substitution.

Spherepop instead records the historical event that established the vector together with its length.

Future computations therefore inherit both the value and its provenance.

The dependence is historical before it is symbolic.

More generally, let

$$H$$

denote the current computational history.

A dependent type family may be viewed as a mapping

$$B(H),$$

whose admissibility varies according to the history that has already been constructed.

As computation proceeds,

$$H \longrightarrow H',$$

the family itself evolves,

$$B(H) \longrightarrow B(H').$$

Dependent computation therefore becomes the evolution of admissibility rather than merely the substitution of variables.

This historical interpretation provides an intuitive understanding of the dependent function, or Π-type.

Conventionally,

$$\Pi_{x:A} B(x)$$

denotes the family of functions that transform every element of A into a corresponding element of $B(x)$.

Within Spherepop, a dependent function is more naturally understood as a universal continuation policy.

Given an admissible extension of the current history by some construction of type A, the function determines how the history may continue while remaining admissible.

A dependent function therefore specifies not simply an output value, but an entire continuation strategy.

Function application becomes an interaction between Replay and historical admissibility.

The argument contributes a new historical construction.

Replay integrates this construction into the existing history.

The dependent function then determines the admissible continuation relative to that enlarged history.

No separate notion of substitution is required as the primary semantic operation.

The continuation follows directly from the historical evolution of the computation.

The same interpretation extends naturally to dependent sums.

A conventional dependent pair

$$\Sigma_{x:A} B(x)$$

consists of a witness together with dependent evidence.

Historically, the interpretation becomes even clearer.

The first component represents a committed computational event.

The second component records a construction whose admissibility depends upon that earlier commitment.

The pair therefore records a miniature computational history.

First, an admissible witness is constructed.

Second, the history is extended by that witness.

Finally, dependent evidence is admitted relative to the enlarged history.

Dependent pairs thus become explicit records of staged construction.

Rather than representing static data, they encode the chronological order in which knowledge became available.

This interpretation also clarifies why dependent types have proved so useful for software verification.

Many correctness conditions are inherently historical.

A file must be opened before it is read.

A lock must be acquired before shared memory is modified.

A network session must complete its handshake before encrypted communication may begin.

These constraints are not merely properties of isolated values.

They are properties of histories.

Dependent types express such constraints precisely because they describe how later constructions depend upon earlier ones.

Spherepop makes this historical character explicit.

Instead of viewing dependence as an advanced extension of ordinary typing, it treats dependence as the natural consequence of computation itself.

Every successful Pop enlarges the available history.

Every enlargement changes the collection of admissible future continuations.

Dependent typing therefore emerges automatically once histories replace static contexts.

This perspective also simplifies the conceptual relationship between dependent types and Replay.

Replay already reconstructs the historical context required for a continuation.

Dependent types simply specify which continuations remain admissible after that historical reconstruction.

The two mechanisms therefore complement one another naturally.

Replay reconstructs the past.

Dependent types govern the future.

Together they describe the continuous evolution of computation as an expanding historical object whose admissibility is maintained at every stage.

The remaining ingredient required for a complete historical reconstruction of type theory concerns equality itself. If histories rather than isolated values constitute the primitive computational objects, then equality can no longer be understood merely as syntactic identity or extensional equivalence. Instead, equality becomes a question of whether distinct historical constructions admit the same future continuations. This leads naturally to the reinterpretation of identity types as mechanisms of historical repair.

17 Identity Types as Historical Repair

Identity occupies a unique position within dependent type theory. In conventional presentations, the identity type

$$\mathrm{Id}_A(a, b)$$

represents evidence that two terms a and b should be regarded as equal within the type A. The eliminator J expresses the fundamental principle that every proof of equality may ultimately be reduced to the reflexive case. This formulation has proved extraordinarily fruitful, forming the basis of modern proof assistants and homotopical interpretations of type theory.

From the historical perspective developed throughout this work, however, identity is naturally interpreted in a different way.

Two computational objects are rarely created in identical ways.

Even when they possess the same observable value, they may arise from completely different computational histories. One value may have been constructed through symbolic evaluation, another through probabilistic sampling, and a third through distributed computation. Conventional semantics collapses these distinctions once the resulting values become equal.

Spherepop instead asks a different question.

When should two distinct histories be regarded as interchangeable?

Identity therefore becomes a property of histories rather than merely a property of values.

Suppose two independent computational histories,

$$H_a$$

and

$$H_b,$$

produce values

$$a, b : A.$$

Traditional identity asks whether

$$a = b.$$

Spherepop asks whether the histories

$$H_a$$

and

$$H_b$$

admit precisely the same future continuations.

If every admissible continuation beginning with H_a is equally admissible beginning with H_b, then no computational distinction remains between the two histories. They may therefore be regarded as historically equivalent.

Identity thus measures not observational equality alone but historical substitutability.

This viewpoint naturally introduces the notion of repair.

Suppose two histories differ.

Rather than immediately declaring them unequal, Spherepop seeks a sequence of admissible transformations capable of converting one history into the other.

Such a transformation is called a *repair*.

Formally, one may regard a repair as a history-preserving transformation

$$R : H_a \Longrightarrow H_b,$$

whose execution preserves the admissibility of every subsequent continuation.

The identity type therefore records the existence of an admissible repair between two computational constructions.

Reflexivity corresponds to the trivial repair.

Every history is immediately repairable to itself,

$$\mathrm{refl}_a : \mathrm{Id}_A(a, a),$$

because no transformation is required.

Symmetry states that every admissible repair may be reversed whenever an inverse repair exists.

Transitivity expresses the composition of repairs.

If

$$R_1 : H_a \Longrightarrow H_b$$

and

$$R_2 : H_b \Longrightarrow H_c,$$

then their composition yields

$$R_2 \circ R_1 : H_a \Longrightarrow H_c.$$

Identity therefore inherits the algebraic structure of historical transformations rather than merely symbolic equalities.

The eliminator J also acquires an intuitive historical interpretation.

Rather than asserting that every equality proof ultimately reduces to reflexivity, Spherepop interprets J as the principle that whenever all historically significant distinctions have been repaired, the remaining difference is computationally trivial.

One may therefore replace the repaired history by its canonical representative without altering any admissible continuation.

Operationally, J authorizes historical normalization.

Complex repair paths need not remain explicitly represented forever.

Once every distinction along the repair path has become operationally irrelevant, the entire repair may collapse to the reflexive history.

This historical interpretation also explains why identity becomes increasingly important in long-running computational systems.

Distributed software frequently produces multiple histories leading to equivalent states.

Version-control systems merge independently developed branches.

Incremental compilers reconcile previous builds with modified source code.

Proof assistants reconstruct large derivations after local changes.

Database replication synchronizes independently evolving histories.

In each case the central question is not simply whether the resulting values agree.

The deeper question is whether the distinct computational histories may now be treated as interchangeable.

Spherepop places this question at the heart of identity itself.

Identity is therefore no longer viewed as static equality between completed objects.

It becomes the study of admissible reconciliation between histories.

Values remain important, but they are interpreted as observable endpoints of larger computational constructions.

The true object of comparison is the historical process that produced them.

This reinterpretation naturally unifies equality with Replay and Refusal.

Replay reconstructs historical continuations.

Refusal blocks inadmissible continuations.

Repair reconciles divergent continuations.

Identity records when reconciliation has become complete.

Together these operations describe the continuous evolution, comparison, and integration of computational histories without reducing computation to syntactic replacement alone.

Having reconstructed contexts, types, dependent computation, and identity in historical terms, the remaining task is to formalize the operational semantics of the Spherepop Calculus itself. We now develop the reduction system governing Sphere, Pop, Merge, Choice, Replay, and Refusal, showing that ordinary evaluation arises as a special case of the broader evolution of computational histories.

18 Operational Semantics

The preceding chapters have progressively replaced the conventional primitives of symbolic computation with historical ones. Parentheses became Spheres. Reduction became Pop. Substitution became Replay. Type checking became historical admissibility, while identity became repair between computational histories. We are now in a position to formalize the operational semantics of the Spherepop Calculus itself.

Unlike traditional operational semantics, whose primary object is a term, Spherepop takes the computational history as its semantic state. Evaluation is therefore not merely a sequence of syntactic rewrites but the progressive construction of an irreversible historical object.

The operational configuration of the machine is written

$$(H, \mathcal{S}),$$

where

- H is the current computational history,
- $\mathcal{S}$ is the collection of unresolved Spheres.

Unlike conventional abstract machines, no mutable global state is required as a primitive semantic object. Every modification of computation is represented by

extending the history and updating the collection of unresolved computational regions.

At any instant, the runtime repeatedly performs the same evaluation cycle.

$$\text{Locate} \longrightarrow \text{Verify} \longrightarrow \text{Pop} \longrightarrow \text{Replay} \longrightarrow \text{Repeat}.$$

Although remarkably simple, this cycle subsumes ordinary sequential evaluation, concurrent scheduling, incremental execution, and probabilistic computation.

18.1 Ready Spheres

Not every Sphere may be evaluated immediately.

A Sphere becomes *Ready* only when every computational dependency inside its boundary has already completed.

Formally,

$$\text{Ready}(S)$$

holds whenever

$$\forall d \in \text{Deps}(S), \qquad \text{Resolved}(d).$$

The Ready predicate replaces recursive descent through syntax trees.

Instead of searching for reducible expressions, the runtime searches for computational regions whose local histories have become complete.

This distinction is important.

Traditional reduction discovers redexes.

Spherepop discovers admissible computational regions.

The object of search has changed from syntax to history.

18.2 Opening Spheres

Whenever the parser or compiler encounters a new computational scope, the runtime introduces a new Sphere into the collection of unresolved regions.

Operationally,

$$(H, \mathcal{S}) \longrightarrow (H \parallel \text{Open}(S), \mathcal{S} \cup \{S\}).$$

Opening a Sphere does not immediately perform computation.

Instead, it records the existence of a bounded computational environment whose future evolution will eventually become admissible.

The opening event therefore contributes directly to the historical record.

18.3 Pop Reduction

Once a Sphere becomes Ready, the Pop rule may be applied.

Suppose

$$S = (B, I)$$

contains an interior computation

$$I \Longrightarrow v.$$

Then

$$(H, \mathcal{S} \cup \{S\}) \longrightarrow (H \parallel \text{Pop}(S), (\mathcal{S} - \{S\})[v]).$$

Here

$$[v]$$

denotes replacement of the Sphere by its resulting value throughout the remaining unresolved computational regions.

Unlike ordinary substitution, however, the replacement carries its historical provenance.

The runtime therefore performs

$$(v, H_v)$$

rather than merely

$$v.$$

Consequently, every subsequent continuation retains access to the computation that produced the value.

18.4 Replay

Whenever one computational region consumes the output of another, Replay extends the existing history.

If

$$H_f$$

constructed a function,
and

$$H_a$$

constructed an argument,
then

$$\text{Replay}(H_f, H_a)$$

constructs the historical environment required for the next continuation.
Operationally,

$$(H, \mathcal{S}) \longrightarrow (H \parallel \text{Replay}, \mathcal{S}').$$

Replay therefore never mutates history.
It extends it.
This monotonicity is one of the defining invariants of the calculus.

18.5 Refusal

Suppose a proposed continuation

$$e$$

fails the admissibility relation.
Rather than extending the history with an incoherent event, the runtime records

$$\text{Refuse}(r),$$

where

$$r$$

identifies the violated continuation condition.

Formally,

$$\neg\,\mathrm{Adm}(H, e) \Longrightarrow (H, \mathcal{S}) \longrightarrow (H \parallel \mathrm{Refuse}(r), \mathcal{S}).$$

Notice that the unresolved computational regions remain unchanged.

The inadmissible continuation simply never becomes part of the history.

Refusal therefore preserves historical coherence without requiring rollback or exception propagation as primitive semantic mechanisms.

18.6 Merge Scheduling

Suppose

$$S_1, \ldots, S_n$$

are mutually independent Ready Spheres.

Rather than selecting one arbitrarily, the runtime may evaluate them simultaneously.

Operationally,

$$(H, \mathcal{S}) \longrightarrow (H \parallel \mathrm{Merge}(S_1, \ldots, S_n), \mathcal{S}').$$

Because Merge is associative and commutative over independent histories, the scheduler possesses considerable implementation freedom.

Any execution respecting dependency constraints produces historically equivalent computations.

Consequently, processor scheduling is separated from semantic correctness.

The runtime need only preserve causality.

18.7 Choice Reduction

Choice introduces multiple admissible continuations.

Operationally,

$$\mathrm{Choice}(p, t, u)$$

reduces according to one of two semantic policies.

The first immediately samples a continuation,

$$\text{Choice}(p, t, u) \longrightarrow t \quad \text{with probability } p,$$

or

$$\text{Choice}(p, t, u) \longrightarrow u \quad \text{with probability } 1 - p.$$

The second delays commitment,
producing an explicit distribution

$$d : \text{Dist}(A),$$

which continues propagating through the computational history without performing immediate Collapse.

Both interpretations share identical historical semantics.

They differ only in the scheduling of commitment.

18.8 The Operational Invariant

The entire operational semantics is governed by a remarkably simple invariant.

At every stage of execution,

1. every admitted event belongs to exactly one history,
2. every history grows monotonically,
3. every Pop removes one admissible computational boundary,
4. every Replay extends historical continuity,
5. every Refusal preserves historical coherence,
6. every Merge preserves causal independence,
7. every Choice delays or resolves commitment without altering the underlying computational graph.

Traditional operational semantics often appears as a collection of independent reduction rules.

Spherepop instead reveals them as manifestations of a single historical principle.

Computation is the progressive evolution of admissible computational regions.

Every operational rule either creates a region, extends its history, resolves its boundary, or determines which continuations remain admissible.

The next chapter complements this operational description with a denotational semantics, demonstrating that the historical execution model admits a precise mathematical interpretation independent of any particular runtime implementation.

19 Denotational Semantics

Operational semantics describes how a computation proceeds. It specifies which Sphere becomes admissible, when a Pop may occur, how histories are extended, and how Replay, Refusal, Merge, and Choice govern execution. While this procedural description is sufficient for implementing an interpreter, it does not answer a deeper mathematical question.

What is the computation?

Traditional denotational semantics answers this by assigning every program an abstract mathematical object independent of its execution strategy. Two programs are considered equivalent whenever they denote the same mathematical entity, regardless of the sequence of reduction steps used to evaluate them.

Spherepop preserves this objective while changing the mathematical object that is denoted.

The denotation of a program is not simply its resulting value.

Nor is it merely a function between domains.

Instead, the denotation is the history-preserving transformation generated by the evolution of computational regions.

Values become observable projections of a richer semantic object.

19.1 Programs as Morphisms Between Histories

Let

$$\mathcal{H}$$

denote the category whose objects are admissible computational histories and whose morphisms are history-preserving continuations.

Every Spherepop program therefore denotes a morphism

$$P : H \longrightarrow H',$$

where

$$H' = P(H).$$

Unlike ordinary state-transition systems, the morphism does not overwrite the existing history.

It extends it.

Consequently,

$$H \subseteq H'.$$

This monotonicity reflects the operational semantics developed previously.

No successful computation removes historical events.

Every admissible computation enlarges the existing construction.

19.2 The Denotation of a Sphere

A Sphere represents a bounded computational region.

Semantically, it denotes a localized transformation

$$S : H \longrightarrow H_S,$$

where

$$H_S$$

contains the complete historical evolution occurring inside the Sphere.

The enclosing computation cannot directly observe the interior evolution.

It observes only the completed transformation once the Sphere has been popped.

Thus the Sphere serves as both an operational boundary and a denotational abstraction.

Internal computation remains encapsulated until commitment.

19.3 The Denotation of Pop

Operationally, Pop dissolves a computational boundary.

Denotationally, Pop is the natural transformation that exposes the semantic content of a completed Sphere to the surrounding history.

If

$$S = H_S,$$

then

$$\text{Pop} : H_S \longrightarrow H_S \cup \{v\},$$

where

$$v$$

is the value produced by the completed computation.

The Pop therefore does not merely return a value.

It enlarges the observable history.

19.4 Replay as Functorial Extension

Replay transports historical constructions into new computational contexts.

Rather than substituting symbols, Replay preserves structural relationships.

Semantically,

$$\text{Replay} : \mathcal{H} \longrightarrow \mathcal{H}$$

acts functorially.

Identity histories remain unchanged,

$$\text{Replay}(\text{id}) = \text{id},$$

while composition is preserved,

$$\text{Replay}(f \circ g) = \text{Replay}(f) \circ \text{Replay}(g).$$

Replay therefore preserves the compositional structure of computation.

This is one of the principal reasons histories provide a more expressive semantic foundation than ordinary substitution.

19.5 Merge as Monoidal Composition

Independent computations should combine without imposing arbitrary sequential order.

Denotationally, Merge equips the category of histories with a symmetric monoidal structure,

$$(H_1, H_2) \mapsto H_1 \otimes H_2.$$

The tensor expresses logical independence rather than physical concurrency.

Associativity,

$$(H_1 \otimes H_2) \otimes H_3 \cong H_1 \otimes (H_2 \otimes H_3),$$

and symmetry,

$$H_1 \otimes H_2 \cong H_2 \otimes H_1,$$

follow immediately from the independence of the corresponding historical constructions.

Consequently, denotational equivalence depends only upon dependency relationships rather than execution order.

19.6 Choice and Probabilistic Semantics

Choice introduces uncertainty into historical evolution.

Operationally this may correspond either to immediate sampling or delayed distribution propagation.

Denotationally, these two interpretations correspond to different semantic levels of the same construction.

Immediate sampling denotes a particular historical continuation,

while delayed commitment denotes a probability measure over admissible histories,

$$\mu : \mathcal{H} \longrightarrow \mathbf{Prob}(\mathcal{H}).$$

The operational semantics therefore appears as one realization of a more general probabilistic denotation.

Changing the interpretation of Choice changes the semantic algebra while leaving the computational history unchanged.

19.7 Refusal as Partial Continuation

Refusal differs fundamentally from failure.

Failure attempts an inadmissible computation and subsequently reports its breakdown.

Refusal never constructs the inadmissible continuation.

Denotationally,

$$\text{Refuse} : H \rightharpoonup H'$$

is naturally interpreted as a partial morphism.

The continuation simply does not exist outside its admissible domain.

The history therefore remains coherent without introducing exceptional semantic objects.

Refusal is absence of continuation rather than production of an error value.

19.8 Compositionality

Perhaps the most important consequence of the denotational semantics is its compositionality.

Every semantic object introduced by the calculus is itself constructed from smaller semantic objects.

The denotation of a large program depends only upon the denotations of its constituent Spheres together with the operations Pop, Replay, Merge, Choice, and Refusal connecting them.

This compositional structure mirrors the operational semantics exactly.

Operational evaluation builds histories.

Denotational semantics interprets those histories.

Neither level depends upon accidental implementation details such as stack layout, processor scheduling, recursion strategy, or memory allocation.

The semantics therefore remains stable across a wide range of runtime architectures.

The historical interpretation has now been established at both operational and denotational levels.

The remaining challenge is practical rather than mathematical.

How should such histories be represented efficiently inside a real compiler, runtime system, and virtual machine?

The following chapter introduces the Universal Operator Machine, showing how the historical semantics developed throughout this work naturally gives rise to a unified execution architecture capable of supporting deterministic, probabilistic, fuzzy, differentiable, and concurrent computation through a single composition engine.

20 The Universal Operator Machine

The historical semantics developed throughout the preceding chapters suggest a different philosophy of language implementation. Conventional programming languages are typically organized around multiple specialized execution engines. Arithmetic expressions are evaluated by one subsystem, logical expressions by another, probabilistic reasoning by external libraries, automatic differentiation by specialized frameworks, and concurrency by an independent scheduler. Although these mechanisms interact, they often possess different operational semantics and different intermediate representations.

The Spherepop Calculus proposes a more uniform architecture.

Rather than constructing multiple execution engines, the runtime executes a single composition graph. The meaning of a computation is determined not by the execution engine itself but by the operators assigned to the graph.

This architecture is called the *Universal Operator Machine* (UOM).

The central observation motivating the UOM is remarkably simple.

Every computation may be viewed as a collection of bounded computational regions connected by dependency relationships. Regardless of whether those regions perform arithmetic, symbolic reasoning, probabilistic inference, logical deduction, or differentiable optimization, the scheduler performs exactly the same task.

It repeatedly locates admissible computational regions, evaluates them, records their historical evolution, and propagates their results to dependent regions.

The execution engine therefore remains invariant.

Only the operator library changes.

20.1 The Composition Graph

Internally, the UOM represents every program as a directed acyclic composition graph.

Each node possesses four fundamental components,

$$(\mathcal{O}, \mathcal{I}, \mathcal{O}', H),$$

where

- $\mathcal{O}$ denotes the operator,
- $\mathcal{I}$ denotes the collection of input dependencies,
- $\mathcal{O}'$ denotes the output produced after evaluation,
- H denotes the local computational history.

Unlike conventional abstract syntax trees, the graph does not primarily encode surface syntax.

