Types as Refusal Structures

Toward a Calculus of Constructions in Spherepop

Flyxion

June 2026

Abstract

This essay develops a path from the operational calculus of Spherepop to a dependent type calculus and finally to a Spherepop-native version of the calculus of constructions. The guiding idea is that a type is not merely a collection of values, nor merely a proposition, nor merely a syntactic classifier. A type is a disciplined refusal structure: it determines which histories may continue, which collapses are admissible, and which malformed constructions must be refused before they become executable commitments. In ordinary type theory, judgments such as $\Gamma \vdash t : A$ classify terms relative to contexts. In Spherepop, the same judgment is interpreted dynamically as the claim that the history Γ can be extended by the event t without violating the continuation conditions encoded by A. This shifts the emphasis from static membership to controlled continuation. The calculus of constructions is then reconstructed as a hierarchy of admissible continuation principles: terms inhabit types, types depend on terms, propositions are types of proofs, and universes classify the stages at which refusal rules themselves may be formed. The resulting system treats computation, proof, and program construction as modes of history repair under admissibility constraints.

1 Introduction

Spherepop begins from a simple operational reversal. Instead of treating programs as inert expressions that later acquire histories, it treats histories as primary and programs as controlled ways of extending them. The basic operations of the calculus are pop, refuse, collapse, and bind. A computation may open a possible continuation, refuse an inadmissible one, collapse a resolved branch into a committed value, or bind one history-producing operation to another. This gives Spherepop the character of a process calculus, but its philosophical orientation is closer to repair theory: a

program is not merely a function from inputs to outputs, but a disciplined way of preserving coherence while new distinctions are introduced.

Type theory is a natural next step because types are already the formal language of disciplined continuation. In a simply typed lambda calculus, a term cannot be applied to an argument unless the argument has the required type. In a dependent type theory, a type may depend on a value, so that the admissibility of a continuation can vary with the history already produced. In the calculus of constructions, propositions and types are unified: to prove a proposition is to construct an inhabitant of the corresponding type. This is exactly the kind of structure Spherepop needs if it is to become not only a language of executable histories, but a language of certified histories.

The central claim of this essay is that a Spherepop type should be understood as a refusal boundary over possible continuations. A value of type A is not simply an element of a set. It is a history fragment that survives the refusal tests imposed by A. A function type $A \to B$ is not merely a set of mappings. It is a repair operator that transforms any admissible A-continuation into an admissible B-continuation. A dependent product $\Pi x : A.\, B(x)$ is a uniform repair operator whose output admissibility may depend on the input history. A dependent sum $\Sigma x : A.\, B(x)$ is a paired commitment: first a witness x is admitted, then a second witness is admitted relative to it. Identity types express controlled substitutability between histories. Universes classify the levels at which these refusal structures may themselves be named, transported, and constructed.

This interpretation does not replace ordinary type theory. Rather, it reinterprets its judgments operationally. The familiar judgment

$$\Gamma \vdash t : A$$

is read as follows: under the accumulated history Γ, the expression t is an admissible continuation satisfying the refusal structure A. The judgment

$$\Gamma \vdash A : \mathsf{Type}$$

says that A is itself a well-formed refusal structure in the current history. The judgment

$$\Gamma \vdash A \equiv B : \mathsf{Type}$$

says that A and B impose the same continuation discipline up to normalization, repair, or collapse equivalence.

The point of building a calculus of constructions inside Spherepop is not merely to imitate Coq or Agda. It is to make proof, execution, and repair share one operational substrate. A proof is a history that cannot be refused by the proposition it inhabits. A program is a history transformer that preserves admissibility. A type error is not a failed decoration pass after syntax has already been accepted, but an early refusal event preventing an incoherent history from becoming a committed collapse.

2 The Operational Core of Spherepop

We begin with a minimal operational vocabulary. Let Hist be a class of finite histories. A history $h \in \mathsf{Hist}$ records the sequence of commitments, refusals, collapses, and bindings that have occurred so far. Spherepop programs are not interpreted as direct maps from values to values, but as partial continuation procedures on histories.