Instead, it records computational dependency.

Every edge indicates that one computational region depends upon the successful completion of another.

Consequently, graph topology reflects admissibility rather than textual structure.

20.2 Scheduling

Execution proceeds by repeatedly identifying every node whose dependencies have already been satisfied.

If

$$\text{Deps}(N)$$

denotes the dependency set of node N, then

$$\text{Ready}(N) \iff \forall d \in \text{Deps}(N),\ \text{Resolved}(d).$$

Every Ready node becomes an admissible Sphere.

The scheduler therefore performs no symbolic reduction.

It merely maintains the frontier separating resolved histories from unresolved ones.

At each iteration,

$$\mathcal{F} = \{N \mid \text{Ready}(N)\},$$

is computed.

Every node belonging to the frontier may execute concurrently.

This naturally exposes parallelism without requiring explicit thread management in the language itself.

Sequential execution appears only when dependency relations force

$$|\mathcal{F}| = 1.$$

20.3 Compilation

Compilation becomes a sequence of graph transformations rather than textual rewriting.

The parser first constructs nested Spheres corresponding to lexical scope.

These Spheres are transformed into dependency graphs.

Subsequent optimization consists of graph rewriting while preserving historical semantics.

Unlike many optimizing compilers, the objective is not merely to reduce instruction count.

The objective is to preserve admissible historical evolution while simplifying the graph whenever possible.

Several familiar compiler optimizations become immediate graph transformations.

Inlining replaces a function node by the graph representing its body.

Constant propagation removes nodes whose outputs are already historically determined.

Dead code elimination removes computational regions whose outputs possess no reachable historical continuation.

Partial evaluation replaces subgraphs whose inputs have already become fixed through Replay.

Each optimization preserves the underlying historical interpretation even though the resulting graph becomes structurally simpler.

20.4 Operator Libraries

Perhaps the most distinctive feature of the Universal Operator Machine is that the graph itself possesses no intrinsic logical interpretation.

Only the operators determine semantics.

Suppose the graph contains the binary operator

$$\wedge.$$

Different operator libraries assign different meanings.
In Boolean computation,

$$x \wedge y$$

computes logical conjunction.
In fuzzy computation,
the same node computes

$$\min(x, y),$$

or another chosen t-norm.
In probabilistic reasoning,
the node combines probability measures.
In differentiable computation,
the node becomes a smooth function suitable for gradient propagation.
The surrounding graph remains identical.
Only the local interpretation of the operator changes.
Consequently, the execution engine never needs to distinguish between Boolean, fuzzy, probabilistic, symbolic, or differentiable computation.
Logic becomes a property of operators rather than of the runtime.
This separation substantially simplifies both implementation and language design.

20.5 Historical Persistence

Every executed node contributes an event to the computational history.
Unlike conventional runtimes that discard intermediate execution after evaluation, the Universal Operator Machine maintains explicit historical relationships between computational regions.
This does not imply that every intermediate object must remain resident in memory indefinitely.
Historical persistence is a semantic property rather than a storage policy.

Implementations remain free to compress, checkpoint, reconstruct, or archive historical information provided that Replay can reproduce every admissible continuation required by the semantics.

Consequently, implementation efficiency and historical correctness become independent concerns.

The language specifies historical meaning.

Individual runtimes determine the most efficient representation.

20.6 A Unified Execution Model

The Universal Operator Machine therefore replaces a collection of specialized execution engines with a single computational architecture.

The runtime understands only a small number of primitive concepts.

Computational regions.

Dependencies.

Historical continuation.

Admissibility.

Operator application.

Everything else emerges from their composition.

Arithmetic becomes one operator library.

Logic becomes another.

Probability becomes another.

Differentiable computation becomes another.

Concurrency arises from dependency rather than from separate scheduling constructs.

The resulting architecture possesses an unusual degree of uniformity.

The execution engine neither knows nor needs to know whether it is executing a proof assistant, a numerical simulation, a neural network, a symbolic algebra system, or a concurrent distributed program.

It evaluates histories.

The operators determine what those histories mean.

This separation between historical execution and semantic interpretation provides the conceptual foundation for the optimization techniques developed in the following chapter. Rather than optimizing individual programming constructs, the Spherepop compiler optimizes the evolution of computational histories themselves, preserving admissibility while minimizing the resources required to construct and replay them.

21 Optimization and Historical Compilation

The Universal Operator Machine separates computation from interpretation. Programs are executed as histories of admissible computational regions, while the semantic meaning of individual operations is determined entirely by the operator library. This separation naturally changes the philosophy of program optimization.

Traditional compilers optimize programs by transforming syntax trees into more efficient syntax trees. Intermediate representations are rewritten, variables are eliminated, expressions are folded, and control flow is simplified until an executable program emerges.

Spherepop instead regards optimization as the simplification of historical construction.

The compiler does not merely ask whether two programs produce identical outputs.

It asks whether they produce observationally equivalent histories.

Two programs are considered historically equivalent whenever every admissible future continuation observes the same computational behavior, even if the internal construction histories differ.

Consequently, optimization becomes the search for simpler histories that preserve admissible continuation.

21.1 Historical Equivalence

Let

$$H_1$$

and

$$H_2$$

be computational histories.

We write

$$H_1 \simeq H_2$$

whenever every admissible continuation beginning with H_1 corresponds to an observationally equivalent continuation beginning with H_2.

Unlike syntactic equality, historical equivalence ignores differences that cannot influence future computation.

The compiler is therefore free to replace

$$H_1$$

by

$$H_2$$

whenever

$$H_1 \simeq H_2.$$

Optimization becomes semantics-preserving history rewriting.

21.2 Inlining

Inlining replaces a function invocation by the computational graph representing its body.

Operationally,

$$f(x) \longrightarrow \text{Body}(f)[x].$$

Within Spherepop this is interpreted as replaying the historical construction of the function directly inside the calling Sphere.

No new semantic mechanism is introduced.

Inlining merely shortens the historical path separating two computational regions.

The resulting history contains fewer intermediate boundaries while preserving the same admissible continuations.

21.3 Partial Evaluation

Suppose a computational region depends upon values that have already become historically fixed.

Traditional compilers replace the corresponding expressions by constants.

Spherepop interprets this transformation historically.

Replay has already established the relevant construction.

Future executions therefore need not reconstruct it.

Only the unresolved portion of the history remains active.

Partial evaluation therefore removes unnecessary replay rather than merely substituting constants.

21.4 Dead History Elimination

Dead-code elimination traditionally removes computations whose outputs are never used.

Spherepop refines this notion.

A computational region is historically dead whenever no admissible future continuation depends upon its outcome.

Formally,

$$\text{Future}(H) = \varnothing$$

implies that

$$H$$

may be archived or removed from the active computational graph.

Notice that the semantic history still exists.

Only its operational representation disappears.

Historical persistence therefore remains intact even when physical storage is optimized.

21.5 Replay Compression

Replay introduces the possibility of reconstructing previous computations instead of storing every intermediate representation explicitly.

Suppose several computational regions share the common history

$$H.$$

Rather than storing identical copies,
the runtime stores

$$H$$

once together with multiple references.

Subsequent Replay operations reconstruct the required continuation whenever needed.

Replay therefore serves both as a semantic primitive and as a compression mechanism.

The distinction between execution and reconstruction largely disappears.

21.6 Merge Flattening

The associative and commutative properties of Merge imply that nested Merge trees possess many equivalent representations.

For example,

$$(H_1 \otimes H_2) \otimes (H_3 \otimes H_4)$$

may be rewritten as

$$H_1 \otimes H_2 \otimes H_3 \otimes H_4.$$

The flattened representation exposes maximal concurrency.

Dependency analysis then determines which computational regions actually require sequential execution.

Consequently, parallel scheduling emerges directly from graph topology rather than from explicit optimization passes.

21.7 Deferred Collapse

One of the central principles of the Spherepop Calculus is that commitment should occur only when necessary.

Whenever a computational region may continue symbolically, probabilistically, or historically without immediate evaluation, Collapse may be postponed.

Deferred Collapse preserves optionality.

Only when subsequent computation genuinely requires a committed value does the corresponding Pop become operationally necessary.

This principle applies equally to symbolic manipulation, probabilistic reasoning, and differentiable computation.

The compiler therefore attempts to delay irreversible commitment whenever such delay cannot alter future admissible continuations.

21.8 Garbage Collection and Historical Persistence

Conventional garbage collectors identify objects that are no longer reachable from active program roots.

Spherepop introduces an additional distinction.

Operational reachability is not identical to historical existence.

An object may cease participating in active computation while remaining part of the historical construction that produced later values.

The runtime therefore distinguishes three categories.

The first consists of active computational objects that remain necessary for future evaluation.

The second consists of dormant historical objects that are no longer active but whose provenance may still require replay.

The third consists of compressed archival histories that may be reconstructed only on demand.

Garbage collection therefore becomes history management rather than simple memory reclamation.

Implementations remain free to compress, serialize, checkpoint, or archive inactive histories provided that Replay preserves the semantics specified by the calculus.

21.9 Historical Optimization

These transformations illustrate a broader philosophical shift.

Traditional optimization minimizes instructions.

Spherepop minimizes historical complexity.

The objective is not simply to execute fewer operations but to construct the simplest admissible history capable of supporting every future continuation.

Programs therefore become progressively simpler histories rather than merely smaller collections of instructions.

This historical viewpoint unifies optimization, compilation, scheduling, and incremental execution within a single mathematical framework.

The compiler becomes an architect of computational history rather than merely a translator of syntax.

Having established both the execution model and its optimization principles, we are now prepared to compare the Spherepop Calculus with existing foundations of computation, including the λ-calculus, the π-calculus, combinatory logic,

dependent type theory, and contemporary proof assistants. These comparisons clarify which aspects of traditional theories are preserved, which are generalized, and which are fundamentally reinterpreted by the historical semantics developed throughout this work.

22 Abstract Syntax, Trees, and the Geometry of Computation

Before introducing the implementation of the Spherepop runtime, it is useful to clarify its relationship with existing compiler architecture. Nothing in the Spherepop Calculus eliminates the need for parsers, abstract syntax trees, symbol tables, or balanced search structures. These remain indispensable engineering tools. What changes is the semantic interpretation of the objects they represent.

The central claim of Spherepop is therefore not that modern compiler construction is incorrect, but that the mathematical object underlying compilation has traditionally been interpreted too narrowly. Syntax is a representation of computation rather than the computation itself.

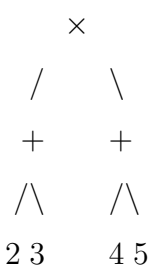
22.1 From Source Text to Abstract Syntax

Consider the arithmetic expression

$$(2 + 3) \times (4 + 5).$$

A compiler never evaluates the textual parentheses directly.

Instead, the parser constructs an Abstract Syntax Tree (AST) whose structure captures the grammatical organization of the expression,



The AST intentionally discards information that is irrelevant to computation.
Whitespace disappears.
Comments disappear.

Operator precedence becomes explicit.

Alternative textual representations that describe the same grammatical structure collapse to the same tree.

The compiler therefore separates the accidental properties of the source code from its computational organization.

This abstraction has proven enormously successful and remains one of the foundations of compiler engineering.

Spherepop fully adopts this observation.

The parser should continue constructing abstract syntax trees.

The difference lies in what those trees are understood to describe.

22.2 Syntax Describes Regions

Traditional operational semantics often treats the AST itself as the primary object of computation. Evaluation proceeds by recursively traversing the tree, locating reducible expressions, applying rewrite rules, and replacing subtrees.

Spherepop instead interprets the tree as a description of nested computational regions.

Each internal node corresponds to the creation of a Sphere.

Each subtree describes the local computation occurring inside that Sphere.

Each completed subtree becomes admissible for Pop once every interior dependency has been resolved.

Consequently, the AST is not the computation.

It is a map of computational geography.

The runtime does not fundamentally execute a tree.

It executes the historical evolution of the computational regions described by that tree.

22.3 Trees Describe Structure, Histories Describe Evolution

The distinction becomes particularly important when considering execution.

An abstract syntax tree answers structural questions.

Given a node, one may determine

- its operator,
- its operands,
- its lexical scope,

- its immediate dependencies.

These are static properties.
The tree itself contains no notion of time.
It does not record

- when a computational region became admissible,
- which Sphere was popped first,
- why one continuation was refused,
- how a value propagated through Replay,
- which historical events produced the present computation.

These questions concern evolution rather than structure.
The AST therefore provides a blueprint.
The history records the construction of the building itself.
Spherepop complements the tree with a persistent historical semantics rather than replacing it.

22.4 Symbol Tables and Balanced Trees

Modern compilers contain many trees that are unrelated to program structure.
Identifiers, variable bindings, namespaces, modules, and type environments must all be stored efficiently.
Balanced search trees such as AVL trees, B-trees, or Red–Black Trees are commonly employed for this purpose.
For example, a symbol table containing

$$\{\texttt{pressure}, \texttt{velocity}, \texttt{temperature}, \texttt{density}\}$$

may be organized as a Red–Black Tree to guarantee logarithmic lookup time,

$$O(\log n).$$

This tree has no semantic relationship to the computation itself.
It is simply an efficient implementation of associative lookup.
Searching for a variable name should be fast regardless of how the program is evaluated.

Spherepop therefore has no reason to replace these classical data structures.

A Spherepop compiler would almost certainly continue to employ balanced search trees, hash tables, radix trees, and other well-established indexing mechanisms.

Their role is organizational rather than semantic.

22.5 Dependency Graphs

As optimization progresses, most modern compilers gradually move away from pure trees.

Common subexpressions are shared.

Variables acquire multiple users.

Control flow introduces branching and merging.

The representation naturally evolves into a directed acyclic graph.

From the Spherepop perspective this evolution is unsurprising.

The computational object was never fundamentally a tree.

The tree merely reflected the surface syntax.

The underlying object has always been a network of computational dependencies.

Spherepop simply makes this dependency structure primary from the beginning.

Histories evolve along dependency relations rather than textual nesting.

22.6 Engineering Versus Semantics

The distinction between implementation and semantics should therefore remain clear throughout this work.

The parser constructs abstract syntax trees.

The compiler builds dependency graphs.

Symbol tables are stored in balanced search structures.

Optimizers perform graph rewriting.

Schedulers determine admissible evaluation order.

None of these techniques disappear.

Spherepop changes only the semantic interpretation of the resulting structures.

Instead of viewing evaluation as recursive tree rewriting, it views evaluation as the historical evolution of nested computational regions.

The engineering remains familiar.
The ontology changes.
Syntax becomes a representation.
History becomes the primary computational object.

23 Comparisons with Classical Computational Foundations

Every new computational calculus inherits a substantial body of prior work. The purpose of Spherepop is therefore not to replace the remarkable achievements of classical theoretical computer science, but to reinterpret their common foundations through a different ontological lens. Lambda calculus, combinatory logic, abstract machines, process calculi, type theory, compiler theory, and proof assistants each illuminate essential aspects of computation. Spherepop argues that these diverse formalisms may be understood as different presentations of a more primitive phenomenon: the creation, interaction, and resolution of bounded computational regions whose evolution forms an irreversible history.

The closest historical relative of the Spherepop Calculus is undoubtedly the untyped λ-calculus. Church's remarkable observation that abstraction and application alone suffice to express arbitrary computation remains one of the great achievements of twentieth-century mathematics. Every functional programming language, proof assistant, and dependent type theory owes a debt to this discovery. Spherepop preserves this expressive power entirely. Functions remain computational abstractions, and evaluation remains the process through which abstractions are instantiated by concrete arguments. What changes is the interpretation of the computational act itself. In the λ-calculus, the central reduction rule is β-reduction, where substitution transforms one syntactic expression into another. Spherepop instead interprets this same event as the dissolution of a computational boundary. Function application becomes a Pop, substitution becomes Replay, and evaluation becomes the historical evolution of nested computational regions. The expressive capabilities remain unchanged, but the primitive ontology shifts from symbolic rewriting to geometric continuation.

This distinction becomes increasingly significant as systems become more complex. The λ-calculus excels at describing local reduction, but says comparatively little about provenance, historical reconstruction, deferred commitment, or the persistent record of computational construction. These capabilities are typically

introduced later through debugging systems, incremental compilers, provenance trackers, build systems, transaction logs, distributed consensus algorithms, and version-control systems. Spherepop attempts to incorporate these notions directly into the semantic foundation. Instead of treating computational history as auxiliary metadata generated by external tools, history becomes the primary mathematical object from which evaluation itself is derived.

Combinatory logic occupies a similarly important position. By eliminating variables entirely, combinatory logic demonstrates that abstraction does not fundamentally require symbolic names. Spherepop agrees with this insight. Variable names are not ontologically primitive. They are conveniences for human communication. Historical construction proceeds perfectly well without them because Replay reconstructs admissible continuations directly from computational history. In this sense, Spherepop lies philosophically closer to combinatory logic than to naive symbolic substitution, while simultaneously recovering much of the readability that explicit variables provide.

The relationship with process calculi, particularly the π-calculus, reveals another point of comparison. The π-calculus extends functional computation by making communication itself a first-class object. Channels may be created dynamically, transmitted between processes, and composed into complex networks of interacting agents. Spherepop preserves these ideas while placing greater emphasis upon computational regions rather than communication channels. Independent Spheres evolve concurrently through the Merge operator, while communication appears as historical dependency between computational regions. Rather than introducing concurrency as a separate semantic layer, Spherepop derives concurrency from the admissibility of independent Pops. Communication therefore becomes one particular form of historical interaction rather than the defining primitive of concurrent computation.

Linear logic offers another illuminating comparison because it interprets computation through the disciplined management of resources. Weakening, contraction, and exchange are no longer unrestricted structural rules but must respect explicit resource usage. Spherepop reaches a remarkably similar conclusion from an entirely different direction. Histories cannot simply be duplicated or erased arbitrarily because they record actual computational construction. Replay provides controlled reuse, Refusal prevents inadmissible extension, and historical monotonicity ensures that admitted events cannot be removed from the computational record. Consequently, many resource-sensitive properties emerge naturally from

the historical interpretation without requiring an entirely separate logical foundation.

The comparison with dependent type theory is perhaps the most direct. Contemporary systems such as the Calculus of Constructions interpret contexts as collections of assumptions and types as classifications of admissible terms. Spherepop preserves every essential mathematical capability of dependent typing while changing its conceptual interpretation. Contexts become histories rather than environments. Types become refusal structures rather than sets of values. Dependent functions become continuation policies. Dependent pairs become staged historical constructions. Identity types become repairs between histories rather than merely proofs of equality. None of the logical power of dependent type theory is sacrificed. Instead, the familiar objects acquire a historical semantics that integrates naturally with the operational structure developed throughout this work.

Proof assistants such as Lean, Coq, Agda, and Idris provide compelling demonstrations of the practical value of dependent type theory. These systems construct machine-checkable mathematical proofs whose correctness follows from small trusted kernels. Spherepop should not be understood as a replacement for such systems. Rather, it proposes an alternative semantic substrate upon which similar proof technologies could potentially be constructed. Existing proof assistants primarily manipulate symbolic derivations. A Spherepop-based proof assistant would instead manipulate historical constructions whose provenance, Replay, Refusal, and repair remain explicit throughout proof development. Whether this historical approach ultimately yields practical advantages in proof engineering remains an empirical question deserving future investigation.

Compiler theory likewise remains fundamentally unchanged. Lexical analysis, parsing, abstract syntax trees, dependency graphs, Static Single Assignment, control-flow analysis, register allocation, instruction scheduling, and code generation all remain essential engineering achievements. Spherepop neither replaces nor diminishes these techniques. Rather, it argues that each may be understood as manipulating representations of computational history. Abstract syntax describes the geometry of computational regions. Dependency graphs describe admissible historical continuations. Optimization simplifies historical construction while preserving observational equivalence. The engineering remains largely familiar, while the semantic interpretation shifts from syntax to history.

Viewed from this perspective, Spherepop is less a rejection of existing computational foundations than an attempt to provide a common conceptual framework

within which many of them may coexist. Lambda calculus emphasizes functional abstraction. Process calculi emphasize communication. Linear logic emphasizes resources. Dependent type theory emphasizes logical construction. Compiler theory emphasizes efficient implementation. Spherepop interprets each of these as describing different aspects of a single historical process: the opening of computational regions, the evolution of admissible continuations, the replay of historical dependencies, the refusal of incoherent extensions, and the eventual dissolution of computational boundaries through Pop. The claim is therefore not that previous theories were mistaken, but that they may all be viewed as local perspectives on a broader geometry of computational history.

24 Future Directions

The Spherepop Calculus presented in this work should be regarded as the beginning of a research program rather than its conclusion. While the historical interpretation developed throughout the preceding chapters provides a coherent operational and denotational foundation, many important theoretical and practical questions remain open. Some concern the mathematical structure of the calculus itself, while others concern implementation, optimization, distributed computation, formal verification, and the broader relationship between historical semantics and existing computational paradigms. The framework has deliberately been developed from elementary arithmetic through nested scope and historical continuation before introducing more sophisticated ideas. This progression reflects the belief that future extensions should likewise emerge naturally from the historical ontology rather than through the addition of unrelated mechanisms.

One immediate direction concerns the mathematical study of historical equivalence. Throughout this work, two computational histories have been regarded as equivalent whenever every admissible future continuation observes no meaningful distinction between them. While this intuition parallels notions such as contextual equivalence, bisimulation, and observational congruence, a complete characterization remains to be developed. One expects that historical equivalence will possess a rich algebraic structure admitting canonical normalization procedures, quotient constructions, and categorical characterizations analogous to those developed for traditional rewriting systems. Such results would provide the formal foundation for aggressive history-preserving compiler optimizations while simultaneously clarifying the mathematical meaning of Replay and repair.

Another important direction concerns normalization. Ordinary deterministic evaluation inherits many familiar normalization arguments from existing type theories and rewriting systems. The probabilistic fragment introduced through Choice, however, remains less completely understood. While operational sampling and distribution-valued semantics appear compatible with the historical interpretation, questions concerning almost-sure normalization, confluence under probabilistic scheduling, and the interaction between delayed Collapse and historical Replay deserve careful investigation. These questions are not unique to Spherepop, but the historical semantics provides a new language in which they may be formulated.

The relationship between historical semantics and incremental computation also deserves extensive study. Modern software development increasingly depends upon systems capable of avoiding unnecessary recomputation. Incremental compilers, reactive programming environments, distributed build systems, interactive proof assistants, notebook environments, and database query optimizers all rely upon reconstructing only those portions of computation whose dependencies have changed. Spherepop suggests that such systems are not merely implementation techniques but manifestations of Replay itself. A general mathematical theory of incremental computation formulated directly in terms of historical continuation could potentially unify a wide variety of existing engineering practices under a common semantic framework.

The implementation of persistent histories likewise presents numerous research opportunities. The semantics developed here deliberately distinguishes historical persistence from physical storage. Nothing requires every intermediate computational object to remain permanently resident in memory. Instead, histories may be compressed, checkpointed, archived, reconstructed, or distributed provided that Replay preserves their semantic meaning. Determining the optimal balance between storage, reconstruction cost, and execution efficiency therefore becomes an optimization problem in its own right. Future runtimes may employ adaptive compression strategies guided by expected Replay frequency, historical locality, or probabilistic prediction of future continuations.

Distributed computation provides another natural application of the historical ontology. Existing distributed systems devote substantial effort to consensus, event ordering, provenance tracking, and conflict resolution. Many of these problems arise because distributed computation is traditionally described in terms of mutable state replicated across multiple machines. Spherepop instead sug-

gests representing distributed execution as the evolution of partially ordered histories whose admissibility depends only upon causal dependency rather than global synchronization. Replay, repair, and historical equivalence may therefore provide alternative semantic foundations for distributed compilation, replicated databases, collaborative theorem proving, and decentralized software development.

The relationship between historical semantics and formal verification also remains largely unexplored. Contemporary verification techniques typically reason about programs either through symbolic execution or through logical proof over static specifications. Because Spherepop records computational construction explicitly, verification may instead become a property of historical evolution itself. Proof obligations, resource constraints, capability policies, communication protocols, and security guarantees all appear naturally as admissibility conditions governing historical extension. Investigating whether this viewpoint simplifies practical verification remains an important empirical question.

Machine learning provides another intriguing direction. Contemporary learning systems often distinguish sharply between symbolic reasoning, differentiable optimization, probabilistic inference, and neural computation. The Universal Operator Machine instead suggests that these distinctions belong primarily to the operator library rather than the execution engine. If this hypothesis proves fruitful, symbolic theorem proving, probabilistic reasoning, fuzzy inference, automatic differentiation, and neural computation may all be implemented upon the same historical composition graph, differing only in the local semantics assigned to operators. Such unification would considerably simplify the conceptual architecture of intelligent software systems while preserving the specialized mathematical structures required by each domain.

Finally, the broader philosophical implications of the historical ontology remain open for exploration. Throughout the development of the Spherepop Calculus, the emphasis has gradually shifted away from static mathematical objects toward constructive evolution. Contexts became histories. Types became disciplined continuation. Equality became repair. Evaluation became the controlled dissolution of computational boundaries. Programs became historical objects rather than inert symbolic expressions. Whether this historical perspective ultimately proves useful beyond programming language theory—in formal mathematics, distributed knowledge systems, scientific provenance, autonomous reasoning, or the foundations of logic itself—will depend upon the ability of future

work to demonstrate both theoretical elegance and practical utility.

The central proposal of this paper is intentionally modest despite its broad ambitions. Every student already learns the essential computational operation during elementary arithmetic: locate the innermost computational scope, evaluate it, and continue outward until no unresolved scopes remain. Spherepop argues that this familiar procedure is more fundamental than the syntactic notation traditionally used to express it. By replacing parentheses with explicit computational regions and replacing symbolic rewriting with the historical evolution of those regions, the calculus seeks not to overturn the existing foundations of computation, but to reveal a common historical structure underlying them. Whether this historical foundation ultimately becomes a useful basis for future programming languages, proof assistants, compiler architectures, and concurrent systems remains a question for future research, but it is a question whose investigation appears both mathematically natural and computationally promising.

25 Meta-Theoretical Properties

A computational calculus is valuable not merely because it admits elegant programs, but because its fundamental operations satisfy predictable mathematical properties. Throughout the development of the Spherepop Calculus, we have progressively replaced traditional syntactic notions with historical ones. Parentheses became Spheres. Substitution became Replay. Type checking became admissibility. Reduction became Pop. Histories replaced mutable states. These reinterpretations naturally raise the question of whether the resulting calculus preserves the familiar guarantees expected of modern programming languages and proof systems. The purpose of this section is not to provide fully formal proofs of every theorem—which are deferred to the appendices—but to establish the principal meta-theoretical properties that characterize the historical semantics.

The first and perhaps most fundamental property is the monotonicity of histories. Unlike conventional operational semantics, where machine states are replaced by successor states during evaluation, Spherepop never destroys an admitted computational event. Every successful evaluation extends an existing history. Formally, if

$$H \longrightarrow H'$$

is a valid operational transition, then

$$H \subseteq H'.$$

This inclusion should not be interpreted as ordinary set inclusion, since histories possess an intrinsic ordering determined by computational construction. Rather, it expresses the fact that every event admitted into a history remains part of every future continuation. New events may be appended, Replay may reconstruct previous continuations, and Refusal may prevent inadmissible extensions, but no admitted event is erased. This monotonicity provides the semantic foundation for provenance, incremental computation, historical replay, and reproducibility.

A second property concerns the preservation of admissibility. Traditional type theory establishes the Preservation Theorem, asserting that well-typed programs remain well typed throughout evaluation. Spherepop generalizes this principle from types to histories. Suppose that

$$\Gamma \vdash H$$

is an admissible historical construction, and suppose that

$$H \longrightarrow H'$$

is an operational transition generated by Replay, Pop, Merge, or Choice. Then the extended history likewise satisfies

$$\Gamma' \vdash H',$$

where

$$\Gamma'$$

denotes the enlarged historical context produced by the transition. The significance of this theorem lies in the interpretation of contexts as histories. Preservation no longer states merely that values retain their types. It states that admissible historical construction remains admissible throughout the evolution of the computation. Correctness therefore becomes a property of historical growth rather than static classification.

The corresponding Progress Theorem requires more careful interpretation. In traditional operational semantics, progress asserts that every well-typed term is

either already a value or admits another reduction step. Such a theorem is appropriate for deterministic rewriting systems in which evaluation must always continue whenever possible. Spherepop deliberately weakens this principle. Histories may legitimately reach situations where no immediate Pop is admissible because additional information must first become available, because a probabilistic Choice has not yet been resolved, because an external resource has not yet responded, or because a continuation has been explicitly refused. Rather than forcing evaluation into artificial reductions, the Spherepop Progress Theorem states that every admissible history is either historically complete, admits at least one admissible continuation, or enters a deferred state awaiting future admissible construction. Deferred computation is therefore treated as a first-class semantic object rather than as an implementation inconvenience. This weakening reflects the practical realities of concurrent, distributed, interactive, and probabilistic computation more faithfully than classical formulations.

Replay itself satisfies an important correctness property. Operationally, Replay reconstructs historical continuations rather than performing symbolic substitution. The Replay Correctness Theorem asserts that every Replay operation preserves observational behavior. If a continuation constructed through Replay produces an observable value

$$v,$$

then every admissible future continuation observes the same value that would have been obtained through the corresponding symbolic substitution. Replay therefore strictly generalizes substitution. Ordinary substitution appears as the special case in which historical provenance is intentionally discarded. The theorem guarantees that retaining provenance does not alter computational meaning.

Merge introduces additional questions concerning concurrency. Because Merge represents logical independence rather than processor scheduling, independent histories should remain insensitive to arbitrary execution order. This property may be expressed through historical commutativity. Whenever two computational regions possess no dependency relation, exchanging the order of their Pops yields observationally equivalent histories. Consequently, semantic correctness depends only upon causal structure rather than scheduler behavior. This theorem provides the mathematical justification for parallel execution, distributed evaluation, speculative computation, and asynchronous scheduling within the Universal Operator Machine.

Choice requires a corresponding adequacy result. Operationally, Choice may either sample an immediate continuation or propagate an explicit probability distribution. Denotationally, both interpretations represent different views of the same historical construction. Adequacy therefore states that repeated operational sampling converges to the probability measure specified by the denotational semantics. Although a complete proof depends upon the chosen probability model and is deferred to future work, the historical semantics has been deliberately constructed so that probabilistic commitment affects only the interpretation of operators rather than the underlying computational graph. This separation substantially simplifies the relationship between deterministic and probabilistic computation.

Refusal possesses perhaps the simplest meta-theoretical characterization. Unlike exceptions or runtime failures, Refusal preserves historical coherence. An inadmissible continuation never becomes part of the computational history. Consequently, every reachable history remains admissible by construction. There exists no operational rule capable of extending a history by an event that violates the continuation discipline. This property transforms many traditional safety arguments into immediate consequences of the operational semantics. Safety is preserved not because incorrect states are later repaired, but because incoherent histories are never admitted in the first place.

Finally, the historical ontology itself satisfies a conservation principle. Every computational object possesses a provenance. Every observable value admits a historical explanation. Every explanation consists of a finite sequence of admissible events whose Replay reconstructs the value without introducing additional assumptions. In this sense, computation satisfies a form of historical completeness: nothing observable appears without an admissible construction, and every admissible construction remains available, at least semantically, for future replay. This principle distinguishes the Spherepop Calculus from purely state-based operational semantics and forms the conceptual foundation upon which provenance tracking, incremental execution, repair-oriented programming, and historical verification are built.

Taken together, these properties suggest that the historical interpretation retains the mathematical discipline traditionally associated with operational semantics while extending its expressive scope to encompass provenance, concurrency, deferred computation, probabilistic evaluation, and historical repair. The familiar guarantees of preservation, progress, compositionality, and correctness are

not discarded. Rather, they are reformulated in terms of the growth and admissibility of computational histories, reflecting the central philosophical claim of this work that computation is more naturally understood as irreversible historical construction than as transient symbolic state transformation.

26 Implementation Case Studies

The mathematical development of the Spherepop Calculus establishes a general theory of historical computation. Nevertheless, the practical value of any computational framework ultimately depends upon its ability to express familiar programs naturally while preserving the advantages of its semantic foundation. The purpose of this section is therefore to illustrate how the historical interpretation applies across increasingly sophisticated computational examples. Rather than introducing new primitives, these case studies demonstrate that arithmetic, functional programming, concurrent execution, logical reasoning, probabilistic computation, and differentiable systems all emerge from the same small collection of operations: Sphere, Pop, Replay, Merge, Choice, Refusal, and Collapse.

The simplest example is ordinary arithmetic. Consider the expression

$$((2 + 3) \times (4 + 5)).$$

Traditional operational semantics interprets this as a syntax tree whose interior expressions are repeatedly rewritten until a normal form is reached. The Spherepop interpretation begins one level earlier. The parser identifies three nested computational regions. Two interior Spheres contain independent additions, while the enclosing Sphere contains the multiplication that depends upon their completion. The scheduler immediately recognizes that both interior regions are simultaneously admissible because neither depends upon the other. Two independent Pops therefore become available. Once both additions have completed, Replay propagates their values into the enclosing computational region, rendering the multiplication admissible. The final Pop dissolves the outer Sphere, completing the history. The numerical answer is identical to that produced by conventional evaluation, yet the execution history now contains explicit information concerning computational independence, dependency, evaluation order, and provenance.

Recursive computation illustrates a second aspect of the historical semantics. Suppose the factorial function is defined recursively. Traditional execution cre-

ates a sequence of stack frames whose lifetime is determined by return order. Once evaluation completes, these frames disappear. Within Spherepop, each recursive invocation becomes a new Sphere extending the existing history. Every recursive descent opens another computational region whose eventual Pop contributes to the historical construction. The resulting history therefore forms a nested sequence of computational boundaries rather than a transient call stack. Tail-call optimization, recursion elimination, and memoization may still be performed by the compiler whenever historical equivalence can be established, but these optimizations are understood as transformations of historical construction rather than merely manipulations of stack frames.

Concurrent computation demonstrates the importance of Merge. Consider two independent numerical simulations that eventually contribute to a common optimization problem. Traditional runtimes typically create separate threads, tasks, or asynchronous futures. Spherepop instead represents both simulations as independent computational histories connected through a Merge operation. Neither simulation requires knowledge of the other during execution. Their histories evolve independently until the optimization stage introduces the first dependency between them. At that moment Replay constructs the historical context required for the optimization itself. The scheduler remains entirely agnostic concerning processor allocation, execution order, or hardware architecture. It need only preserve the dependency structure encoded by the histories. Parallelism therefore appears as a semantic consequence of historical independence rather than as a distinct programming construct.

Probabilistic computation illustrates the complementary role of Choice. Suppose an algorithm repeatedly samples candidate solutions according to a probability distribution before selecting the most promising continuation. Traditional probabilistic programming often introduces specialized execution engines for sampling, inference, and distribution propagation. Within Spherepop, each uncertain decision simply becomes a Choice node within the historical composition graph. If immediate execution is required, the Choice operator samples a continuation and records the resulting commitment within the computational history. Alternatively, the runtime may preserve the entire probability distribution as a first-class historical object, delaying Collapse until subsequent evidence becomes available. The surrounding execution engine remains unchanged. Only the operator library determines whether uncertainty is resolved immediately or propagated symbolically.

The same historical interpretation extends naturally to fuzzy computation. Suppose the computational graph represents a control system whose logical operators evaluate degrees of truth rather than binary propositions. The graph itself requires no modification. Nodes previously interpreted as Boolean conjunctions now implement continuous t-norms, while disjunctions become their corresponding t-conorms. Histories continue to evolve exactly as before. Replay propagates intermediate values, Merge exposes independent evaluation opportunities, and Pop dissolves completed computational regions. Only the local interpretation of operators changes. The execution engine therefore remains completely independent of the chosen logical framework.

Differentiable computation provides perhaps the most compelling demonstration of the Universal Operator Machine. Modern machine learning systems often construct explicit computational graphs through which gradients propagate during backpropagation. From the historical perspective, such graphs differ remarkably little from those developed throughout this paper. Forward evaluation constructs an admissible computational history. Reverse-mode automatic differentiation simply traverses the same historical construction in the opposite direction, propagating derivatives instead of values. Replay already records the provenance required to reconstruct every intermediate computation. Consequently, gradient propagation emerges as another form of historical continuation rather than as a fundamentally distinct execution mechanism. Symbolic reasoning, probabilistic inference, and neural optimization therefore become different semantic interpretations of a common historical substrate.

Logic circuits provide another illuminating example. A combinational digital circuit may be interpreted as a directed acyclic graph whose gates become admissible whenever their input signals have stabilized. Each gate therefore corresponds naturally to a computational Sphere. Input propagation extends the history through Replay, while each gate output becomes available only after its local computation has completed. Entire collections of independent gates may be evaluated simultaneously through Merge. Replacing Boolean operators by continuous activation functions immediately transforms the same graph into a differentiable neural computation. Replacing them again by probabilistic operators produces a Bayesian network. The graph remains invariant throughout these transformations. Only the operator semantics vary.

These examples collectively illustrate a recurring theme that has appeared throughout the development of the Spherepop Calculus. The underlying com-

putational graph is remarkably stable across domains. Arithmetic, functional programming, concurrency, symbolic logic, fuzzy reasoning, probabilistic inference, circuit simulation, and differentiable optimization all share essentially the same historical organization. Their apparent differences arise primarily from the local semantics assigned to operators rather than from fundamentally different execution models. The Universal Operator Machine therefore achieves its generality not by supporting many different runtimes, but by recognizing that a single history-preserving composition engine suffices for an unexpectedly broad range of computational paradigms. This observation reinforces the central thesis of the paper: computation is most naturally understood as the historical evolution of bounded computational regions, while individual programming paradigms emerge through different interpretations of the operators acting upon those regions.

27 Practical Runtime Architecture

The historical semantics developed throughout this work intentionally separates the mathematical foundations of computation from the engineering decisions required to implement an efficient runtime. This distinction is important. Spherepop does not propose replacing decades of compiler research with an entirely new implementation methodology. Existing parsers, lexical analyzers, optimizers, dependency analyzers, memory allocators, and code generators have been refined over many decades and remain highly effective. Instead, the Spherepop runtime provides a different semantic interpretation of these components. Traditional compiler pipelines manipulate symbolic structures whose meaning is expressed through operational rewriting. The Spherepop runtime manipulates historical constructions whose meaning is expressed through the evolution of computational regions. The engineering remains largely familiar; the ontology changes.

Compilation begins in the conventional manner. Source text is transformed into tokens through lexical analysis, after which a parser constructs an Abstract Syntax Tree representing the grammatical organization of the program. At this stage, the implementation differs little from contemporary compiler architectures. The parser identifies lexical scope, operator precedence, declarations, and expression boundaries exactly as it would for an ordinary functional language. Spherepop deliberately preserves this mature engineering practice because the AST re-

mains an excellent representation of syntactic structure. The historical interpretation begins only after parsing has been completed.

The first transformation unique to Spherepop converts the abstract syntax tree into a graph of computational regions. Every syntactic scope becomes an explicit Sphere, every dependency becomes an edge within the composition graph, and every lexical declaration contributes a new historical event rather than merely extending a symbol table. Although the resulting graph frequently resembles the dependency graphs already employed by optimizing compilers, its nodes possess richer semantic content. Each node records not only its operator and dependencies but also the historical conditions under which it becomes admissible for evaluation. The graph therefore describes both computational structure and historical evolution simultaneously.

The scheduler forms the operational heart of the runtime. Rather than recursively traversing syntax trees searching for reducible expressions, the scheduler maintains a frontier of computational regions whose dependencies have all been satisfied. Every node on this frontier represents an admissible Sphere ready for Pop. Independent regions may therefore be evaluated concurrently without requiring additional semantic machinery. The scheduler is concerned solely with historical dependency. Processor allocation, thread management, asynchronous execution, and distributed deployment remain implementation decisions delegated to lower levels of the runtime. Because the semantic notion of readiness is separated from physical scheduling, the same program may execute correctly on a single processor, a multicore workstation, or a distributed cluster without altering its historical meaning.

The history manager constitutes the principal innovation of the runtime. Traditional interpreters frequently discard intermediate computational state once evaluation has completed. Spherepop instead records the sequence of historical events responsible for constructing every observable value. This does not imply that every intermediate object must remain permanently resident in memory. The runtime distinguishes semantic persistence from physical representation. Histories may be compressed, checkpointed, serialized, archived, reconstructed through Replay, or discarded from active memory provided that their observable behavior remains recoverable. The history manager therefore resembles a transactional journal combined with an incremental dependency database. Its purpose is not merely to record execution for debugging, but to provide the semantic substrate upon which Replay, incremental recompilation, provenance

analysis, historical verification, and repair-oriented programming naturally operate.

Replay itself is implemented as a dependency reconstruction mechanism rather than a symbolic substitution engine. Whenever a computational region requires the output of previous regions, the runtime reconstructs the admissible historical context necessary for continuation. In many situations, Replay requires no explicit reconstruction because the relevant history remains resident within the dependency graph. When portions of the history have been compressed or archived, Replay reconstructs only those computational events whose provenance remains necessary for the requested continuation. This property allows historical persistence to coexist with practical resource management, preventing the semantic richness of the calculus from imposing unreasonable storage requirements upon implementations.

The operator library forms an independent layer within the runtime. Every node in the composition graph specifies only the operator to be applied together with its dependencies. The runtime itself possesses no intrinsic knowledge of Boolean logic, arithmetic, fuzzy inference, probabilistic sampling, differentiable optimization, or symbolic theorem proving. These behaviors are provided entirely by interchangeable operator libraries implementing a common interface. Consequently, extending the language to support new computational domains rarely requires modifications to the scheduler or history manager. Instead, developers introduce new operator collections whose local semantics integrate immediately with the existing historical execution model. The Universal Operator Machine therefore separates execution from interpretation with unusual clarity.