Definition 2.1 (Spherepop operation)**.** *A primitive Spherepop operation is one of the following history-transforming acts:*

$$\mathsf{pop}(a), \qquad \mathsf{refuse}(r), \qquad \mathsf{collapse}(v), \qquad \mathsf{bind}(p, q).$$

Here $\mathsf{pop}(a)$ *opens or exposes a possible continuation labelled by* a*;* $\mathsf{refuse}(r)$ *blocks a continuation for reason* r*;* $\mathsf{collapse}(v)$ *commits a resolved continuation to value* v*; and* $\mathsf{bind}(p, q)$ *composes a history-producing procedure* p *with a continuation procedure* q*.*

The distinction between refusal and collapse is essential. A collapse commits. A refusal prevents commitment. A malformed program should not collapse to an error value if the error represents a violation of admissibility. It should be refused before it becomes a member of the history of accepted computation.

This means that Spherepop already contains the seed of type theory. A type system is precisely a disciplined refusal mechanism. It prevents some expressions from entering the accepted operational history. The usual distinction between dynamic and static semantics becomes a distinction between late refusal and early refusal. A dynamically checked language allows many histories to proceed until failure. A typed Spherepop system moves refusal closer to the point where inadmissibility first appears.

Definition 2.2 (Admissible continuation)**.** *Given a history* h*, a continuation* k *is admissible relative to* h *when extending* h *by* k *does not force a refusal:*

$$\mathsf{Adm}(h, k) \quad \Longleftrightarrow \quad h \cdot k \not\longrightarrow^{*} \mathsf{refuse}(r)$$

for any terminal refusal reason r*.*

A type can now be introduced as a predicate on continuations, but this predicate should not be thought of as merely external. A type is part of the operational language. It is a programmable refusal surface.

Definition 2.3 (Type as refusal structure)**.** *A Spherepop type over a history* h *is a rule* A *assigning to each candidate continuation* k *either admission, refusal, or deferred admissibility:*

$$A_h(k) \in \{\mathsf{admit}, \mathsf{refuse}, \mathsf{defer}\}.$$

A term t *has type* A *in history* h *when the continuation generated by* t *is admitted by* A *relative to* h*.*

The value defer is important. It allows open terms, dependent obligations, unresolved constraints, and interactive proof states. A type calculus for Spherepop should not require every admissibility question to be settled immediately. It should allow some refusals to remain suspended until the missing history has been supplied.

3 Contexts as Histories

In ordinary type theory, a context is a list of variable declarations:

$$x_1 : A_1, \ldots, x_n : A_n.$$

In Spherepop, this is reinterpreted as a history of admissible openings. Each variable declaration says that a possible continuation has been popped and accepted as a stable dependency for later construction.

Definition 3.1 (Spherepop context)**.** *A context is a finite admissible history of typed openings:*

$$\Gamma ::= \varnothing \mid \Gamma, x : A.$$

The extension $\Gamma, x : A$ *is valid only when* $\Gamma \vdash A : \mathsf{Type}$ *and the name* x *is fresh in* Γ*.*

Thus a context is not merely an environment. It is a sequence of permitted assumptions. Every variable in the context is a controlled unresolved continuation. It has not been collapsed to a concrete value, but it has been admitted as a legitimate dependency.

The context formation rules become the first layer of the type calculus:

$$\frac{}{\varnothing\ \mathsf{ctx}} \qquad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A : \mathsf{Type} \qquad x \notin \Gamma}{\Gamma, x : A\ \mathsf{ctx}}.$$

The variable rule says that a previously admitted opening may be used:

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

Operationally, this is a controlled pop. The variable x names a continuation that has already passed the refusal boundary encoded by A.

4 The Basic Judgment Forms

A Spherepop type calculus requires several judgment forms. The minimal collection is:

$$\Gamma\ \mathsf{ctx},$$

$$\Gamma \vdash A : \mathsf{Type},$$
$$\Gamma \vdash t : A,$$
$$\Gamma \vdash A \equiv B : \mathsf{Type},$$
$$\Gamma \vdash t \equiv u : A.$$

The first says that Γ is an admissible history of assumptions. The second says that A is a well-formed refusal structure. The third says that t is an admitted continuation of type A. The fourth and fifth express definitional equality: two types or terms are the same for purposes of computation and refusal.

Definitional equality is crucial in a calculus of constructions because types may compute. If B reduces to A, then an inhabitant of A may also be used as an inhabitant of B. In Spherepop terms, definitional equality identifies histories whose collapses produce the same refusal behavior.

Definition 4.1 (Collapse equivalence)**.** *Two terms t and u are collapse-equivalent at type A in context Γ when they normalize to the same committed history:*

$$\Gamma \vdash t \equiv u : A \quad \Longleftrightarrow \quad t \Downarrow v,\ u \Downarrow v,$$

up to admissible renaming and suspended obligations.

This definition is intentionally schematic. A real implementation must specify the reduction relation, the normalization strategy, and the permitted equivalences. But the guiding idea is clear: definitional equality is equality of computational behavior before proof-relevant distinctions are considered.

5 Simple Types in Spherepop

The simply typed fragment contains base types, function types, product types, sum types, and perhaps a unit type. Let ι range over base types. Formation begins with:

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \iota : \mathsf{Type}}.$$

Function types are formed by:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \to B : \mathsf{Type}}.$$

The introduction rule for functions is lambda abstraction:

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

The elimination rule is application:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B}.$$

The computation rule is beta reduction:

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

The extensional principle is eta conversion:

$$f \equiv \lambda x.f\, x.$$

In Spherepop, a function is a continuation transformer. The term $\lambda x.t$ says: if the history is extended by an admissible A-opening named x, then t produces an admissible B-continuation. Application binds a function history to an argument history. Beta reduction is the collapse of that binding.

Thus the ordinary computation rule

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

becomes a repair rule: an explicit dependency on an open admissible continuation is repaired by substituting a concrete admitted continuation.

6 Dependent Types

Simple types are insufficient for Spherepop because admissibility often depends on history. A file path may be valid only after a directory has been opened. A proof obligation may be meaningful only after a proposition has been formed. A repair rule may be applicable only after a defect has been witnessed. These are dependent phenomena. The type of a continuation may depend on a previous continuation.

The dependent product is written:

$$\Pi x : A.\, B(x).$$

It generalizes the function type. A term of this type is a procedure that, for every admitted $x : A$, produces an admitted continuation of type $B(x)$.

The formation rule is:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B(x) : \mathsf{Type}}{\Gamma \vdash \Pi x : A.\, B(x) : \mathsf{Type}}.$$

Introduction is:

$$\frac{\Gamma, x : A \vdash t : B(x)}{\Gamma \vdash \lambda x.t : \Pi x : A.\, B(x)}.$$

Elimination is:

$$\frac{\Gamma \vdash f : \Pi x : A.\, B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B(a)}.$$

The beta rule remains:

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

The dependent sum is written:

$$\Sigma x : A.\, B(x).$$

It represents a pair consisting of a witness $a : A$ and a second witness $b : B(a)$. Its formation rule is:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B(x) : \mathsf{Type}}{\Gamma \vdash \Sigma x : A.\, B(x) : \mathsf{Type}}.$$

Its introduction rule is:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B(a)}{\Gamma \vdash (a, b) : \Sigma x : A.\, B(x)}.$$

Its eliminations are projections:

$$\frac{\Gamma \vdash p : \Sigma x : A.\, B(x)}{\Gamma \vdash \pi_1(p) : A},$$

$$\frac{\Gamma \vdash p : \Sigma x : A.\, B(x)}{\Gamma \vdash \pi_2(p) : B(\pi_1(p))}.$$

In Spherepop, Π is universal repair and Σ is witnessed repair. The product says that no matter which admissible history fragment is supplied, continuation can be preserved. The sum says that a particular admissible fragment has been found, together with the dependent evidence required to continue from it.

7 Propositions as Types

The Curry–Howard interpretation identifies propositions with types and proofs with inhabitants. Spherepop gives this identification an operational reading. A proposition is a refusal structure whose inhabitants are histories that survive its demand for evidence. To prove P is to construct a continuation that P cannot refuse.