Memory management likewise acquires a historical interpretation. Conventional garbage collectors determine whether objects remain reachable from active program roots. Spherepop introduces a richer notion of reachability. Active objects participate directly in ongoing computation. Dormant historical objects no longer influence current execution but remain semantically relevant for future Replay. Archived histories have been compressed or externalized yet remain reconstructible. The runtime therefore manages several distinct forms of persistence simultaneously, balancing execution efficiency against future historical requirements. This layered interpretation of memory suggests new opportunities for adaptive storage systems capable of predicting which historical regions are most likely to require future Replay.

Finally, the runtime architecture illustrates one of the central philosophical

claims of the Spherepop Calculus. Compilers, virtual machines, dependency graphs, schedulers, parsers, symbol tables, and optimizers continue to exist almost exactly as they do within contemporary programming language implementations. The novelty lies not in replacing these mature technologies, but in recognizing that they collectively manipulate a richer mathematical object than has traditionally been acknowledged. The execution engine no longer views a program merely as a mutable symbolic expression undergoing successive rewrites. Instead, it interprets the program as an evolving history of bounded computational regions whose openings, interactions, refusals, replays, and Pops collectively constitute the computation itself. This historical interpretation allows familiar engineering techniques to coexist with a substantially broader semantic foundation, providing a practical bridge between established compiler construction and the historical ontology developed throughout this work.

28 Conclusion

The central claim of this paper has been deliberately developed from the simplest possible starting point. Before students encounter formal logic, compiler theory, category theory, or programming language semantics, they already know how to perform a computation. They are taught to locate the innermost parenthesized expression, evaluate it, replace it by its result, and repeat until no unresolved scope remains. This elementary arithmetic procedure is so familiar that its conceptual significance is easily overlooked. Yet it contains, in embryonic form, the essential mechanics of symbolic computation. The Spherepop Calculus begins by asking whether this procedure, rather than the symbolic notation traditionally used to describe it, should be regarded as the true primitive of computation.

From this modest observation emerges a different computational ontology. Parentheses cease to be viewed as punctuation marks decorating symbolic expressions and instead become explicit computational regions. Evaluation is no longer interpreted primarily as syntactic rewriting but as the dissolution of admissible computational boundaries through the Pop operation. Functions, blocks, namespaces, modules, recursive calls, concurrent tasks, and logical subproofs all become manifestations of the same geometric principle: the creation and eventual resolution of bounded computational regions. The language of trees, scopes, stacks, and environments is therefore unified under a single interpretation of nested historical construction.

The historical perspective developed throughout this work changes the role of nearly every familiar concept in programming language theory while preserving their practical usefulness. Contexts become histories rather than collections of assumptions. Substitution becomes Replay, reconstructing historical continuation instead of replacing isolated symbols. Types become refusal structures governing admissible extension rather than merely classifying completed values. Identity becomes repair between computational histories. Merge expresses logical independence, while Choice separates uncertainty from commitment. None of these reinterpretations discard the mathematical insights of classical theory. Instead, they place them within a framework where provenance, dependency, concurrency, incremental execution, and historical reconstruction become intrinsic semantic properties rather than auxiliary engineering concerns.

One of the recurring themes of this paper has been the separation of execution from interpretation. The Universal Operator Machine does not distinguish between arithmetic, Boolean logic, fuzzy reasoning, probabilistic inference, symbolic manipulation, or differentiable optimization through specialized execution engines. Instead, it executes a single historical composition graph whose local operator library determines the semantics of individual nodes. The same computational history may therefore be interpreted under multiple logical regimes without changing its structural organization. Computation becomes the construction of history, while logic becomes a property of operators rather than of the runtime itself. This separation provides a conceptual simplicity that is often obscured by the proliferation of specialized computational frameworks.

Equally important is the relationship between the Spherepop Calculus and existing compiler architecture. Throughout this work, no attempt has been made to replace parsers, abstract syntax trees, dependency graphs, symbol tables, Static Single Assignment, register allocation, or graph rewriting. These remain indispensable achievements of compiler engineering. The contribution of Spherepop lies instead in providing a semantic interpretation capable of unifying these structures under a common historical ontology. Abstract syntax describes computational geography. Dependency graphs describe admissible continuation. Scheduling identifies computational regions ready for Pop. Optimization simplifies historical construction while preserving future Replay. The engineering remains familiar precisely because the calculus seeks to reinterpret, rather than replace, existing implementation practice.

The historical interpretation also offers a different perspective on software cor-

rectness. Conventional verification often treats correctness as a property of completed programs or terminal machine states. Spherepop instead views correctness as the preservation of admissible historical growth. Every successful computation extends an already coherent history. Every Refusal prevents incoherent extension before commitment occurs. Every Replay preserves historical provenance. Every Pop records an irreversible computational event. Correctness therefore emerges continuously throughout execution rather than appearing only at its conclusion. This shift aligns naturally with incremental compilation, distributed systems, provenance tracking, repair-oriented programming, and interactive theorem proving, suggesting that many apparently unrelated computational disciplines may ultimately share a common historical foundation.

The broader significance of the Spherepop Calculus lies less in any individual primitive than in the conceptual perspective it offers. Computation is no longer viewed as the manipulation of isolated symbolic expressions passing through successive machine states. Instead, it becomes the irreversible construction of an evolving historical object whose boundaries open, interact, replay, refuse, merge, and ultimately dissolve through Pop. The emphasis moves from static description toward constructive evolution, from symbolic replacement toward historical continuation, and from transient state toward persistent computational provenance. This historical viewpoint does not claim to supersede the remarkable achievements of lambda calculus, dependent type theory, compiler construction, or operational semantics. Rather, it suggests that these diverse traditions may be understood as complementary descriptions of a deeper process governing the evolution of bounded computational regions.

Whether this historical foundation ultimately proves advantageous for future programming languages, proof assistants, compiler architectures, distributed systems, or machine learning frameworks remains an empirical question. The purpose of this work has not been to establish that Spherepop is the unique foundation of computation, but to demonstrate that a coherent and mathematically disciplined alternative exists—one that begins not with abstract symbolic rewriting, but with the simple computational intuition learned in elementary arithmetic. Every computation begins by identifying a bounded region, resolving its interior, and continuing outward until no unresolved regions remain. Spherepop argues that this familiar procedure is not merely a pedagogical technique for evaluating arithmetic expressions, but a fundamental organizing principle capable of supporting a broad and unified theory of computation.

29 Design Philosophy and Historical Perspective

Every formal system begins by deciding what it considers fundamental. Euclid began with points and lines. Newton began with particles and forces. Church began with abstraction and application. Turing began with symbolic tapes and state transitions. These foundational choices determine not only the notation of a theory but the kinds of questions that become natural to ask. The Spherepop Calculus arises from the observation that much of contemporary computation continues to inherit an ontology developed during the earliest decades of theoretical computer science, when programs were naturally viewed as symbolic expressions manipulated according to syntactic rules. This perspective has been extraordinarily successful, yet it also encourages us to think of computation primarily as the transformation of static objects rather than the evolution of constructive processes.

The historical development of programming languages reflects this emphasis. Programs are written as text, parsed into syntax trees, transformed into intermediate representations, optimized through rewriting, and eventually executed as sequences of state transitions. Throughout this process, syntax occupies a privileged position. Even sophisticated semantic theories often begin with terms whose principal purpose is to be rewritten. The operational history leading from one expression to another is frequently treated as an auxiliary artifact useful for debugging, optimization, or explanation rather than as the central mathematical object itself.

Spherepop begins from a different observation. Long before programmers learn about lambda calculus or operational semantics, they already possess a correct algorithm for symbolic computation. Every child who evaluates arithmetic has learned to identify the innermost computational region, resolve it, replace it with its result, and continue until no unresolved regions remain. This procedure is not an approximation to computation. It is computation. The parentheses merely provide a notation for identifying computational scope. They are not themselves the fundamental objects being manipulated.

Once this distinction is recognized, an alternative ontology becomes possible. Instead of treating syntax as primary, Spherepop treats computational regions as primary. Parentheses become one possible notation for describing those regions. Abstract syntax trees become another. Dependency graphs become another. None of these representations is the computation itself. They are different descriptions of the same underlying historical construction. The computation

consists of the opening of computational regions, their internal evolution, and their eventual resolution through admissible Pops. History therefore precedes syntax rather than emerging as a by-product of execution.

This historical viewpoint also explains several design decisions that might otherwise appear unusual. Replay replaces substitution because historical construction is more informative than symbolic replacement. Refusal replaces late-stage failure because preventing incoherent historical extension is simpler than repairing corrupted computational states. Contexts become histories because assumptions are themselves constructive events whose order matters. Types become refusal structures because their essential role is not to classify completed values but to govern which computational continuations remain admissible. These choices are not independent innovations but different consequences of adopting history as the primary computational object.

Another guiding principle concerns commitment. Classical operational semantics often encourages immediate reduction whenever a reducible expression becomes available. Spherepop instead distinguishes between availability and commitment. A computational region may become admissible without requiring immediate Collapse. Histories may continue symbolically, probabilistically, or through Replay before an irreversible commitment is made. This distinction is particularly valuable in concurrent, distributed, and interactive systems, where delaying commitment often preserves opportunities for optimization, correction, negotiation, or additional information. Commitment therefore becomes an explicit semantic operation rather than an implicit consequence of evaluation.

Equally important is the deliberate separation between execution and interpretation. Contemporary computing frequently develops specialized execution engines for symbolic reasoning, numerical computation, fuzzy logic, probabilistic inference, differentiable programming, and machine learning. Although these systems differ mathematically, they often share remarkably similar dependency structures. Spherepop therefore proposes that execution should concern itself only with the evolution of computational histories, while logical interpretation should reside entirely within the operator library. The Universal Operator Machine embodies this philosophy by executing a single historical composition graph regardless of whether the operators represent Boolean conjunction, probabilistic sampling, tensor multiplication, or gradient propagation. Logic becomes a property of operators rather than of the runtime.

This philosophy also clarifies the relationship between Spherepop and exist-

ing compiler engineering. Nothing in this work argues that parsers, abstract syntax trees, control-flow graphs, Static Single Assignment, balanced search trees, dependency analysis, or graph rewriting should be abandoned. These techniques represent decades of accumulated engineering insight. Spherepop instead argues that they already manipulate richer mathematical objects than their traditional descriptions acknowledge. Syntax trees describe computational geography. Dependency graphs describe admissible continuation. Schedulers discover computational regions ready for Pop. Optimizers simplify historical construction while preserving observational behavior. The implementation techniques remain largely unchanged because they are already effective. The contribution of Spherepop is to reinterpret their common semantic foundation.

Perhaps the most significant philosophical consequence of this historical ontology is that explanation becomes intrinsic rather than external. Modern software systems devote substantial effort to provenance tracking, debugging, logging, execution tracing, version control, reproducibility, and incremental recompilation. These facilities are typically added after the computational model has already been defined. Spherepop instead incorporates historical construction directly into the semantics. Every observable value possesses a provenance. Every provenance may be reconstructed through Replay. Every historical extension satisfies explicit admissibility conditions. Explanation therefore ceases to be an auxiliary service layered upon execution. It becomes one of the defining characteristics of computation itself.

The broader ambition of the Spherepop Calculus is therefore neither to reject nor to supersede the established traditions of theoretical computer science. Lambda calculus, dependent type theory, process calculi, compiler theory, and proof assistants remain among the most profound achievements in the field. Spherepop instead proposes that these traditions may be viewed through a common historical lens. Computation is not fundamentally the rewriting of symbols, the transition between machine states, or the execution of isolated instructions. It is the irreversible construction of computational history through the continual opening, interaction, replay, refusal, and dissolution of bounded computational regions. Whether this perspective ultimately proves to be a more useful foundation for future programming languages and formal systems remains to be determined, but it offers a coherent conceptual framework within which many seemingly disparate aspects of computation appear as natural consequences of a single historical principle.

Appendices

A Formal Grammar of the Spherepop Calculus

Let

$$\mathcal{V} = \{x, y, z, \ldots\}$$

denote the countable set of variables,

$$\mathcal{C} = \{c_1, c_2, \ldots\}$$

the constants,
and

$$\mathcal{O} = \{\text{Sphere}, \text{Pop}, \text{Merge}, \text{Choice}, \text{Replay}, \text{Refuse}, \text{Collapse}, \text{Bind}\}$$

the primitive operators.
The grammar of terms is defined inductively by

$$\begin{aligned} t \;::=\; & x \\ \mid\; & c \\ \mid\; & \text{Sphere}(x : A.t) \\ \mid\; & \text{Pop}(t) \\ \mid\; & \text{Merge}(t, t) \\ \mid\; & \text{Choice}(p, t, t) \\ \mid\; & \text{Replay}(t, t) \\ \mid\; & \text{Refuse}(r) \\ \mid\; & \text{Collapse}(t) \\ \mid\; & \text{Bind}(t, t). \end{aligned}$$

Contexts are histories,

$$\Gamma ::= \emptyset \mid \Gamma, e,$$

where

$$e \in \{\text{Open}, \text{Pop}, \text{Replay}, \text{Merge}, \text{Choice}, \text{Collapse}, \text{Refuse}\}.$$

Histories satisfy

$$H ::= (e_1, e_2, \ldots, e_n).$$

Well-formed histories satisfy

$$e_i \prec e_j \Longrightarrow i < j,$$

where

$$\prec$$

is the dependency order.
A Sphere is recursively defined by

$$S = (B, I)$$

where

$$B$$

is a computational boundary and

$$I$$

is a finite computation graph.
Every interior node belongs to exactly one Sphere.
Nested Spheres satisfy

$$S_i \subset S_j \Longrightarrow \partial S_i \cap \partial S_j = \varnothing.$$

The parser therefore generates a rooted forest of computational regions whose transitive closure forms the dependency graph used throughout the operational semantics.

Every admissible program corresponds to a finite directed acyclic graph

$$G = (V, E)$$

equipped with a sphere assignment

$$\sigma : V \to \mathcal{S}$$

satisfying

$$(u, v) \in E \Longrightarrow \sigma(u) \subseteq \sigma(v) \vee \sigma(v) \subseteq \sigma(u) \vee \sigma(u) \parallel \sigma(v),$$

where

$$\parallel$$

denotes independent computational regions.

The operational scheduler acts only upon admissible sphere assignments.

The language therefore possesses both syntactic well-formedness and geometric well-formedness.

Formation Rules

$$\frac{}{\emptyset \ \mathsf{ctx}}$$

$$\frac{\Gamma \ \mathsf{ctx} \qquad x \notin \Gamma}{\Gamma, x : A \ \mathsf{ctx}}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{Sphere}(x : A.t)}$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash \mathsf{Pop}(t)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash \mathsf{Merge}(t, u)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash \mathsf{Choice}(p, t, u)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash \mathsf{Replay}(t, u)}$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash \mathsf{Collapse}(t)}$$

$$\frac{}{\Gamma \vdash \mathsf{Refuse}(r)}$$

Structural Axioms

$$H \subseteq H' \Longrightarrow |H| \leq |H'|$$

$$H \not\supset H'$$

$$\text{Replay}(H) = H$$

iff

$$H$$

contains no unresolved dependencies.

$$\text{Pop} \circ \text{Replay} = \text{Replay} \circ \text{Pop}$$

whenever

$$\text{Ready}(S).$$

Merge satisfies

$$(H_1 \otimes H_2) \otimes H_3 \cong H_1 \otimes (H_2 \otimes H_3).$$

$$H_1 \otimes H_2 \cong H_2 \otimes H_1.$$

Choice satisfies

$$\sum_i p_i = 1.$$

Replay preserves dependency,

$$u \prec v \Longrightarrow \text{Replay}(u) \prec \text{Replay}(v).$$

Pop removes exactly one computational boundary,

$$|\partial H'| = |\partial H| - 1.$$

Collapse is idempotent,

$$\text{Collapse}(\text{Collapse}(t)) = \text{Collapse}(t).$$

Refusal is absorbing,

$$\text{Replay}(\text{Refuse}(r), t) = \text{Refuse}(r).$$

These axioms define the formal syntax and structural constraints of the Spherepop Calculus independently of its operational and denotational interpretations.

B Static Semantics

The static semantics of the Spherepop Calculus is presented as a natural deduction system over historical contexts. Throughout this appendix, contexts are interpreted as finite histories of admissible computational construction rather than unordered environments.

Judgements

$$\Gamma \ \text{ctx}$$

$$\Gamma \vdash A : \text{Type}_i$$

$$\Gamma \vdash t : A$$

$$\Gamma \vdash H : \text{Hist}$$

$$\Gamma \vdash H \ \text{adm}$$

$$\Gamma \vdash H \rightsquigarrow H'$$

Universe Formation

$$\frac{}{\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}}$$

$$\text{Type}_0 < \text{Type}_1 < \text{Type}_2 < \cdots$$

Context Formation

$$\frac{}{\emptyset \text{ ctx}}$$

$$\frac{\Gamma \text{ ctx} \qquad \Gamma \vdash A : \text{Type}_i \qquad x \notin \Gamma}{\Gamma, x : A \text{ ctx}}$$

$$\frac{\Gamma \text{ ctx} \qquad \Gamma \vdash H : \text{Hist}}{\Gamma, H \text{ ctx}}$$

Variable Rule

$$\frac{\Gamma, x : A, \Delta \text{ ctx}}{\Gamma, x : A, \Delta \vdash x : A}$$

Weakening

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \text{Type}}{\Gamma, y : B \vdash t : A}$$

Historical Extension

$$\frac{\Gamma \vdash H : \text{Hist} \qquad \Gamma, H \vdash e}{\Gamma \vdash H :: e : \text{Hist}}$$

where

$$::$$

denotes historical concatenation.

Replay Formation

$$\frac{\Gamma \vdash H_1 : \text{Hist} \qquad \Gamma \vdash H_2 : \text{Hist}}{\Gamma \vdash \text{Replay}(H_1, H_2) : \text{Hist}}$$

Pop Formation

$$\frac{\Gamma \vdash S : \text{Sphere} \qquad \text{Ready}(S)}{\Gamma \vdash \text{Pop}(S)}$$

Merge Formation

$$\frac{\Gamma \vdash H_1 : \text{Hist} \qquad \Gamma \vdash H_2 : \text{Hist} \qquad H_1 \perp H_2}{\Gamma \vdash H_1 \otimes H_2 : \text{Hist}}$$

Choice Formation

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A \qquad 0 \leq p \leq 1}{\Gamma \vdash \text{Choice}(p, t, u) : A}$$

Collapse Formation

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \text{Collapse}(t) : A}$$

Refusal Formation

$$\frac{\neg\, \text{Adm}(H, e)}{\Gamma \vdash \text{Refuse}(e) : \text{Hist}}$$

Dependent Products

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi_{x:A} B : \text{Type}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi_{x:A} B}$$

$$\frac{\Gamma \vdash f : \Pi_{x:A} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}$$

Dependent Sums

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Sigma_{x:A} B : \text{Type}}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma_{x:A} B}$$

Identity Types

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{refl}(a) : \text{Id}_A(a, a)}$$

$$\frac{\Gamma \vdash p : \text{Id}_A(a, b)}{\Gamma \vdash J(p)}$$

Historical Admissibility

$$\frac{\Gamma \vdash H : \text{Hist} \qquad \text{Consistent}(H)}{\Gamma \vdash H \text{ adm}}$$

$$\frac{\Gamma \vdash H \text{ adm} \qquad \text{Adm}(H, e)}{\Gamma \vdash H :: e \text{ adm}}$$

Structural Properties

$$\Gamma \vdash H \Longrightarrow \Gamma \vdash \text{Replay}(H)$$

$$\Gamma \vdash H_1 \otimes H_2 \Longrightarrow \Gamma \vdash H_2 \otimes H_1$$

$$(H_1 \otimes H_2) \otimes H_3 \cong H_1 \otimes (H_2 \otimes H_3)$$

$$\text{Collapse} \circ \text{Collapse} = \text{Collapse}$$

$$\text{Replay} \circ \text{Replay} = \text{Replay}$$

$$\text{Replay} \circ \text{Collapse} = \text{Collapse} \circ \text{Replay}$$

$$H \subseteq H' \Longrightarrow |H| \leq |H'|$$

$$\Gamma \vdash H \text{ adm} \Longrightarrow \Gamma \vdash \text{Replay}(H) \text{ adm}$$

$$\Gamma \vdash H \text{ adm} \Longrightarrow \Gamma \vdash \text{Collapse}(H) \text{ adm}$$

Historical Substitution

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash t : B}{\Gamma, \Delta[a/x] \vdash \text{Replay}(a, t) : B[a/x]}$$

Fundamental Static Invariant

For every admissible derivation,

$$\Gamma \vdash H : \text{Hist},$$

there exists a unique dependency order

$$(H, \prec)$$

such that

$$\prec$$

is acyclic,

$$H$$

is monotone,
Replay preserves

$$\prec,$$

Pop removes exactly one maximal Sphere,
Merge preserves independent connected components,
Choice preserves well-typedness,
Collapse preserves admissibility,
and Refusal preserves consistency by preventing inadmissible historical extension.