We introduce a universe Prop of propositions:

$$\Gamma \vdash \mathsf{Prop} : \mathsf{Type}.$$

If $P : \mathsf{Prop}$, then a proof of P is a term $p : P$. Logical implication is represented by function type:

$$P \Rightarrow Q \quad := \quad P \to Q.$$

Universal quantification is represented by dependent product:

$$\forall x : A.\, P(x) \quad := \quad \Pi x : A.\, P(x).$$

Existential quantification is represented by dependent sum:

$$\exists x : A.\, P(x) \quad := \quad \Sigma x : A.\, P(x).$$

The Spherepop interpretation is direct. A proof of $P \to Q$ is a repair operation that converts any admitted proof-history of P into an admitted proof-history of Q. A proof of $\forall x : A.\, P(x)$ is a uniform method that produces evidence for $P(x)$ after any admissible x is supplied. A proof of $\exists x : A.\, P(x)$ is a witness and a dependent proof relative to that witness.

Negation can be defined through refusal:

$$\neg P := P \to \bot.$$

Here $\bot$ is the uninhabited type, the type whose only operational behavior is refusal. To prove $\neg P$ is to show that any attempted proof of P forces collapse into impossibility.

This gives Spherepop a particularly natural account of contradiction. A contradiction is not merely a syntactic pair P and $\neg P$. It is a history that admits both a continuation and a rule proving that continuation inadmissible. Such a history cannot be collapsed coherently. It must be repaired, quarantined, or refused.

8 Identity Types and Substitution

A dependent type theory needs identity types. Given $a, b : A$, the identity type

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

is the type of evidence that a and b may be treated as the same at type A.

The formation rule is:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \mathsf{Id}_A(a, b) : \mathsf{Type}}.$$

The introduction rule is reflexivity:

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

In ordinary intensional type theory, identity proofs need not collapse all differences between a and b. They provide controlled substitutability. This is exactly the

Spherepop reading. An identity proof is a repair path showing that one history fragment may replace another without violating the relevant refusal structures.

The eliminator for identity, often called J, says that to prove a property for all identity proofs, it is enough to prove it for reflexivity. In schematic form:

$$\frac{\Gamma, x : A \vdash C(x, x, \mathsf{refl}_x) : \mathsf{Type} \qquad \Gamma, x : A \vdash c(x) : C(x, x, \mathsf{refl}_x)}{\Gamma, a : A, b : A, p : \mathsf{Id}_A(a, b) \vdash J(c, a, b, p) : C(a, b, p)}.$$

Operationally, J is the principle that a repair path may be collapsed back to the trivial repair path when the only admissible distinction is reflexive. It is not an assertion that all equalities are judgmentally identical. It is a disciplined way of transporting constructions across admitted identity evidence.

9 Universes and the Calculus of Constructions

The calculus of constructions requires types whose elements may themselves be types. To avoid paradoxes, one usually uses a hierarchy of universes:

$$\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \cdots .$$

Spherepop should adopt such a hierarchy. A universe is a level at which refusal structures may be named as objects.

The formation rule is:

$$\frac{\Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}.$$

If $A : \mathsf{Type}_i$ and $B : \mathsf{Type}_i$ under $x : A$, then $\Pi x : A.\, B(x)$ also belongs to an appropriate universe:

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma, x : A \vdash B(x) : \mathsf{Type}_j}{\Gamma \vdash \Pi x : A.\, B(x) : \mathsf{Type}_{\max(i,j)}}.$$

Similarly,

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma, x : A \vdash B(x) : \mathsf{Type}_j}{\Gamma \vdash \Sigma x : A.\, B(x) : \mathsf{Type}_{\max(i,j)}}.$$

A Spherepop calculus of constructions is now obtained by combining four ingredients: dependent products, dependent sums, universes, and propositions-as-types. Its judgments are histories; its types are refusal structures; its terms are admissible continuations; its reductions are collapse rules; its proof terms are certified repairs.

The resulting system may be summarized as:

$$\text{program} = \text{history transformer},$$

$$\text{type} = \text{refusal structure},$$

$$\text{proof} = \text{unrefused continuation},$$

$$\text{computation} = \text{collapse of admissible history},$$

$$\text{type checking} = \text{early refusal of incoherent continuation}.$$

10 Spherepop Syntax for Dependent Construction

A concrete syntax might begin with the following forms:

$$t, u ::= x \mid \lambda x : A.\, t \mid t\, u \mid \Pi x : A.\, B \mid (t, u) \mid \Sigma x : A.\, B \mid \pi_1(t) \mid \pi_2(t) \mid \mathsf{Type}_i \mid \mathsf{refl}(t) \mid J(c, a, b, p).$$

Spherepop-specific forms may be added:

$$t ::= \mathsf{pop}(a) \mid \mathsf{refuse}(r) \mid \mathsf{collapse}(v) \mid \mathsf{bind}(t, u) \mid \mathsf{repair}(t) \mid \mathsf{defer}(o).$$

The key design question is whether these operational forms live at the term level, the proof level, or both. The most natural answer is that they live at both levels but are typed differently. A computational refusal and a logical contradiction are not the same event, but both are refusal events. A computational collapse and a proof normalization are not the same event, but both are collapse events.

A typed form of refuse may be written:

$$\frac{\Gamma \vdash r : \mathsf{Reason}(A)}{\Gamma \vdash \mathsf{refuse}(r) : A}$$

only in a controlled exceptional or empty context. In a total proof calculus, unrestricted refusal would trivialize the system, because from refusal one could inhabit arbitrary types. Therefore Spherepop must distinguish between object-level refusal and meta-level refusal. Object-level refusal records a failed continuation. Meta-level refusal prevents a judgment from being formed.

This distinction is fundamental. The type checker should not produce a term $\mathsf{refuse}(r) : A$ whenever checking fails. It should refuse the judgment $\Gamma \vdash t : A$ itself. In other words, refusal is not generally an inhabitant. It is a boundary condition on inhabitation.

11 Normalization and Type Checking

A practical Spherepop calculus requires normalization. Type checking dependent types often requires comparing types after computation. For example, if A reduces

to B, then a term of type A should be usable at type B. This requires a normalization or conversion procedure.

We write:

$$t \Downarrow n$$

when t normalizes to n. Definitional equality may then be checked by normalization:

$$\frac{t \Downarrow n \qquad u \Downarrow n}{\Gamma \vdash t \equiv u : A}.$$

For an implementation, this suggests a bidirectional type checker. Inference synthesizes a type:

$$\Gamma \vdash t \Rightarrow A,$$

while checking verifies a term against a known type:

$$\Gamma \vdash t \Leftarrow A.$$

Variables, applications, projections, and annotated terms synthesize. Lambda terms, pairs, and proof terms usually check against expected types. This is especially important for dependent types, where full inference is undecidable or impractical.

The Spherepop checking algorithm can be described operationally as follows. First, parse a candidate history expression. Second, elaborate surface syntax into core syntax with explicit binders, universes, and annotations. Third, normalize types enough to expose their outer constructors. Fourth, check whether each continuation is admitted by the expected refusal structure. Fifth, collapse definitional equalities by normalization. Sixth, refuse the judgment if an admissibility boundary is crossed.

12 Repair Terms

Spherepop should also expose repair explicitly. A repair term records how an apparently blocked continuation can be transformed into an admissible one. If t fails to check against A, but a repair procedure ρ transforms it into t', and t' checks against A, then one may record:

$$\Gamma \vdash \rho : t \rightsquigarrow_A t'.$$

This is not ordinary definitional equality. It is proof-relevant repair. The original malformed or incomplete construction remains visible as history, but it is not allowed to collapse as if it were already correct.

A repair type may therefore be introduced:

$$\mathsf{Repair}_A(t, t')$$

whose inhabitants are admissible transformations from t to t' at type A. The formation rule is:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash t : \mathsf{Raw} \qquad \Gamma \vdash t' : A}{\Gamma \vdash \mathsf{Repair}_A(t, t') : \mathsf{Type}}.$$

A repair proof does not say that the broken term was valid all along. It says that there is an admissible path from the broken construction to a valid one.