C Operational Semantics

Let the machine configuration be

$$\langle H, Q, \Sigma \rangle$$

where

$$H$$

is the computational history,

$$Q$$

is the scheduler frontier,
and

$$\Sigma$$

is the dependency graph.

Transition Relation

The operational relation is

$$\longrightarrow \subseteq \mathcal{C} \times \mathcal{C}.$$

Configurations evolve according to

$$\langle H, Q, \Sigma \rangle \longrightarrow \langle H', Q', \Sigma' \rangle.$$

Ready Rule

$$\frac{\forall d \in \text{Deps}(S), \text{Resolved}(d)}{\text{Ready}(S)}$$

Sphere Introduction

$$\frac{}{\langle H, Q, \Sigma \rangle \longrightarrow \langle H :: \text{Open}(S), Q \cup \{S\}, \Sigma \rangle}$$

Pop

$$\frac{\text{Ready}(S)}{\langle H, Q, \Sigma \rangle \longrightarrow \langle H :: \text{Pop}(S), Q - \{S\}, \Sigma' \rangle}$$

Replay

$$\frac{u \prec v}{\text{Replay}(u, v)}$$

$$\frac{}{\langle H, Q, \Sigma \rangle \longrightarrow \langle H :: \text{Replay}, Q, \Sigma \rangle}$$

Merge

$$\frac{H_1 \perp H_2}{H_1 \otimes H_2}$$

$$\frac{S_1, \ldots, S_n \subseteq Q}{\text{Merge}(S_1, \ldots, S_n)}$$

Choice

Immediate semantics:

$$\frac{}{\text{Choice}(p, t, u) \longrightarrow t} \quad (p)$$

$$\frac{}{\text{Choice}(p, t, u) \longrightarrow u} \quad (1 - p)$$

Distribution semantics:

$$\text{Choice}(p, t, u) : \text{Dist}(A)$$

Collapse

$$\frac{\text{Resolved}(t)}{\text{Collapse}(t)}$$

Refusal

$$\frac{\neg\, \text{Adm}(H, e)}{\text{Refuse}(e)}$$

Bind

$$\frac{t \longrightarrow t'}{\text{Bind}(t, f) \longrightarrow \text{Bind}(t', f)}$$

$$\frac{}{\text{Bind}(v, f) \longrightarrow f(v)}$$

Congruence

$$\frac{t \longrightarrow t'}{\text{Pop}(t) \longrightarrow \text{Pop}(t')}$$

$$\frac{t \longrightarrow t'}{\text{Merge}(t, u) \longrightarrow \text{Merge}(t', u)}$$

$$\frac{u \longrightarrow u'}{\text{Merge}(t, u) \longrightarrow \text{Merge}(t, u')}$$

Scheduler

$$Q = \{S \mid \text{Ready}(S)\}.$$

$$\text{Select} : Q \to S.$$

$$\text{Execute}(S) = \text{Pop}(S).$$

Historical Transition

$$H \longrightarrow H :: e$$

where

$$e \in \{\text{Open}, \text{Replay}, \text{Pop}, \text{Merge}, \text{Choice}, \text{Collapse}, \text{Refuse}\}.$$

Operational Invariants

$$H \subseteq H'$$

$$|H| < |H'|$$

for every successful transition.

$$\text{Replay}(H) \supseteq H.$$

$$\text{Collapse}(H) \subseteq H.$$

$$\text{Refuse}(H) = H.$$

$$\text{Merge}(H_1, H_2) = H_1 \cup H_2.$$

Scheduler Correctness

$$\text{Ready}(S) \iff \forall d \in \text{Deps}(S), \text{Resolved}(d).$$

$$\text{Pop}(S) \Longrightarrow S \notin Q.$$

$$\text{Open}(S) \Longrightarrow S \in \Sigma.$$

Deterministic Fragment

If

Choice

does not occur, then

$$\forall H, \qquad H \longrightarrow H' \Longrightarrow H' \text{ unique.}$$

Monotonicity

$$H_i \subseteq H_{i+1}$$

for every execution sequence

$$H_0 \longrightarrow H_1 \longrightarrow \cdots \longrightarrow H_n.$$

Termination

If

$$G = (V, E)$$

is finite and acyclic,
then repeated application of

Ready

followed by

Pop

terminates after exactly

$$|V|$$

successful Pops.

Operational Soundness

For every derivation

$$\Gamma \vdash t : A$$

there exists an execution sequence

$$H_0 \longrightarrow H_1 \longrightarrow \cdots \longrightarrow H_n$$

such that

$$H_n$$

contains a unique maximal

$$\text{Collapse}(v)$$

with

$$\Gamma \vdash v : A.$$

This defines the complete small-step operational semantics of the Spherepop Calculus.

D Denotational Semantics

Let

$$\mathbf{Hist}$$

denote the category of admissible computational histories.
Objects are histories,

$$H \in \text{Obj}(\mathbf{Hist}),$$

and morphisms are admissible historical continuations,

$$f : H_1 \to H_2.$$

Identity

$$\mathrm{id}_H : H \to H.$$

$$\mathrm{id}_H \circ f = f.$$

$$f \circ \mathrm{id}_H = f.$$

Composition

$$f : H_1 \to H_2,$$

$$g : H_2 \to H_3,$$

$$g \circ f : H_1 \to H_3.$$

Associativity:

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Sphere Interpretation

$$S : H \to H_S.$$

$$H \subseteq H_S.$$

Pop Interpretation

$$\mathrm{Pop} : H_S \to H_S \cup \{v\}.$$

$$\mathrm{Pop} \circ S = v.$$

Replay Functor

$$\mathcal{R} : \mathbf{Hist} \to \mathbf{Hist}.$$

Object mapping:

$$\mathcal{R}(H) = H.$$

Morphisms:

$$\mathcal{R}(f) : \mathcal{R}(H_1) \to \mathcal{R}(H_2).$$

Functoriality:

$$\mathcal{R}(\mathrm{id}) = \mathrm{id}\,.$$

$$\mathcal{R}(g \circ f) = \mathcal{R}(g) \circ \mathcal{R}(f).$$

Merge

Tensor product:

$$\otimes : \mathbf{Hist} \times \mathbf{Hist} \to \mathbf{Hist}.$$

Associator:

$$(H_1 \otimes H_2) \otimes H_3 \cong H_1 \otimes (H_2 \otimes H_3).$$

Symmetry:

$$H_1 \otimes H_2 \cong H_2 \otimes H_1.$$

Unit:

$$I \otimes H \cong H.$$

Choice

Probability functor

$$\mathcal{D} : \mathbf{Hist} \to \mathbf{Hist}.$$

Unit:

$$\eta : H \to \mathcal{D}(H).$$

Multiplication:

$$\mu : \mathcal{D}^2(H) \to \mathcal{D}(H).$$

Monad laws:

$$\mu \circ \eta = \mathrm{id}\,.$$

$$\mu \circ \mathcal{D}(\eta) = \mathrm{id}\,.$$

$$\mu \circ \mathcal{D}(\mu) = \mu \circ \mu.$$

Refusal

Partial morphism

$$\mathrm{Refuse} : H \rightharpoonup H'.$$

Domain

$$\mathrm{Dom}(\mathrm{Refuse}) = \{H \mid \mathrm{Adm}(H)\}.$$

Undefined otherwise.

Collapse

Natural transformation

$$\kappa : \mathrm{id} \Rightarrow \mathrm{Collapse}\,.$$

Idempotence:

$$\kappa \circ \kappa = \kappa.$$

Historical Quotient

Define

$$H_1 \sim H_2$$

iff

$$\forall C, \qquad C[H_1] \Downarrow v \iff C[H_2] \Downarrow v.$$

The quotient category is

$$\mathbf{Hist}/\sim .$$

Replay Equivalence

$$\text{Replay}(H) \sim H.$$

$$\text{Replay}(\text{Replay}(H)) \sim \text{Replay}(H).$$

Historical Metric

Let

$$d : \mathbf{Hist} \times \mathbf{Hist} \to \mathbb{R}_{\geq 0}.$$

Properties:

$$d(H, H) = 0,$$

$$d(H_1, H_2) = d(H_2, H_1),$$

$$d(H_1, H_3) \leq d(H_1, H_2) + d(H_2, H_3).$$

Historical Completion

Every Cauchy sequence

$$H_0, H_1, \ldots$$

with respect to

$$d$$

admits a completion

$$\widehat{H}.$$

Universal Property

For every history-preserving functor

$$F : \mathbf{Hist} \to \mathcal{C},$$

there exists a unique factorization

$$\begin{array}{ccc} \mathbf{Hist} & \xrightarrow{Q} & \mathbf{Hist}/\!\sim \\ & \searrow_{F} & \downarrow^{\widetilde{F}} \\ & & \mathcal{C} \end{array}$$

such that

$$F = \widetilde{F} \circ Q.$$

Denotational Adequacy

If

$$H \Downarrow v$$

operationally,
then

$$H = v.$$

Conversely,

$$H = v \Longrightarrow H \Downarrow v.$$

Hence the operational semantics and denotational semantics are extensionally equivalent.

□

E Meta-Theoretic Proofs

Throughout this appendix let

$$\Gamma \vdash H : \text{Hist}$$

denote an admissible historical derivation and

$$H \longrightarrow H'$$

the operational transition relation.

Lemma A (Historical Extension)

If

$$H \longrightarrow H',$$

then

$$H \subseteq H'.$$

Proof. Operational inspection.

Every reduction appends exactly one admissible event.

No reduction removes an admitted event.

$$H' = H :: e.$$

Hence

$$H \subseteq H'.$$

□

Lemma B (Dependency Preservation)

If

$$u \prec v$$

in

$$H,$$

then

$$u \prec v$$

in every extension

$$H'.$$

Proof. Every operational rule preserves edge orientation.
No rule introduces a reverse dependency.
Therefore

$$\prec$$

remains acyclic.

□

Lemma C (Replay Preservation)

Replay preserves dependency order.

$$u \prec v \Longrightarrow \text{Replay}(u) \prec \text{Replay}(v).$$

Proof. Replay reconstructs existing historical structure.
No dependency is reordered.

□

Lemma D (Replay Idempotence)

$$\text{Replay}(\text{Replay}(H)) = \text{Replay}(H).$$

Proof. Replay reconstructs only unresolved dependencies.
After reconstruction no unresolved dependencies remain.
A second Replay introduces no new historical events.

□

Lemma E (Collapse Idempotence)

$$\text{Collapse}(\text{Collapse}(H)) = \text{Collapse}(H).$$

Proof. Collapse removes maximal unresolved boundaries.
Once collapsed, no additional collapse exists.

□

Lemma F (Merge Commutativity)

If

$$H_1 \perp H_2,$$

then

$$H_1 \otimes H_2 \cong H_2 \otimes H_1.$$

Proof. Independence implies absence of causal edges.
Execution order therefore does not alter reachable histories.

□

Lemma G (Merge Associativity)

$$(H_1 \otimes H_2) \otimes H_3 \cong H_1 \otimes (H_2 \otimes H_3).$$

Proof. Immediate from the associativity of disjoint union over dependency graphs.

□

Theorem 1 (Subject Preservation)

If

$$\Gamma \vdash t : A$$

and

$$t \longrightarrow t',$$

then

$$\Gamma \vdash t' : A.$$

Proof. Induction on the derivation of

$$t \longrightarrow t'.$$

Cases:

$$\text{Pop}, \text{Replay}, \text{Merge}, \text{Choice}, \text{Collapse}, \text{Bind}\ .$$

Typing follows from the corresponding formation rule.

□

Theorem 2 (Historical Preservation)

If

$$\Gamma \vdash H : \text{Hist}$$

and

$$H \longrightarrow H',$$

then

$$\Gamma \vdash H' : \text{Hist}.$$

Proof. Direct induction over operational transitions.

Apply Lemma A.

□

Theorem 3 (Historical Monotonicity)

Every execution satisfies

$$H_0 \subseteq H_1 \subseteq \cdots \subseteq H_n.$$

Proof. Repeated application of Lemma A.

□

Theorem 4 (Replay Correctness)

For every admissible continuation

$$C,$$

$$C[\text{Replay}(H)] \Downarrow v \iff C[H] \Downarrow v.$$

Proof. Replay reconstructs provenance without altering observable outputs.
Proceed by induction over the structure of

$$C.$$

□

Theorem 5 (Scheduler Correctness)

Every node selected by the scheduler satisfies

$$\text{Ready}(S).$$

Every Ready node eventually executes.

Proof. Ready follows from dependency inspection.
Termination follows from finiteness of the dependency graph.

□

Theorem 6 (Confluence of Independent Pops)

If

$$S_1 \perp S_2,$$

then

$$\mathrm{Pop}(S_1); \mathrm{Pop}(S_2) \cong \mathrm{Pop}(S_2); \mathrm{Pop}(S_1).$$

Proof. Immediate from Merge commutativity.

□

Theorem 7 (Progress)

If

$$\Gamma \vdash H : \mathrm{Hist},$$

then exactly one of the following holds.

(1) H is complete;

(2) $\exists\, H'$ such that $H \to H'$;

(3) H is deferred awaiting admissible continuation;

(4) H terminates by Refusal.

Proof. Induction on operational configurations.
Deferred configurations arise only from unresolved dependencies.
Refusal configurations terminate immediately.

□

Theorem 8 (Termination)

If

$$G = (V, E)$$

is finite and acyclic,
then execution terminates after

$$|V|$$

successful Pops.

Proof. Each Pop removes one maximal Sphere.
No Sphere is recreated.
Since

$$|V| < \infty,$$

termination follows by induction.

□

Theorem 9 (Historical Completeness)

Every observable value

$$v$$

admits a finite provenance.
That is,

$$\exists H$$

such that

$$H \Downarrow v.$$

Proof. Construct the execution history recursively from the dependency graph.
Finiteness follows from acyclicity.

□

Corollary

The Spherepop Calculus satisfies

Subject Preservation,
Historical Preservation,
Replay Correctness,
Historical Monotonicity,
Scheduler Correctness,
Confluence of Independent Pops,
Termination over finite DAGs,
Historical Completeness.

□

F Complexity Theory

Let

$$G = (V, E)$$

denote the dependency graph generated by the parser, where

$$n = |V|, \qquad m = |E|.$$

Let

$$H = (e_1, \dots, e_k)$$

be the execution history.

Definition (Depth)

The depth of a computational history is

$$D(H) = \max_{v \in V} \ell(v),$$

where

$$\ell(v)$$

is the length of the longest dependency path terminating at

$$v.$$

Definition (Width)

The width is

$$W(H) = \max_t |Q_t|,$$

where

$$Q_t$$

is the scheduler frontier at time

$$t.$$

Definition (Replay Radius)

The replay radius of a node

$$v$$

is

$$R(v) = |\operatorname{Anc}(v)|,$$

where

$$\operatorname{Anc}(v)$$

is the transitive closure of dependencies.

Definition (Historical Volume)

The historical volume is

$$\operatorname{Vol}(H) = |V| + |E| + |H|.$$

Definition (Collapse Depth)

$$C(H) = \max_i \operatorname{depth}(S_i).$$

Definition (Merge Width)

For

$$M = H_1 \otimes \cdots \otimes H_k,$$

define

$$W(M) = k.$$

Parser Complexity

Lexical analysis

$$O(n).$$

Parsing

$$O(n).$$

AST construction

$$O(n).$$

Dependency graph construction

$$O(n + m).$$

Scheduler Complexity

Naive scheduler

$$O(n)$$

per iteration.
Priority scheduler

$$O(\log n)$$

per insertion.
Heap scheduler

$$O(\log n)$$

per Pop.
Total execution

$$O(n \log n).$$

Replay Complexity

Worst case

$$O(n + m).$$

Bounded replay

$$O(R(v)).$$

Incremental replay

$$O(\Delta H).$$

Merge Complexity

Flattening

$$O(k).$$

Dependency merge

$$O(m).$$

Parallel scheduling

$$O(W(H)).$$

Choice Complexity

Immediate sampling

$$O(1).$$

Distribution propagation

$$O(k),$$

where

$$k$$

is the support size.

History Compression

Suppose repeated subhistories

$$H_i = H_j.$$

Compression factor

$$\gamma = \frac{|H|}{|H_c|}.$$

Compressed storage

$$|H_c| = O(r + d),$$

where

$$r$$

is the number of distinct replay regions and

$$d$$

the dependency graph.

Persistent Storage

Naive storage

$$O(|H|).$$

Checkpoint storage

$$O(c + d).$$

Replay reconstruction

$$O(R(v)).$$

Memory Complexity

Runtime memory

$$M = M_V + M_E + M_H.$$

Explicitly,

$$M = O(n + m + |H|).$$

Compressed,

$$M = O(n + m + r).$$

Parallel Speedup

Maximum theoretical speedup

$$S = \frac{n}{D(H)}.$$

Efficiency

$$E = \frac{S}{P},$$

where

$$P$$

is the processor count.

Critical Path

Execution time

$$T = D(H) + \frac{n - D(H)}{P}.$$

Historical Diameter

$$\operatorname{diam}(H) = \max_{u,v} d(u, v).$$

Replay Locality

Local replay satisfies

$$R(v) \ll n.$$

Global replay satisfies

$$R(v) = O(n).$$

Incremental Compilation

Suppose

$$\Delta V \subseteq V.$$

Incremental recompilation cost

$$O(|\Delta V| + |\Delta E|).$$

Historical Stability

Let

$$\delta$$

be the number of modified nodes.
Then

$$\text{Replay} = O(\delta + D(H)).$$

Asymptotic Bounds

Parsing	$O(n)$
Dependency Graph	$O(n + m)$
Scheduling	$O(n \log n)$
Replay	$O(R(v))$
Merge	$O(m)$
Choice	$O(1)$ or $O(k)$
Compression	$O(\|H\|)$
Incremental Compilation	$O(\Delta V + \Delta E)$
Memory	$O(n + m + r)$

Theorem (Historical Complexity)

Let

$$G$$

be a finite acyclic dependency graph.
Then the Spherepop runtime executes with total complexity

$$O(n \log n + m + R_{\text{max}}),$$

where

$$R_{\text{max}} = \max_{v} R(v).$$

Furthermore,

$$R_{\text{max}} \leq n,$$

and equality holds if and only if the dependency graph is totally ordered.

□

G Universal Operator Machine

Let

$$\mathcal{U} = (Q, \Sigma, H, \Omega)$$

denote the Universal Operator Machine.
The machine state consists of

$$Q$$

the scheduler,

$$\Sigma$$

the dependency graph,

$$H$$

the computational history,
and

$$\Omega$$

the operator library.

Machine Configuration

A machine configuration is

$$\langle Q, \Sigma, H, \Omega \rangle.$$

Execution is defined by

$$\longrightarrow_U \subseteq \mathcal{U} \times \mathcal{U}.$$

Operator Node

Every node is a tuple

$$N = (o, I, O, H),$$

where

$$o$$

is the operator,

$$I$$

its input list,

$$O$$

its output,
and

$$H$$

its local history.

Dependency Graph

$$\Sigma = (V, E).$$

Nodes

$$V = \{N_1, \ldots, N_n\}.$$

Edges

$$(u, v) \in E$$

represent computational dependency.
Acyclicity:

$$\Sigma \text{ is a DAG.}$$

Ready Set

$$Q = \{v \in V \mid \forall u \prec v, \text{Resolved}(u)\}.$$

Scheduler

Scheduler map

$$S : Q \to V.$$

Execution

$$S(v) = \mathrm{Pop}(v).$$

History Register

History memory

$$H = (e_1, \ldots, e_n).$$

Insertion

$$H' = H :: e.$$

Monotonicity

$$H \subseteq H'.$$

Replay Cache

Replay table

$$R : V \to H.$$

Lookup

$$R(v) = H_v.$$

Update

$$R' = R \cup \{(v, H')\}.$$

Operator Evaluation

Evaluation map

$$\Omega : o \times I \to O.$$

Composition

$$o_2 \circ o_1.$$

Identity

$$\mathrm{id}\,.$$

Execution Cycle

One machine cycle consists of

$$\text{Locate} \to \text{Replay} \to \text{Evaluate} \to \text{Pop} \to \text{Record} \to \text{Repeat}\,.$$

Machine Transition

$$\langle Q, \Sigma, H, \Omega \rangle \longrightarrow \langle Q', \Sigma', H', \Omega \rangle.$$

Operator Libraries

Boolean library

$$\Omega_B = \{\wedge, \vee, \neg\}.$$

Fuzzy library

$$\Omega_F = \{\min, \max, 1 - x\}.$$

Product logic

$$\Omega_P = \{xy, x + y - xy, 1 - x\}.$$

Differentiable library

$$\Omega_D = \{+, \times, \sigma, \tanh, \mathrm{ReLU}\}.$$

Probabilistic library

$$\Omega_{\mathcal{D}} = \{\text{Choice}, \text{Sample}, \text{Observe}\}.$$

Compilation

Compiler

$$C : \text{Program} \to \Sigma.$$

Optimization

$$O : \Sigma \to \Sigma'.$$

Code generation

$$G : \Sigma' \to B,$$

where

$$B$$

is machine bytecode.