This is where Spherepop differs most sharply from ordinary proof assistants. In many systems, invalid terms disappear from the formal world. In Spherepop, they may remain as refused histories, repair obligations, or deferred continuations. The calculus can therefore represent not only finished proofs but the ecology of proof development: failed attempts, blocked branches, partial constructions, and admissible repairs.

13 A Spherepop Calculus of Constructions

We can now state the core system.

Definition 13.1 (Spherepop Calculus of Constructions)**.** *The Spherepop Calculus of Constructions, written* SPCC*, is a dependent type theory with the following ingredients:*

$$\mathsf{Type}_0, \mathsf{Type}_1, \mathsf{Type}_2, \ldots$$

$$\Pi x : A.\, B(x), \qquad \Sigma x : A.\, B(x), \qquad \mathsf{Id}_A(a, b),$$

together with operational history forms

$$\mathsf{pop}, \mathsf{refuse}, \mathsf{collapse}, \mathsf{bind}$$

and a judgmental interpretation in which contexts are admissible histories, types are refusal structures, terms are admitted continuations, and reduction is admissible collapse.

The typing judgment

$$\Gamma \vdash t : A$$

means that the history Γ may be extended by t under the refusal discipline A. The reduction judgment

$$t \longrightarrow u$$

means that t collapses by one admissible computational step to u. The refusal judgment

$$\Gamma t : A$$

means not merely that t lacks a type, but that admitting t as an A-continuation would corrupt the history.

The central soundness theorem should have the following shape.

Theorem 13.2 (Preservation)**.** *If $\Gamma \vdash t : A$ and $t \longrightarrow u$, then $\Gamma \vdash u : A$.*

Proof. The proof proceeds by induction on the typing derivation. The essential case is application. If t is $(\lambda x.b)\, a$, then the typing derivation gives $\Gamma, x : A \vdash b : B(x)$ and $\Gamma \vdash a : A$. By substitution, $\Gamma \vdash b[a/x] : B(a)$. Since beta reduction sends $(\lambda x.b)\, a$ to $b[a/x]$, the type is preserved. The remaining cases follow from the compatibility of reduction with the typing rules. □

Theorem 13.3 (Progress relative to refusal)**.** *If $\varnothing \vdash t : A$, then either t is a value, or there exists u such that $t \longrightarrow u$, or t contains a suspended admissibility obligation explicitly marked as deferred.*

Proof. The proof proceeds by induction on the typing derivation. Canonical forms show that a closed term of function type is a lambda abstraction, a closed term of dependent sum type is a pair, and a closed proof of identity introduced directly is reflexivity. If a term is not already canonical, its typing derivation exposes a reducible subterm unless the derivation contains an explicit deferred obligation. Ill-formed continuations cannot appear because they would have been refused before the judgment was formed. □

This version of progress is deliberately weaker than the ordinary one because Spherepop allows deferred obligations. That is a feature, not a defect. Interactive proof development and repair-oriented programming both require open histories.

14 Implementation Strategy

A minimal implementation can be built in stages. The first stage defines a core abstract syntax for terms, types, contexts, and universes. The second implements substitution, alpha-equivalence, and normalization. The third implements bidirectional checking. The fourth adds dependent products and sums. The fifth adds identity types. The sixth adds explicit history events, refusal reasons, deferred obligations, and repair terms.

The core checker should never treat refusal as a normal runtime exception. A type error is not a value. It is a failed judgment. Therefore the checker should return either a derivation object or a refusal object:

$$\mathsf{check}(\Gamma, t, A) \in \mathsf{Derivation}(\Gamma \vdash t : A) + \mathsf{Refusal}.$$

This is the implementation-level version of the philosophical claim. Type checking does not discover whether an already valid program happens to run. It decides whether a proposed continuation may enter the history at all.

A suitable internal representation is:

$$\mathsf{RawTerm} \to \mathsf{ElaboratedTerm} \to \mathsf{TypedTerm} \to \mathsf{CollapsedTerm}.$$

Each arrow is partial. Failure at any stage produces a refusal reason. Repair may then act on refused or deferred terms, but repaired terms must re-enter the checker rather than bypass it.