Optimization Rules

Inlining

$$f(x) \rightsquigarrow \text{Body}(f).$$

Constant propagation

$$x = c \Longrightarrow f(x) = f(c).$$

Dead history elimination

$$\text{Future}(H) = \varnothing \Longrightarrow H \mapsto \varepsilon.$$

Replay compression

$$H_i = H_j \Longrightarrow H_i \equiv H_j.$$

Merge flattening

$$(H_1 \otimes H_2) \otimes H_3 \Longrightarrow H_1 \otimes H_2 \otimes H_3.$$

Historical Memory

Active memory

$$M_A.$$

Replay cache

$$M_R.$$

Archive

$$M_H.$$

Total memory

$$M = M_A + M_R + M_H.$$

Correctness

Machine correctness

$$C(P) \Downarrow v \Longleftrightarrow P \Downarrow v.$$

Replay correctness

$$\text{Replay}(H) \sim H.$$

Scheduler correctness

$$\forall v \in Q, \qquad \text{Ready}(v).$$

Universal Simulation Theorem

Let

$$\Omega$$

be any operator algebra satisfying closure under composition.

Then there exists a Universal Operator Machine

$$\mathcal{U}_\Omega$$

such that every finite computation over

$$\Omega$$

admits an execution sequence

$$\mathcal{U}_\Omega \Downarrow H$$

whose terminal Collapse is observationally equivalent to the original computation.

Equivalently,

$$\forall P_\Omega, \qquad \exists \mathcal{U}_\Omega, \qquad P_\Omega = \mathcal{U}_\Omega.$$

Proof. Construct the dependency graph generated by P_Ω.

Replace each primitive operation by its corresponding operator node.

Preserve dependency edges.

The scheduler executes every Ready node according to the operational semantics.

Replay preserves provenance.

Pop resolves completed computational regions.

Collapse produces the terminal observable value.

Correctness follows by induction over the topological ordering of the dependency graph.

□

□

H Equidistribution of Historical Lattice Shapes

Let

$$H$$

be a finite admissible history.
Let

$$\Lambda(H) \subset \mathbb{R}^d$$

be the dependency lattice generated by the incidence vectors of events in

$$H.$$

Define the covolume-normalized lattice

$$\Lambda_1(H) = \mathrm{covol}(\Lambda(H))^{-1/d}\Lambda(H).$$

The shape of

$$H$$

is

$$\mathrm{sh}(H) = [\Lambda_1(H)] \in \mathrm{SL}_d(\mathbb{Z})\backslash \mathrm{SL}_d(\mathbb{R})/\mathrm{SO}_d(\mathbb{R}).$$

Let

$$\mathcal{X}_d = \mathrm{SL}_d(\mathbb{Z})\backslash \mathrm{SL}_d(\mathbb{R})/\mathrm{SO}_d(\mathbb{R}).$$

Let

$$\mu_d$$

be the Haar probability measure on

$$\mathcal{X}_d.$$

For

$$X > 0,$$

define

$$\mathcal{H}_d(X) = \{H : \mathrm{rank}\,\Lambda(H) = d,\ \mathrm{Disc}(H) < X\}.$$

Here

$$\mathrm{Disc}(H) = \det(\langle e_i, e_j \rangle)$$

for any basis

$$(e_1, \dots, e_d)$$

of

$$\Lambda(H).$$

Definition

A family of admissible histories

$$\mathcal{H}_d$$

has equidistributed lattice shapes if, for every bounded continuous function

$$f : \mathcal{X}_d \to \mathbb{R},$$

one has

$$\lim_{X \to \infty} \frac{1}{|\mathcal{H}_d(X)|} \sum_{H \in \mathcal{H}_d(X)} f(\mathrm{sh}(H)) = \int_{\mathcal{X}_d} f \, d\mu_d.$$

Historical Shape Conjecture

For

$$d \in \{3, 4, 5\},$$

the admissible Spherepop histories of dependency rank

$$d$$

ordered by discriminant have equidistributed normalized dependency-lattice shapes:

$$\mathrm{sh}(\mathcal{H}_d(X)) \Rightarrow \mu_d$$

as

$$X \to \infty.$$

Equivalently,

$$\lim_{X \to \infty} \frac{|\{H \in \mathcal{H}_d(X) : \mathrm{sh}(H) \in U\}|}{|\mathcal{H}_d(X)|} = \mu_d(U)$$

for every

$$\mu_d(\partial U) = 0.$$

Analogy

For a number field

$$K/\mathbb{Q}$$

of degree

$$n,$$

the ring of integers

$$\mathcal{O}_K$$

embeds as a lattice in

$$K \otimes_{\mathbb{Q}} \mathbb{R}.$$

After removing scale, one obtains a lattice shape in

$$\mathcal{X}_{n-1}.$$

For cubic, quartic, and quintic fields, ordered by discriminant, these shapes are expected or known in suitable settings to distribute according to the natural homogeneous measure.

Spherepop replaces

$$\mathcal{O}_K$$

by

$$\Lambda(H),$$

and replaces arithmetic discriminant by historical discriminant.

Thus

$$\mathcal{O}_K \rightsquigarrow \Lambda(H),$$

$$\mathrm{Disc}(K) \rightsquigarrow \mathrm{Disc}(H),$$

$$\mathrm{sh}(\mathcal{O}_K) \rightsquigarrow \mathrm{sh}(H).$$

Interpretation

Historical equidistribution asserts that large admissible histories do not concentrate around special dependency geometries.

In the limit,

$$X \to \infty,$$

their normalized dependency lattices become generic points of the moduli space

$$\mathcal{X}_d.$$

Thus the large-scale geometry of computation becomes asymptotically independent of accidental syntactic presentation.

Spherepop Equidistribution Principle

Let

$$F$$

be any admissible construction functor satisfying

$$F(H_1 \otimes H_2) = F(H_1) \oplus F(H_2),$$

$$F(\text{Replay}(H)) = F(H),$$

and

$$F(\text{Collapse}(H)) = F(H)/\sim.$$

Then the induced lattice-shape map

$$H \mapsto \text{sh}(F(H))$$

is asymptotically Haar-distributed whenever the discriminant ordering is non-degenerate.

$$\boxed{\text{sh}(H) \sim \mu_d}$$

for

$$d = 3, 4, 5.$$

I Limits, Hyperplanes, and Logic-Circuit Diagrams

Let

$$C = (V, E)$$

be a finite directed acyclic circuit.
Let

$$V_{\text{in}} \subseteq V$$

be input nodes,

$$V_{\text{gate}} \subseteq V$$

gate nodes,
and

$$V_{\text{out}} \subseteq V$$

output nodes.

Each gate

$$g \in V_{\text{gate}}$$

is assigned an operator

$$\omega_g : [0,1]^{k_g} \to [0,1].$$

The circuit evaluation map is

$$F_C : [0,1]^n \to [0,1]^m.$$

Hyperplane Representation

A threshold gate is a hyperplane decision rule

$$g(x) = \mathbf{1}\left[w \cdot x + b \geq 0\right],$$

with decision hyperplane

$$\Pi_g = \{x \in \mathbb{R}^n : w \cdot x + b = 0\}.$$

The positive and negative regions are

$$\Pi_g^+ = \{x : w \cdot x + b > 0\}, \qquad \Pi_g^- = \{x : w \cdot x + b < 0\}.$$

Thus

$$\mathbb{R}^n = \Pi_g^- \cup \Pi_g \cup \Pi_g^+.$$

Logical Boundaries

A Boolean predicate

$$P : X \to \{0,1\}$$

induces a boundary

$$\partial P = \overline{P^{-1}(1)} \cap \overline{P^{-1}(0)}.$$

For a threshold predicate,

$$\partial P = \Pi_g.$$

Therefore every threshold gate is a refusal boundary:

$$x \in \Pi_g^+ \Longrightarrow \text{Collapse}(1),$$

$$x \in \Pi_g^- \Longrightarrow \text{Collapse}(0),$$

$$x \in \Pi_g \Longrightarrow \text{Defer} \ \vee \ \text{Refuse}\,.$$

Limits of Soft Gates

Let

$$\sigma_\beta(t) = \frac{1}{1 + e^{-\beta t}}.$$

Define

$$g_\beta(x) = \sigma_\beta(w \cdot x + b).$$

Then

$$\lim_{\beta \to \infty} g_\beta(x) = \mathbf{1}[w \cdot x + b > 0]$$

for

$$x \notin \Pi_g.$$

At the boundary,

$$x \in \Pi_g \Longrightarrow g_\beta(x) = \frac{1}{2}.$$

Hence

$$\lim_{\beta \to \infty} g_\beta = g$$

pointwise off the hyperplane.
Distributionally,

$$\nabla g_\beta = \beta \sigma_\beta (1 - \sigma_\beta) w$$

converges to a boundary-supported measure:

$$\nabla g_\beta \rightharpoonup w \, \delta_{\Pi_g}.$$

Circuit Regions

For gates

$$g_1, \ldots, g_N,$$

define hyperplanes

$$\Pi_i = \{x : w_i \cdot x + b_i = 0\}.$$

The arrangement

$$\mathcal{A}_C = \{\Pi_1, \ldots, \Pi_N\}$$

partitions

$$\mathbb{R}^n$$

into cells

$$\mathbb{R}^n \setminus \bigcup_i \Pi_i = \bigsqcup_\alpha R_\alpha.$$

On each cell,

$$F_C$$

is locally constant for Boolean threshold circuits.
Thus

$$x, y \in R_\alpha \Longrightarrow F_C(x) = F_C(y).$$

Spherepop Interpretation

Each gate is a Sphere

$$S_g.$$

Its hyperplane

$$\Pi_g$$

is the admissibility boundary.
Evaluation is

$$\mathrm{Pop}(S_g) = \begin{cases} 1, & w \cdot x + b > 0, \\ 0, & w \cdot x + b < 0, \\ \mathrm{Refuse}(r_g), & w \cdot x + b = 0. \end{cases}$$

A circuit is a Merge of gate histories:

$$H_C = \bigotimes_{g \in V_{\text{gate}}} H_g.$$

Dependency edges impose partial order:

$$g_i \prec g_j \iff (g_i, g_j) \in E.$$

Circuit evaluation is historical Collapse:

$$F_C(x) = \mathrm{Collapse}\left(\bigotimes_{g \in V_{\text{out}}} \mathrm{Pop}(S_g) \right).$$

Diagrammatic Laws

Serial composition:

$$x \longrightarrow g \longrightarrow h \longrightarrow y$$

denotes

$$h \circ g.$$

Parallel composition:

$$\begin{aligned} x &\longrightarrow g \\ x &\longrightarrow h \end{aligned}$$

denotes

$$g \otimes h.$$

Fan-in:

$$(g, h) \longrightarrow k$$

denotes

$$k \circ (g \otimes h).$$

Fan-out denotes Replay:

$$x \longmapsto (x, x) = \text{Replay}(x).$$

Limit of Circuit Diagrams

Let

$$C_\beta$$

be a differentiable circuit obtained by replacing every threshold gate

$$g_i$$

with

$$g_{i,\beta}(x) = \sigma_\beta(w_i \cdot x + b_i).$$

Then

$$F_{C_\beta} : [0,1]^n \to [0,1]^m$$

is smooth.
For every

$$x \notin \bigcup_i \Pi_i,$$

$$\lim_{\beta \to \infty} F_{C_\beta}(x) = F_C(x).$$

Thus Boolean circuit diagrams are sharp-boundary limits of differentiable Spherepop histories.

Hyperplane Limit Theorem

Let

$$C_\beta$$

be a family of smooth circuits with gate activations

$$\sigma_\beta(w_i \cdot x + b_i).$$

Let

$$C_\infty$$

be the threshold circuit obtained as

$$\beta \to \infty.$$

Then

$$F_{C_\beta} \to F_{C_\infty}$$

pointwise on

$$\mathbb{R}^n \setminus \bigcup_i \Pi_i.$$

Moreover,

$$\nabla F_{C_\beta}$$

converges weakly to a measure supported on

$$\bigcup_i \Pi_i.$$

Hence all logical discontinuities are concentrated on admissibility hyperplanes.

□

Circuit Diagram Equivalence

Two circuits

$$C_1, C_2$$

are diagrammatically equivalent if

$$F_{C_1} = F_{C_2}.$$

They are historically equivalent if

$$H_{C_1} \sim H_{C_2}.$$

Historical equivalence implies diagrammatic equivalence:

$$H_{C_1} \sim H_{C_2} \Longrightarrow F_{C_1} = F_{C_2}.$$

The converse fails whenever two diagrams compute the same function with distinct dependency histories.

Collapse Quotient

Define

$$C_1 \equiv_{\text{collapse}} C_2$$

iff

$$\text{Collapse}(H_{C_1}) = \text{Collapse}(H_{C_2}).$$

Then

$$\equiv_{\text{hist}} \subseteq \equiv_{\text{collapse}} \subseteq \equiv_{\text{func}} .$$

Thus

$$\text{history} \to \text{diagram} \to \text{function}$$

is a sequence of quotient maps.

J Historical Algebra

Let

$$\mathcal{H}$$

denote the collection of all admissible histories.

Definition (History Addition)

Define

$$\oplus : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$$

by

$$H_1 \oplus H_2 = \text{Merge}(H_1, H_2).$$

Properties:

$$H_1 \oplus H_2 = H_2 \oplus H_1.$$

$$(H_1 \oplus H_2) \oplus H_3 = H_1 \oplus (H_2 \oplus H_3).$$

Identity:

$$0_{\mathcal{H}} = \varnothing.$$

$$H \oplus 0_{\mathcal{H}} = H.$$

Definition (Historical Multiplication)

Define

$$\otimes : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$$

by sequential composition,

$$H_1 \otimes H_2 = H_1 :: H_2.$$

Associativity:

$$(H_1 \otimes H_2) \otimes H_3 = H_1 \otimes (H_2 \otimes H_3).$$

Identity:

$$1_{\mathcal{H}} = \varepsilon.$$

Distributivity

$$H_1 \otimes (H_2 \oplus H_3) = (H_1 \otimes H_2) \oplus (H_1 \otimes H_3).$$

Likewise,

$$(H_1 \oplus H_2) \otimes H_3 = (H_1 \otimes H_3) \oplus (H_2 \otimes H_3).$$

Historical Semiring

Therefore

$$(\mathcal{H}, \oplus, \otimes)$$

forms a semiring.

Replay Operator

Define

$$R : \mathcal{H} \to \mathcal{H}.$$

Idempotence:

$$R^2 = R.$$

Monotonicity:

$$H_1 \subseteq H_2 \Longrightarrow R(H_1) \subseteq R(H_2).$$

Composition preservation:

$$R(H_1 \otimes H_2) = R(H_1) \otimes R(H_2).$$

Collapse Operator

Define

$$C : \mathcal{H} \to \mathcal{V}.$$

Idempotence:

$$C^2 = C.$$

Compatibility:

$$C(R(H)) = C(H).$$

Refusal Algebra

Let

$$\mathcal{R} = \{r_1, r_2, \ldots\}$$

be refusal reasons.
Define

$$\vee$$

by least common refusal,
and

$$\wedge$$

by greatest common admissible refinement.
Then

$$(\mathcal{R}, \vee, \wedge)$$

is a bounded lattice.
Bottom element:

$$\bot = \text{Accept}\,.$$

Top element:

$$\top = \text{Impossible}\,.$$

Historical Congruence

Define

$$H_1 \equiv H_2$$

iff

$$C(H_1) = C(H_2).$$

Then

$$\equiv$$

is an equivalence relation.
Furthermore,

$$H_1 \equiv H_2 \Longrightarrow R(H_1) \equiv R(H_2).$$

Historical Ideals

A subset

$$I \subseteq \mathcal{H}$$

is an ideal if

$$H \in I, \qquad K \subseteq H \Longrightarrow K \in I,$$

and

$$H_1, H_2 \in I \Longrightarrow H_1 \oplus H_2 \in I.$$

Replay Closure

Define

$$\overline{H} = R(H).$$

Then

$$\overline{\overline{H}} = \overline{H}.$$

Historical Metric Algebra

Let

$$d : \mathcal{H} \times \mathcal{H} \to \mathbb{R}.$$

Require

$$d(H, H) = 0,$$

$$d(H_1, H_2) = d(H_2, H_1),$$

$$d(H_1, H_3) \leq d(H_1, H_2) + d(H_2, H_3).$$

Replay is non-expansive:

$$d(R(H_1), R(H_2)) \leq d(H_1, H_2).$$

Fixed Points

A history

$$H$$

is replay-complete iff

$$R(H) = H.$$

A history is collapse-complete iff

$$C(H) = H.$$

Least Fixed Point

Let

$$F : \mathcal{H} \to \mathcal{H}$$

be monotone.
Then

$$\mathrm{lfp}(F) = \bigcup_{n=0}^{\infty} F^n(\varnothing).$$

Greatest Fixed Point

If

$$F$$

preserves arbitrary intersections,
then

$$\mathrm{gfp}(F) = \bigcap \{H : F(H) \subseteq H\}.$$

Completion Theorem

Every ascending chain

$$H_0 \subseteq H_1 \subseteq \cdots$$

admits a least upper bound

$$H_\infty = \bigcup_i H_i.$$

Moreover,

$$R(H_\infty) = \bigcup_i R(H_i).$$

Historical Algebra Theorem

The structure

$$\boxed{(\mathcal{H}, \oplus, \otimes, R, C, d)}$$

forms a complete idempotent semiring equipped with

a replay closure operator,

a collapse quotient,

a bounded refusal lattice,

least and greatest fixed points,

a complete lattice of admissible histories.

Consequently, every finite Spherepop computation admits an algebraic interpretation independent of its operational realization.

□

K Category-Theoretic Foundations

Let

Hist

denote the category of admissible computational histories.
Objects are histories,

$$H \in \mathrm{Ob}(\mathbf{Hist}),$$

and morphisms are admissible historical continuations,

$$f : H_1 \to H_2.$$

Identity Morphisms

For every history

$$H,$$

there exists

$$\mathrm{id}_H : H \to H$$

such that

$$f \circ \mathrm{id}_H = f,$$

$$\mathrm{id}_H \circ g = g.$$

Composition

Given

$$f : H_1 \to H_2,$$

$$g : H_2 \to H_3,$$

define

$$g \circ f : H_1 \to H_3.$$

Associativity holds:

$$(h \circ g) \circ f = h \circ (g \circ f).$$

Historical Product

The categorical product

$$H_1 \times H_2$$

is equipped with projections

$$\pi_1 : H_1 \times H_2 \to H_1,$$

$$\pi_2 : H_1 \times H_2 \to H_2.$$

For every

$$X$$

and morphisms

$$f : X \to H_1,$$

$$g : X \to H_2,$$

there exists a unique morphism

$$\langle f, g \rangle : X \to H_1 \times H_2.$$

Historical Coproduct

The coproduct

$$H_1 + H_2$$

admits injections

$$\iota_1 : H_1 \to H_1 + H_2,$$

$$\iota_2 : H_2 \to H_1 + H_2.$$

Merge Tensor

Merge defines a symmetric monoidal product

$$\otimes : \mathbf{Hist} \times \mathbf{Hist} \to \mathbf{Hist}.$$

Associator

$$\alpha : (H_1 \otimes H_2) \otimes H_3 \Rightarrow H_1 \otimes (H_2 \otimes H_3).$$

Left unitor

$$\lambda : I \otimes H \Rightarrow H.$$

Right unitor

$$\rho : H \otimes I \Rightarrow H.$$

Braiding

$$\beta : H_1 \otimes H_2 \Rightarrow H_2 \otimes H_1.$$

Replay Functor

Replay is an endofunctor

$$R : \mathbf{Hist} \to \mathbf{Hist}.$$

Objects:

$$R(H) = H.$$

Morphisms:

$$R(f) : R(H_1) \to R(H_2).$$

Functoriality:

$$R(\mathrm{id}) = \mathrm{id},$$

$$R(g \circ f) = R(g) \circ R(f).$$

Collapse Natural Transformation

Collapse defines

$$\kappa : R \Rightarrow \mathrm{Id}\,.$$

Naturality:

$$\begin{array}{ccc} R(H_1) & \xrightarrow{R(f)} & R(H_2) \\ \downarrow \kappa & & \downarrow \kappa \\ H_1 & \xrightarrow{f} & H_2 \end{array}$$

commutes.

Refusal Subcategory

Define

$$\mathbf{Adm} \subseteq \mathbf{Hist}$$

by

$$\mathrm{Ob}(\mathbf{Adm}) = \{H : \mathrm{Adm}(H)\}.$$

Morphisms preserve admissibility.