To build a type calculus in Spherepop is to make refusal constructive. A type is not merely a label attached to a term. It is the operational boundary that determines whether a proposed continuation may become part of the accepted history. Dependent types generalize this boundary by allowing admissibility to depend on previous commitments. The calculus of constructions completes the picture by allowing propositions, proofs, programs, and types to live in one stratified universe of admissible construction.

The Spherepop interpretation clarifies why type theory belongs naturally inside a repair-theoretic language. Programs are histories under construction. Proofs are histories that cannot be refused by the propositions they inhabit. Type errors are inadmissible continuations. Normalization is collapse. Substitution is repair of an open dependency by an admitted witness. Universes are levels at which refusal structures themselves become constructible.

The result is not merely a typed programming language. It is a calculus of admissible continuation. In such a system, correctness is not imposed after computation. Correctness is the condition under which computation is allowed to become history.

15 Composition Before Typing

One of the central philosophical claims of this work is that composition is more fundamental than typing. Every computational system ultimately consists of primitive operations together with rules for composing them. Whether the primitive objects are logic gates, lambda terms, combinators, Lisp pairs, machine instructions, or Spherepop operations is largely incidental. What matters is that one operation may be connected to another to produce a larger computation.

The expressive power of a computational system therefore comes primarily from composition rather than classification. Once composition is available, arbitrarily complex constructions can be assembled recursively. Every parser, compiler, theorem prover, operating system, and neural network is ultimately a finite composition of simpler operations.

Type systems do not increase the expressive power of the underlying computational model. A simply typed lambda calculus, an untyped lambda calculus, a combinatory logic, and a Turing machine differ dramatically in their reasoning

principles and safety guarantees, but their computational foundations remain composition.

Spherepop therefore treats composition as ontologically prior. Histories are built by composing operational events. Typing is introduced later as a structured way of describing which histories should be regarded as admissible.

16 Types as Human Compression

From this perspective, a type is not a computational necessity but a semantic compression.

Suppose a programmer writes a function intended to consume integers. Nothing in the physical hardware requires the existence of an integer type. The processor simply receives bit patterns. The distinction between integers, floating-point numbers, characters, pointers, and functions exists because human programmers have found these categories useful for organizing programs.

The same phenomenon appears throughout mathematics. Categories, groups, manifolds, Hilbert spaces, and topological spaces do not create mathematical objects. They organize mathematical objects into classes that make proofs easier to formulate and communicate.

Type theory performs an analogous organizational role for computation.

Instead of reasoning about every possible execution history individually, the programmer reasons about entire families of histories simultaneously. The statement

$$f : A \to B$$

compresses an infinite collection of possible executions into a single semantic description.

The computer does not require this compression to execute the program. The programmer benefits from it because it makes large systems tractable.

Spherepop therefore interprets types as descriptions of histories rather than generators of histories. Histories remain primary.

17 Construction Without a Calculus of Constructions

The calculus of constructions is one of the most elegant formalisms ever developed for reasoning about programs and proofs. Nevertheless, it should not be mistaken for the source of computation itself.

A processor does not execute dependent products.

A transistor does not evaluate universes.

A logic gate has no awareness of propositions.

Instead, these physical systems repeatedly compose primitive operations.

The calculus of constructions provides an extraordinarily rich language for describing those compositions, verifying their correctness, and expressing higher-order abstractions. It is therefore a language of explanation rather than a prerequisite for execution.

Spherepop deliberately separates these roles.

The operational substrate consists only of history-producing operators such as

$$\mathsf{pop}, \qquad \mathsf{bind}, \qquad \mathsf{collapse}, \qquad \mathsf{refuse}.$$

Dependent types, universes, identity types, propositions-as-types, and proof objects may all be defined as higher-level structures over these primitive histories. They enrich the language of reasoning without enlarging the underlying computational capability.

18 Composition Trees and Lisp

Lisp illustrates this principle particularly clearly.