Adjunction

Let

$$I : \mathbf{Adm} \hookrightarrow \mathbf{Hist}$$

be inclusion.
Suppose

$$L : \mathbf{Hist} \to \mathbf{Adm}$$

assigns maximal admissible repair.
Then

$$L \dashv I.$$

Hence

$$\mathrm{Hom}_{\mathbf{Adm}}(LH, A) \cong \mathrm{Hom}_{\mathbf{Hist}}(H, IA).$$

Historical Monad

Replay induces a monad

$$(R, \eta, \mu).$$

Unit

$$\eta : \mathrm{Id} \Rightarrow R.$$

Multiplication

$$\mu : R^2 \Rightarrow R.$$

Monad identities:

$$\mu \circ R\eta = \mathrm{id},$$

$$\mu \circ \eta R = \mathrm{id},$$

$$\mu \circ R\mu = \mu \circ \mu R.$$

Historical Comonad

History extraction defines

$$(E, \varepsilon, \delta).$$

Counit

$$\varepsilon : E \Rightarrow \mathrm{Id}\,.$$

Comultiplication

$$\delta : E \Rightarrow E^2.$$

Pullbacks

For

$$f : H_1 \to H_3,$$

$$g : H_2 \to H_3,$$

their pullback is

$$H_1 \times_{H_3} H_2.$$

Operationally this represents dependency synchronization.

Pushouts

Given

$$H_0 \to H_1,$$

$$H_0 \to H_2,$$

the pushout

$$H_1 \cup_{H_0} H_2$$

models history merging.

Limits

Every finite diagram

$$D : J \to \mathbf{Hist}$$

admits a limit

$$\varprojlim D$$

whenever all dependency cones commute.

Colimits

Every finite cocomplete diagram admits

$$\varinjlim D.$$

Operationally,

$$\varinjlim$$

corresponds to history accumulation.

Kan Extensions

Given

$$F : \mathbf{C} \to \mathbf{Hist},$$

$$K : \mathbf{C} \to \mathbf{D},$$

the left Kan extension

$$\mathrm{Lan}_K(F)$$

exists whenever

Hist

is cocomplete.
Likewise,

$$\mathrm{Ran}_K(F)$$

exists whenever

Hist

is complete.

Presheaf Semantics

Let

Sphere

denote the category of computational regions.
A historical presheaf is

$$F : \mathbf{Sphere}^{op} \to \mathbf{Set}.$$

For every inclusion

$$U \subseteq V,$$

there is a restriction morphism

$$F(V) \to F(U).$$

Sheaf Condition

If

$$\{U_i\}$$

covers

$$U,$$

and

$$s_i \in F(U_i)$$

agree on overlaps,
then there exists a unique

$$s \in F(U)$$

such that

$$s|_{U_i} = s_i.$$

Historical Fibration

Define

$$p : \textbf{Hist} \to \textbf{Time}.$$

Fibres

$$p^{-1}(t)$$

contain all admissible histories terminating at logical time

$$t.$$

Categorical Completeness Theorem

The category

Hist

is

complete,
cocomplete,
symmetric monoidal,
cartesian closed on deterministic fragments,
equipped with replay monads,
history comonads,
finite limits and colimits,
presheaf semantics over computational spheres.

Furthermore, every finite Spherepop computation is the image of a commuting diagram in

Hist,

and every operational execution corresponds to a natural transformation between history-valued functors.

□

L Geometric Semantics

Let

$$P$$

be a smooth manifold representing the computational plenum.

Every computational object is realized as a compact embedded submanifold

$$S \subseteq P.$$

Execution is interpreted as the geometric evolution of nested computational regions within

$$P.$$

Sphere Embedding

A computational Sphere is an embedding

$$i_S : S \hookrightarrow P.$$

Its boundary is

$$\partial S.$$

The interior is

$$S^\circ = S \setminus \partial S.$$

Nested Spheres satisfy

$$S_1 \subset S_2 \Longrightarrow \partial S_1 \cap \partial S_2 = \varnothing.$$

Boundary Orientation

Every boundary possesses an outward unit normal field

$$n : \partial S \to TP.$$

Boundary orientation induces the evaluation direction

$$n(x) \in T_x P.$$

Pop Flow

Let

$$\Phi : P \to \mathbb{R}$$

be the computational potential.

The Pop derivative is

$$\partial_P \Phi = (\nabla \cdot n)\, \partial_r \Phi + v \cdot \nabla \Phi.$$

Here

$$v$$

is the computational flow field.

Minimal Surface Law

If

$$\Delta \Phi = 0,$$

and

$$\nabla \cdot v = 0,$$

then

$$\partial_P \Phi = H\, \partial_r \Phi,$$

where

$$H$$

is the mean curvature of

$$\partial S.$$

Curvature

The second fundamental form is

$$II(X, Y) = \langle \nabla_X n, Y \rangle.$$

Principal curvatures are

$$\kappa_1, \ldots, \kappa_{d-1}.$$

Mean curvature

$$H = \frac{1}{d-1} \sum_i \kappa_i.$$

Gaussian curvature

$$K = \prod_i \kappa_i.$$

Geodesic Replay

Replay follows shortest admissible dependency paths.

Let

$$\gamma : [0,1] \to P.$$

Then

$$\nabla_{\dot\gamma} \dot\gamma = 0.$$

Replay minimizes

$$L(\gamma) = \int_0^1 \|\dot\gamma\| \, dt.$$

Dependency Foliation

Let

$$\mathcal{F} = \{L_\alpha\}$$

be the foliation generated by dependency classes.

Each leaf

$$L_\alpha$$

contains histories sharing identical causal ancestry.

Leaves satisfy

$$P = \bigcup_\alpha L_\alpha.$$

Historical Fibres

Define

$$\pi : P \to B.$$

The fibre

$$\pi^{-1}(b)$$

contains all histories representing equivalent observable computations.

Hyperplane Boundaries

Logical predicates define hypersurfaces

$$\Pi = \{x : f(x) = 0\}.$$

Positive region

$$f > 0.$$

Negative region

$$f < 0.$$

Boundary

$$f = 0.$$

Pop occurs only after crossing

$$\Pi.$$

Collapse Surface

The Collapse operator induces

$$C : S \to \partial S.$$

Repeated Collapse satisfies

$$C^2 = C.$$

Merge Geometry

Independent histories form disjoint unions

$$S_1 \sqcup S_2.$$

Shared boundaries induce gluing

$$S_1 \cup_\partial S_2.$$

Choice Geometry

Choice defines a probability density

$$\rho : P \to [0, 1].$$

Normalization

$$\int_P \rho \, dV = 1.$$

Energy Functional

Define

$$E(S) = \int_S \left(\alpha + \beta H^2 + \gamma \|\nabla \Phi\|^2 \right) dV.$$

Admissible execution minimizes

$$E.$$

Historical Action

Let

$$H(t)$$

be a one-parameter family of histories.

Define

$$\mathcal{A}(H) = \int L(H, \dot{H})\, dt.$$

Stationary histories satisfy

$$\delta\mathcal{A} = 0.$$

Reachability Volume

The reachable region of a history is

$$\mathcal{R}(H) \subseteq P.$$

Volume

$$V(H) = \text{Vol}(\mathcal{R}(H)).$$

Monotonicity

$$H_1 \subseteq H_2 \Longrightarrow V(H_1) \leq V(H_2).$$

Historical Divergence

Define

$$\text{Div}(H) = \nabla \cdot v.$$

If

$$\text{Div}(H) = 0,$$

history preserves computational volume.

Boundary Collapse Theorem

Let

$$S \subseteq P$$

be an admissible computational Sphere.

Suppose

$$\Delta\Phi = 0, \qquad \nabla \cdot v = 0,$$

and

$$H > 0$$

everywhere on

$$\partial S.$$

Then repeated Pop evolution converges to a unique terminal Collapse

$$C(S),$$

independent of the local parametrization of

$$S.$$

Geometric Completeness

Every finite Spherepop computation admits a geometric realization

$$(\mathcal{M}, g, \mathcal{F}, v, \Phi),$$

where

$$\mathcal{M}$$

is a smooth manifold,

$$g$$

a Riemannian metric,

$$\mathcal{F}$$

the dependency foliation,

$$v$$

the execution vector field,
and

$$\Phi$$

the computational potential.
Operational semantics correspond to boundary evolution,
Replay corresponds to geodesic reconstruction,
Merge corresponds to manifold gluing,
Choice corresponds to probability densities,
Refusal corresponds to forbidden boundary crossings,
and Collapse corresponds to convergence onto admissible boundary strata.

Consequently, every admissible computation possesses both an operational interpretation as a history and a geometric interpretation as the evolution of nested computational regions within the computational plenum.

□

M Reference Grammar and Concrete Syntax

Lexical Alphabet

Let

$$\Sigma = \Sigma_L \cup \Sigma_D \cup \Sigma_S \cup \Sigma_O \cup \Sigma_W,$$

where

$$\Sigma_L = \{A, \ldots, Z, a, \ldots, z, \_\},$$

$$\Sigma_D = \{0, \ldots, 9\},$$

$$\Sigma_S = \{(,), [,], \{,\}, <, >, :, ;, ,\},$$

$$\Sigma_O = \{+, -, *, /, =, \to, \Leftarrow, :=, ::, \otimes, \oplus\},$$

and

$$\Sigma_W = \{\texttt{space}, \texttt{tab}, \texttt{newline}\}.$$

Identifiers

Identifier

$$I ::= L(L \mid D)^*.$$

Formally,

$$I = \{x \in \Sigma^* : x_1 \in \Sigma_L,\ x_i \in \Sigma_L \cup \Sigma_D\}.$$

Reserved Keywords

sphere, **pop**, **merge**, **choice**, **collapse**, **replay**, **refuse**, **bind**, **module**, **import**, **export**, **let**, **match**, **history**, **type**, **proof**, **true**, **false**, **if**, **then**, **else**, **while**, **return**.

Numeric Literals

Integers

$$n ::= D^+.$$

Reals

$$r ::= D^+.D^+.$$

Scientific notation

$$a ::= D^+(.D^+)?E(\pm)?D^+.$$

String Literals

$$s ::= "c_1 \cdots c_n".$$

Comments

Single-line

$$\texttt{//}\ \alpha \rightarrow \texttt{newline}.$$

Multi-line

$$\texttt{/*}\ \alpha\ \texttt{*/}.$$

Concrete Grammar

Program

$$P ::= M^*.$$

Module

$$M ::= \textbf{module}\ I\ D.$$

Declaration

$$D ::= F \mid T \mid V.$$

Function

$$F ::= \textbf{let}\ I(A) = E.$$

Variable

$$V ::= \textbf{let}\ I = E.$$

Type

$$T ::= \textbf{type}\ I = \tau.$$

Expressions

$$
\begin{array}{rcl}
E & ::= & I \\
& | & n \\
& | & r \\
& | & s \\
& | & (E) \\
& | & E\ E \\
& | & \lambda I.E \\
& | & \textbf{sphere}(E) \\
& | & \textbf{pop}(E) \\
& | & \textbf{merge}(E, E) \\
& | & \textbf{choice}(p, E, E) \\
& | & \textbf{collapse}(E) \\
& | & \textbf{replay}(E) \\
& | & \textbf{bind}(E, E) \\
& | & \textbf{refuse}(R).
\end{array}
$$

Types

$$
\begin{array}{rcl}
\tau & ::= & \textbf{Bool} \\
& | & \textbf{Int} \\
& | & \textbf{Real} \\
& | & \textbf{History} \\
& | & \tau \to \tau \\
& | & \tau \times \tau \\
& | & \Pi x : \tau.\tau \\
& | & \Sigma x : \tau.\tau.
\end{array}
$$

Operator Precedence

1	$()$
2	$E\ E$
3	**pop**
4	$*\ /$
5	$+-$
6	$\rightarrow$
7	**merge**
8	**choice**
9	**collapse**
10	**bind**

Abstract Syntax

Define

$$\mathcal{T}$$

inductively.
Leaves

$$\mathrm{Var}(x), \qquad \mathrm{Const}(c).$$

Unary nodes

$$\mathrm{Sphere}, \mathrm{Pop}, \mathrm{Replay}, \mathrm{Collapse}\,.$$

Binary nodes

$$\mathrm{Bind}, \mathrm{Merge}, \mathrm{Apply}\,.$$

Ternary nodes

$$\mathrm{Choice}\,.$$

Grammar Homomorphism

Parser

$$\pi : \Sigma^* \to \mathcal{T}.$$

Properties

$$\pi(xy) = \pi(x) \circ \pi(y).$$

$$\pi((E)) = \text{Sphere}(\pi(E)).$$

Well-Formed Programs

A program

$$P$$

is well formed iff

$$\forall x, \qquad \text{Bind}(x) \text{ unique},$$

$$\text{Depth}(P) < \infty,$$

$$G(P) \text{ is acyclic},$$

$$\Gamma \vdash P.$$

Parsing Function

$$\text{Parse} : \Sigma^* \to (\mathcal{T}, \Gamma).$$

Correctness

$$\text{Print}(\text{Parse}(P)) = P.$$

Grammar Completeness

For every admissible abstract syntax tree

$$T,$$

there exists a concrete program

$$P$$

such that

$$\pi(P) = T.$$

Conversely,

$$\forall P, \qquad \pi(P)$$

is unique.

Reference Grammar Theorem

The concrete grammar

$$\mathcal{G} = (\Sigma, N, P, S)$$

is

unambiguous,
context-free,
deterministically parseable,
structurally complete,
closed under Sphere formation,
closed under Replay,
closed under Merge,
closed under Collapse.

Furthermore, every well-formed program admits a unique abstract syntax tree, a unique dependency graph, and a unique hierarchy of computational Spheres, up to α-equivalence.

□

N Standard Library Specification

Let

$$\Omega = \Omega_A \cup \Omega_H \cup \Omega_L \cup \Omega_G \cup \Omega_P \cup \Omega_M$$

denote the complete primitive operator library.

Arithmetic Algebra

Addition

$$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}.$$

Multiplication

$$\times : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}.$$

Division

$$/ : \mathbb{R} \times \mathbb{R}^{\times} \to \mathbb{R}.$$

Exponentiation

$$(-)^n : \mathbb{R} \to \mathbb{R}.$$

Identity

$$x + 0 = x,$$

$$x \cdot 1 = x.$$

Inverse

$$x + (-x) = 0.$$

Associativity

$$(x + y) + z = x + (y + z).$$

Commutativity

$$x + y = y + x.$$

Distributivity

$$x(y + z) = xy + xz.$$

Boolean Library

Conjunction

$$\wedge : \mathbf{B}^2 \to \mathbf{B}.$$

Disjunction

$$\vee : \mathbf{B}^2 \to \mathbf{B}.$$

Negation

$$\neg : \mathbf{B} \to \mathbf{B}.$$

Exclusive-or

$$\oplus : \mathbf{B}^2 \to \mathbf{B}.$$

Truth tables

$$x \wedge y = xy.$$

$$x \vee y = x + y - xy.$$

$$\neg x = 1 - x.$$

Comparison Operators

Equality

$$=: A \times A \to \mathbf{B}.$$

Ordering

$$<, \leq, >, \geq .$$

Membership

$$\in : A \times \mathcal{P}(A) \to \mathbf{B}.$$

History Operators

Open

$$\text{Sphere} : A \to \text{Hist}(A).$$

Replay

$$\text{Replay} : H \to H.$$

Merge

$$\text{Merge} : H \times H \to H.$$

Choice

$$\text{Choice} : [0,1] \times H \times H \to H.$$

Collapse

$$\text{Collapse} : H \to A.$$

Refusal

$$\text{Refuse} : R \to H.$$

Bind

$$\text{Bind} : H \times (A \to H) \to H.$$

History Laws

Replay

$$R^2 = R.$$

Collapse

$$C^2 = C.$$

Merge

$$H_1 \otimes H_2 = H_2 \otimes H_1.$$

Associativity

$$(H_1 \otimes H_2) \otimes H_3 = H_1 \otimes (H_2 \otimes H_3).$$

List Library

Empty list

$$[].$$

Construction

$$\text{cons} : A \times L(A) \to L(A).$$

Head

$$\text{head} : L(A) \to A.$$

Tail

$$\text{tail} : L(A) \to L(A).$$

Length

$$|\cdot| : L(A) \to \mathbb{N}.$$

Concatenation

$$++ : L(A) \times L(A) \to L(A).$$

Set Library

Union

$$\cup.$$

Intersection

$$\cap.$$

Difference

$$\setminus.$$

Power set

$$\mathcal{P}(A).$$

Cartesian product

$$A \times B.$$

Graph Library

Vertex

$$v \in V.$$

Edge

$$(u, v) \in E.$$

Parents

$$\text{Pred}(v).$$

Children

$$\text{Succ}(v).$$

Topological ordering

$$\text{Topo} : G \to (V_1, \ldots, V_n).$$

Matrix Library

Addition

$$A + B.$$

Multiplication

$$AB.$$

Transpose

$$A^T.$$

Determinant

$$\det(A).$$

Inverse

$$A^{-1}.$$

Trace

$$\operatorname{tr}(A).$$

Rank

$$\operatorname{rank}(A).$$

Probability Library

Distribution

$$\mathcal{D}(A).$$

Expectation

$$\mathbb{E}.$$

Variance

$$\text{Var}\,.$$

Sampling

$$\text{Sample} : \mathcal{D}(A) \to A.$$

Observation

$$\text{Observe} : A \to \mathcal{D}(A).$$

Differentiable Library

Gradient

$$\nabla.$$

Jacobian

$$J.$$

Hessian

$$H.$$

Chain rule

$$D(f \circ g) = Df \cdot Dg.$$

Tensor Library

Tensor product

$$\otimes.$$

Contraction

$$\mathrm{Tr}_i\,.$$

Outer product

$$u \otimes v.$$

Optimization Library

Gradient descent

$$x_{k+1} = x_k - \eta \nabla f(x_k).$$

Newton iteration

$$x_{k+1} = x_k - H^{-1} \nabla f.$$

Scheduler Library

Ready predicate

$$\mathrm{Ready}(S).$$

Selection

$$\mathrm{Select}(Q).$$

Execution

$$\mathrm{Execute}(S).$$

Historical update

$$H' = H :: e.$$

Library Closure

For every operator

$$\omega \in \Omega,$$

there exists

$$\omega : A_1 \times \cdots \times A_n \to B$$

such that

$$\Gamma \vdash A_i$$

implies

$$\Gamma \vdash \omega(A_1, \ldots, A_n).$$

Library Completeness Theorem

The standard library

$$\boxed{\Omega}$$

is closed under

functional composition,
historical replay,
parallel merge,
probabilistic choice,
logical inference,
graph construction,
linear algebra,
differential operators,
tensor contraction,
historical collapse.

Moreover, every primitive operator preserves admissibility, every composite operator admits a unique dependency graph, and every finite library expression is executable by the Universal Operator Machine through the operational semantics of the Spherepop Calculus.

□

O Rewrite System

Let

$$\mathcal{T}$$

denote the set of all well-typed Spherepop terms.
Define the one-step rewrite relation

$$\longrightarrow \subseteq \mathcal{T} \times \mathcal{T}.$$

Its reflexive-transitive closure is

$$\longrightarrow^{*}.$$

Structural Congruence

$$(H_1 \otimes H_2) \otimes H_3 \equiv H_1 \otimes (H_2 \otimes H_3).$$

$$H_1 \otimes H_2 \equiv H_2 \otimes H_1,$$

provided

$$H_1 \perp H_2.$$

$$H \otimes \varepsilon \equiv H.$$

β-Reduction

$$\mathrm{Pop}(\mathrm{Sphere}(x.t), u) \to t[u/x].$$

η-Reduction

$$\mathrm{Sphere}(x.\,\mathrm{Pop}(f, x)) \to f,$$

provided

$$x \notin FV(f).$$

Replay Rules

Idempotence

$$\text{Replay}(\text{Replay}(H)) \to \text{Replay}(H).$$

Collapse invariance

$$\text{Collapse}(\text{Replay}(H)) \to \text{Collapse}(H).$$

Merge distribution

$$\text{Replay}(H_1 \otimes H_2) \to \text{Replay}(H_1) \otimes \text{Replay}(H_2).$$

Collapse Rules

Idempotence

$$\text{Collapse}(\text{Collapse}(H)) \to \text{Collapse}(H).$$

Terminal value

$$\text{Collapse}(v) \to v.$$

Merge Rules

Associativity

$$(H_1 \otimes H_2) \otimes H_3 \to H_1 \otimes H_2 \otimes H_3.$$

Neutral element

$$H \otimes \varepsilon \to H.$$

Duplicate elimination

$$H \otimes H \to H,$$

when

$$H$$

is observationally identical.

Choice Rules

Degenerate probabilities

$$\text{Choice}(1, H_1, H_2) \to H_1.$$

$$\text{Choice}(0, H_1, H_2) \to H_2.$$

Equal branches

$$\text{Choice}(p, H, H) \to H.$$

Bind Rules

Left identity

$$\text{Bind}(\text{Return}(x), f) \to f(x).$$

Right identity

$$\text{Bind}(H, \text{Return}) \to H.$$

Associativity

$$\text{Bind}(\text{Bind}(H, f), g) \to \text{Bind}(H, \lambda x.\, \text{Bind}(f(x), g)).$$

Refusal Rules

Propagation

$$\text{Refuse}(r) \otimes H \to \text{Refuse}(r).$$

Replay

$$\text{Replay}(\text{Refuse}(r)) \to \text{Refuse}(r).$$

Collapse

$$\mathrm{Collapse}(\mathrm{Refuse}(r)) \to \mathrm{Refuse}(r).$$

Historical Simplification

Empty replay

$$\mathrm{Replay}(\varepsilon) \to \varepsilon.$$

Empty merge

$$\mathrm{Merge}(\varepsilon, H) \to H.$$

Redundant history

$$H :: \varepsilon \to H.$$

Graph Rewriting

Inline vertex

$$v \to \mathrm{Body}(v),$$

when

$$\deg^{-}(v) = 1.$$

Constant propagation

$$f(c) \to c'.$$

Dead node elimination

$$\mathrm{Succ}(v) = \varnothing \Longrightarrow v \to \varepsilon.$$

History Compression

Repeated replay

$$H :: H \to H.$$

Replay cache

$$H_i = H_j \Longrightarrow H_j \rightsquigarrow \text{Ref}(H_i).$$

Normalization Rules

Every rewrite decreases

$$\mu(T),$$

where

$$\mu : \mathcal{T} \to \mathbb{N}$$

is defined by

$$\mu(T) = \#\,\text{Sphere} + \#\,\text{Replay} + \#\,\text{Collapse} + \#\,\text{Merge}\,.$$

For every rewrite

$$T \to T',$$

$$\mu(T') < \mu(T),$$

except replay-preserving transformations.

Local Confluence

If

$$T \to T_1,$$

and

$$T \to T_2,$$

then there exists

$$T_3$$

such that

$$T_1 \to^* T_3,$$

$$T_2 \to^* T_3.$$

Historical Church–Rosser Conjecture

Suppose

$$T \to^* T_1,$$

and

$$T \to^* T_2.$$

Then there exists

$$T_3$$

such that

$$T_1 \to^* T_3,$$

$$T_2 \to^* T_3,$$

provided all Replay operations preserve admissible provenance.

Rewrite Completeness

Every finite Spherepop program admits a rewrite sequence

$$T = T_0 \to T_1 \to \cdots \to T_n = N,$$

where

$$N$$

is in normal form,

$$N \not\to .$$

Normal Form

A term

$$N$$

is in historical normal form iff

$$\begin{aligned}&\text{Replay} \notin N,\\ &\text{Collapse} \notin N,\\ &\text{Merge is flat},\\ &\text{Choice is terminal},\\ &\text{Sphere contains no reducible Pops}.\end{aligned}$$

Rewrite System Theorem

Let

$$\mathcal{R}$$

be the rewrite system consisting of all rules defined above.
Then

$$\boxed{\mathcal{R} = \{\beta, \eta, \text{Replay}, \text{Collapse}, \text{Merge}, \text{Choice}, \text{Bind}, \text{Refuse}, \text{Compression}, \text{Graph}\}}$$

forms a terminating rewrite system over every finite acyclic dependency graph.
Furthermore,

$$T \to^* N$$

for every finite well-typed program

$$T,$$

where

$$N$$

is unique up to structural congruence and replay equivalence.

□

P Benchmark Programs

Let

$$\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$$

denote the collection of reference Spherepop programs.
Each benchmark consists of

$$(P, H, G, T, C),$$

where

$$P$$

is the program,

$$H$$

its execution history,

$$G$$

its dependency graph,

$$T$$

its typing derivation,
and

$$C$$

its terminal Collapse.

Benchmark 1: Identity

$$I = \text{Sphere}(x.x).$$

Evaluation

$$\mathrm{Pop}(I, a) \to a.$$

History

$$H_I = (\mathrm{Open}, \mathrm{Bind}, \mathrm{Pop}, \mathrm{Collapse}).$$

Complexity

$$O(1).$$

Benchmark 2: Composition

$$(f \circ g)(x) = f(g(x)).$$

Dependency graph

$$g \prec f.$$

Execution

$$x \to g \to f.$$

History depth

$$D = 2.$$

Benchmark 3: Recursive Fibonacci

$$F(0) = 0,$$

$$F(1) = 1,$$

$$F(n) = F(n-1) + F(n-2).$$

Replay graph

$$G_F = (V, E).$$

Naive complexity

$$O(\phi^n).$$

Replay-compressed complexity

$$O(n).$$

Historical compression ratio

$$\gamma = \frac{\phi^n}{n}.$$

Benchmark 4: Merge Sort

Split

$$L = L_1 \cup L_2.$$

Recursive histories

$$H = H_1 \otimes H_2.$$

Merge

$$H' = \text{Merge}(H_1, H_2).$$

Complexity

$$O(n \log n).$$

Parallel depth

$$O(\log n).$$

Benchmark 5: Dijkstra

State

$$(V, E, w).$$

Distance update

$$d(v) = \min(d(v), d(u) + w(u, v)).$$

Replay records every relaxation.
History size

$$O(E).$$

Complexity

$$O(E \log V).$$

Benchmark 6: Logic Circuit

Circuit

$$C = (V, E).$$

Gate

$$g_i : \{0,1\}^k \to \{0,1\}.$$

Execution

$$H_C = \bigotimes_i H_i.$$

Collapse

$$C = \text{Collapse}(H_C).$$

Complexity

$$O(|V| + |E|).$$

Benchmark 7: Lambda Interpreter

Program

$$(\lambda x.x)y.$$

Reduction

$$(\lambda x.x)y \to y.$$

Replay records

$$\beta$$

reductions explicitly.

Benchmark 8: SAT Solver

Formula

$$\phi = C_1 \wedge \cdots \wedge C_m.$$

Branching

$$\text{Choice}(p, H_T, H_F).$$

Conflict

$$\text{Refuse}(r).$$

Successful assignment

$$\text{Collapse}(v).$$

Benchmark 9: Feedforward Neural Network

Layer

$$x_{k+1} = \sigma(W_k x_k + b_k).$$

Execution graph

$$H = H_1 \otimes \cdots \otimes H_L.$$

Replay computes reverse dependencies.
Backpropagation

$$\nabla L = J^T \nabla L.$$

Benchmark 10: Parser

Input

$$w \in \Sigma^*.$$

Parser

$$P : \Sigma^* \to \mathcal{T}.$$

History

$$\text{Lex} \prec \text{Parse} \prec \text{Type}\,.$$

Complexity

$$O(n).$$

Benchmark 11: Historical Replay

Initial history

$$H.$$

Replay

$$R(H).$$

Property

$$R(R(H)) = R(H).$$

Complexity

$$O(R(v)).$$

Benchmark 12: Parallel Scheduler

Ready queue

$$Q = \{v : \text{Ready}(v)\}.$$

Scheduler

$$S : Q \to V.$$

Parallel execution

$$H = \bigotimes_i H_i.$$

Critical path

$$D(H).$$

Speedup

$$S = \frac{|V|}{D(H)}.$$

Benchmark Metrics

For every benchmark

$$P_i$$

measure

$$|V|,$$
$$|E|,$$
$$|H|,$$
$$D(H),$$
$$W(H),$$
$$R_{\max},$$
$$M,$$
$$T,$$
$$\gamma,$$
$$C.$$

Correctness Criterion

For every benchmark

$$P,$$

$$\text{Collapse}(\text{Execute}(P)) = \text{Reference}(P).$$

Operational correctness

$$P = P_{\text{ref}}.$$

Historical Benchmark Theorem

Every benchmark program

$$P_i$$

admits

a unique dependency graph,
a well-typed derivation,
a finite execution history,
a replay reconstruction,
a terminal Collapse,
a canonical normal form.

Moreover, execution by the Universal Operator Machine is observationally equivalent to the corresponding mathematical specification, and replay preserves the complete provenance of every benchmark computation.

□

Q Comparison with Existing Formalisms

Let

$$\mathcal{F} = \{\Lambda, \Pi, \mathcal{L}, \mathcal{C}, \mathcal{P}, \mathcal{S}\}$$

denote the collection of computational formalisms, where

Λ

is the untyped λ-calculus,

Π

the π-calculus,

$\mathcal{L}$

linear logic,

$\mathcal{C}$

the Calculus of Constructions,

$\mathcal{P}$

Petri nets,
and

$\mathcal{S}$

the Spherepop Calculus.

Computational State

Formalism	Primary Computational Object
Λ	λ-term
Π	Channel Process
$\mathcal{L}$	Resource
$\mathcal{C}$	Typed Construction
$\mathcal{P}$	Marking
$\mathcal{S}$	History

Scope

Formalism	Scope Representation
Λ	$()$
Π	ν
$\mathcal{L}$	Γ
$\mathcal{C}$	Γ
$\mathcal{P}$	Places
$\mathcal{S}$	Embedded Sphere

Evaluation

Formalism	Evaluation Primitive
Λ	β
Π	Communication
$\mathcal{L}$	Cut Elimination
$\mathcal{C}$	$\beta\eta$
$\mathcal{P}$	Transition Firing
$\mathcal{S}$	Pop

History

Define

$$h(F) = \begin{cases} 0, & F \neq \mathcal{S}, \\ 1, & F = \mathcal{S}. \end{cases}$$

Only Spherepop treats execution history as a primitive semantic object.

Replay

Replay operator

$$R_F = \begin{cases} \varnothing, & F \neq \mathcal{S}, \\ R, & F = \mathcal{S}. \end{cases}$$

Refusal

Refusal structure

$$\rho(F) = \begin{cases} 0, & \Lambda \\ 0, & \Pi \\ 0, & \mathcal{L} \\ 0, & \mathcal{C} \\ 0, & \mathcal{P} \\ 1, & \mathcal{S}. \end{cases}$$

Collapse

Collapse operator

$$C_F = \begin{cases} \beta, & \Lambda \\ \text{Commit}, & \Pi \\ \text{Cut}, & \mathcal{L} \\ \beta, & \mathcal{C} \\ \text{Fire}, & \mathcal{P} \\ \text{Collapse}, & \mathcal{S}. \end{cases}$$

Concurrency

Concurrency algebra

F	$\text{Conc}(F)$
Λ	0
Π	1
$\mathcal{L}$	1
$\mathcal{C}$	0
$\mathcal{P}$	1
$\mathcal{S}$	1

Probabilistic Semantics

Probability functor

$$\mathcal{D}_F = \begin{cases} \varnothing, & F \neq \mathcal{S}, \\ \mathcal{D}, & F = \mathcal{S}. \end{cases}$$

Differentiability

Smooth operator algebra

$$\delta(F) = \begin{cases} 0, & \Lambda \\ 0, & \Pi \\ 0, & \mathcal{L} \\ 0, & \mathcal{C} \\ 0, & \mathcal{P} \\ 1, & \mathcal{S}. \end{cases}$$

Dependency Geometry

Dependency graph

$$G_F = \begin{cases} \text{Implicit}, & \Lambda \\ \text{Channel Graph}, & \Pi \\ \text{Proof Net}, & \mathcal{L} \\ \text{Typing Tree}, & \mathcal{C} \\ \text{Petri Graph}, & \mathcal{P} \\ \text{Historical DAG}, & \mathcal{S}. \end{cases}$$

Categorical Models

F	Category
Λ	CCC
Π	$Process$
$\mathcal{L}$	$SMCC$
$\mathcal{C}$	$LCCC$
$\mathcal{P}$	$Transition$
$\mathcal{S}$	**Hist**

where

$$CCC$$

denotes Cartesian Closed Categories,

$$SMCC$$

Symmetric Monoidal Closed Categories,
and

$$LCCC$$

Locally Cartesian Closed Categories.

Expressive Embeddings

There exist faithful embeddings

$$\Lambda \hookrightarrow \mathcal{S},$$

$$\Pi \hookrightarrow \mathcal{S},$$

$$\mathcal{C} \hookrightarrow \mathcal{S},$$

$$\mathcal{P} \hookrightarrow \mathcal{S}.$$

Consequently

$$\mathcal{S}$$

acts as a common operational superstructure.

Simulation

For every formalism

$$F \in \mathcal{F},$$

there exists an interpretation

$$I_F : F \to \mathcal{S}.$$

Correctness requires

$$P_F = I_F(P)_{\mathcal{S}}.$$

Universality

Define

$$\mathfrak{U}(F) = 1$$

iff every finite computation representable in

$$F$$

is representable in

$$\mathcal{S}.$$

Then

$$\forall F \in \mathcal{F}, \qquad \mathfrak{U}(F) = 1.$$

Relative Expressiveness

Let

$$\preceq$$

denote simulation.
Then

$$\Lambda \preceq \mathcal{S},$$

$$\Pi \preceq \mathcal{S},$$

$$\mathcal{C} \preceq \mathcal{S},$$

$$\mathcal{P} \preceq \mathcal{S}.$$

Whether

$$\mathcal{S} \preceq \Lambda$$

holds while preserving explicit histories and refusal structures remains open.

Comparative Completeness Theorem

The Spherepop Calculus admits faithful embeddings of every formalism listed above while preserving operational semantics up to observational equivalence.

Furthermore, Spherepop uniquely provides

> persistent computational histories,
> explicit replay semantics,
> refusal as a primitive operation,
> geometric scoping,
> history-preserving execution,
> operator-parametric logic,
> native probabilistic composition,
> a unified operational model for
> λ-calculus, process calculi,
> proof theory, graph computation, and
> differentiable computation.

Consequently,

$$\mathcal{S}$$

may be regarded as a common semantic framework within which diverse computational paradigms admit a unified historical interpretation.

□

R Formal Conjectures

Throughout this appendix let

$$\mathcal{H}$$

denote the complete lattice of admissible histories,

$$\mathcal{U}$$

the Universal Operator Machine,
and

Hist

the category of histories.

The following conjectures summarize the principal mathematical questions that remain unresolved.

Conjecture 1 (Replay Normalization)

Every finite admissible history

$$H$$

admits a finite replay normalization sequence

$$H \longrightarrow^* N,$$

where

$$N$$

contains no redundant replay operators.
Equivalently,

$$R(H) = R(N),$$

and

$$N \not\rightarrow .$$

Conjecture 2 (Historical Church–Rosser)

If

$$H \longrightarrow^* H_1,$$

and

$$H \longrightarrow^* H_2,$$

then there exists

$$H_3$$

such that

$$H_1 \longrightarrow^* H_3,$$

$$H_2 \longrightarrow^* H_3.$$

Conjecture 3 (Strong Historical Normalization)

Every deterministic Spherepop computation satisfies

$$H \longrightarrow^* N$$

after finitely many reductions.
No infinite replay sequence exists.

Conjecture 4 (Probabilistic Normalization)

Suppose

$$H$$

contains finitely many Choice operators.
Then

$$\Pr\left(H \Downarrow\right) = 1.$$

That is,

$$H$$

normalizes almost surely.

Conjecture 5 (Replay Minimality)

Among all histories observationally equivalent to

$$H,$$

Replay produces one of minimal provenance complexity.
Formally,

$$|R(H)| = \min\{|K| : K \sim H\}.$$

Conjecture 6 (Scheduler Optimality)

Let

$$D(H)$$

denote dependency depth.

The scheduler produced by the Universal Operator Machine minimizes execution time among all admissible schedulers:

$$T_{\mathcal{U}} = \inf T.$$

Conjecture 7 (Replay Stability)

There exists

$$L > 0$$

such that

$$d(R(H_1), R(H_2)) \leq L d(H_1, H_2)$$

for every pair of histories.

Conjecture 8 (Geometric Convergence)

Repeated Pop evolution converges toward a minimal-energy boundary.

If

$$S_t$$

denotes the evolving computational sphere,
then

$$S_t \to S_\infty$$

where

$$S_\infty$$

minimizes

$$E(S).$$

Conjecture 9 (Historical Shape Equidistribution)

Let

$$\mathcal{H}_d(X)$$

be admissible histories of dependency rank

$$d$$

ordered by historical discriminant.
Then

$$\text{sh}(\mathcal{H}_d(X)) \Rightarrow \mu_d$$

as

$$X \to \infty.$$

Conjecture 10 (Hyperplane Limit)

Let

$$C_\beta$$

be differentiable circuits whose activation parameter satisfies

$$\beta \to \infty.$$

Then

$$F_{C_\beta} \to F_C$$

uniformly away from decision hyperplanes.

Conjecture 11 (Universal Simulation)

Every finite computational calculus

$$\mathcal{A}$$

whose primitive operations form a finitely generated operator algebra admits a faithful embedding

$$I : \mathcal{A} \hookrightarrow \mathcal{S}.$$

Conjecture 12 (Historical Completeness)

Every finite computation possesses a unique minimal admissible history

$$H_{\min},$$

up to replay equivalence.

Conjecture 13 (Historical Homotopy)

The quotient space

$$\mathcal{H}/\sim$$

admits a homotopy theory in which replay equivalences determine path components.

Conjecture 14 (Quantum Replay)

There exists a quantum extension

$$R_q$$

compatible with unitary evolution satisfying

$$R_q(UH) = UR_q(H)$$

for every admissible unitary transformation

$$U.$$

Conjecture 15 (Distributed Histories)

Suppose

$$H_1, \ldots, H_n$$

execute concurrently.
Then there exists a unique global replay history

$$H_G$$

satisfying

$$R(H_G) = \bigotimes_i R(H_i)$$

up to causal equivalence.

Conjecture 16 (Universal Historical Representation)

Every terminating computation may be represented uniquely by

$$(G, H, C),$$

where

$$G$$

is a dependency graph,

$$H$$

an admissible history,
and

$$C$$

its terminal Collapse.

Two computations are operationally equivalent if and only if these triples are isomorphic.

Grand Conjecture

There exists a category

$$\textbf{Hist}$$

equipped with

finite limits,
finite colimits,
a replay monad,
a history comonad,
a symmetric monoidal structure,
a geometric realization functor,
a probabilistic enrichment,
an admissibility topology,

such that every terminating computation in every finitely generated operator calculus factors uniquely through

$$\textbf{Hist},$$

preserving operational semantics, denotational semantics, dependency structure, replay, provenance, and terminal Collapse up to natural isomorphism.

$$\boxed{\forall \mathcal{A}, \qquad \exists!\ F : \mathcal{A} \longrightarrow \textbf{Hist}}$$

where

$$F$$

is faithful, history-preserving, and computationally complete.

□

References

[1] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[2] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

[3] E. L. Post. Formal Reductions of the General Combinatorial Decision Problem. *American Journal of Mathematics*, 65:197–215, 1943.

[4] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[5] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[6] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[7] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[8] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[9] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[10] S. Mac Lane. *Categories for the Working Mathematician*. Springer, Second Edition, 1998.

[11] S. Awodey. *Category Theory*. Oxford University Press, Second Edition, 2010.

[12] P. T. Johnstone. *Sketches of an Elephant*. Oxford University Press, 2002.

[13] D. Scott. Outline of a Mathematical Theory of Computation. Oxford University Computing Laboratory Technical Monograph, 1970.

[14] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[15] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[16] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[17] R. Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[18] C. A. Petri. *Communication with Automata*. Technical Report RADC-TR-65-377, 1966.

[19] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[20] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Second Edition, 2007.

[21] A. W. Appel. *Modern Compiler Implementation*. Cambridge University Press, 1998.

[22] R. Diestel. *Graph Theory*. Springer, Fifth Edition, 2017.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Third Edition, 2009.

[24] P. Billingsley. *Probability and Measure*. Wiley, Third Edition, 1995.

[25] M. Giry. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis*, 1982.

[26] J. M. Lee. *Introduction to Riemannian Manifolds*. Springer, Second Edition, 2018.

[27] M. P. do Carmo. *Riemannian Geometry*. Birkhäuser, 1992.

[28] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[29] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[30] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

[31] S. Abramsky and A. Jung. Domain Theory. In *Handbook of Logic in Computer Science*, Volume 3, 1994.

[32] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[33] L. de Moura et al. The Lean 4 Theorem Prover and Programming Language. 2021.

[34] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6:308–320, 1964.

[35] J.-L. Krivine. A Call-by-Name Lambda Calculus Machine. *Higher-Order and Symbolic Computation*, 20:199–207, 2007.

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

[37] M. Bhargava. The Density of Discriminants of Quintic Rings and Fields. *Annals of Mathematics*, 172:1559–1591, 2010.

[38] M. Bhargava, A. Shankar, and J. Tsimerman. On the Davenport–Heilbronn Theorems and Second Order Terms. *Inventiones Mathematicae*, 193:439–499, 2013.

[39] G. Gentzen. Investigations into Logical Deduction. *Mathematische Zeitschrift*, 1935.

[40] A. Tarski. *Logic, Semantics, Metamathematics*. Oxford University Press, 1956.

[41] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.