At its foundation, Lisp contains only atoms, pairs, and recursive composition through `cons`. Every list, tree, symbolic expression, abstract syntax tree, interpreter, compiler, and macro system emerges from repeated application of this single compositional mechanism.

The remarkable flexibility of Lisp comes not from an elaborate hierarchy of primitive types but from the uniformity of its composition rules.

Spherepop adopts a similar philosophy.

Instead of beginning with a large collection of primitive semantic objects, it begins with a small collection of primitive history operations. Rich semantic structures emerge recursively through repeated composition.

The resulting system remains conceptually small while supporting arbitrarily complex constructions.

19 Directed Acyclic Graphs as Sequential Circuits

The relationship between circuits and directed acyclic graphs is similarly clarified once composition is taken as primitive.

A combinational logic circuit is simply a graph whose vertices represent primitive operations and whose edges represent compositions between them.

A directed acyclic graph imposes one additional structural condition: every composition proceeds in a single direction without returning to an earlier stage.

This topological restriction guarantees that evaluation may proceed according to a topological ordering. Every node is evaluated only after all of its predecessors have been computed.

No explicit sequencing instructions are required because sequencing has become part of the graph itself.

A DAG may therefore be understood as a circuit whose sequential evaluation order is encoded geometrically.

Cycles fundamentally change the situation.

Once feedback edges exist, evaluation can no longer be determined solely from graph structure. Delay operators, fixed-point semantics, recursion, iterative convergence, or temporal state become necessary.

The distinction is therefore not merely graph-theoretic.

A DAG represents pure compositional flow.

A cyclic graph represents compositional flow together with persistent state.

This observation provides another motivation for Spherepop's history-first semantics. Histories naturally capture stateful computation without requiring the operational primitives themselves to become substantially more complicated.

20 Histories Before Types

These observations suggest an inversion of the traditional foundations of programming languages.

Rather than beginning with syntax, typing rules, proof objects, and semantic interpretations before finally reaching execution, Spherepop begins with execution itself.

Primitive operational events generate histories.

Histories compose into larger histories.

Patterns emerge within collections of histories.

Types summarize these patterns.

Proofs certify these summaries.

Universes organize the summaries into increasingly abstract levels.

The hierarchy therefore becomes

$$\text{composition} \rightarrow \text{histories} \rightarrow \text{patterns} \rightarrow \text{types} \rightarrow \text{proofs} \rightarrow \text{meta-theories}.$$

Nothing computationally essential is added after the first step.

Each successive layer exists to improve explanation, abstraction, verification, communication, and maintenance rather than to enlarge the class of computable functions.

This inversion is entirely consistent with the philosophy of Spherepop. Computation is the progressive construction of histories through admissible composition. Types, proofs, and logical systems are powerful descriptive languages built upon those histories rather than the source from which computation itself originates.

21 Simulating Fuzzy Logic Through Composition

One advantage of treating composition rather than types as primitive is that alternative logical systems require no fundamental changes to the execution model. Classical Boolean logic, fuzzy logic, probabilistic logic, paraconsistent logic, and many-valued logics differ primarily in the interpretation assigned to intermediate values, not in the mechanics of function composition itself.

A Spherepop evaluator therefore does not require a separate execution engine for fuzzy logic. The same composition engine may execute Boolean circuits, fuzzy circuits, arithmetic circuits, and probabilistic circuits simply by changing the primitive operators associated with each node.

Suppose a history contains primitive values

$$x_i \in [0, 1].$$

Instead of representing truth as the discrete set

$$\{0, 1\},$$

truth becomes a continuous degree of admissibility.

The computation itself is unchanged.

Nodes continue to receive inputs.

Primitive operators continue to produce outputs.

Histories continue to compose.

Only the interpretation of the primitive operators changes.

22 Generalizing Boolean Gates

Ordinary digital circuits define gates such as

$$\text{AND}(x, y), \qquad \text{OR}(x, y), \qquad \text{NOT}(x).$$

Fuzzy logic replaces these with continuous operators.
For example,

$$\text{AND}(x, y) = \min(x, y),$$

$$\text{OR}(x, y) = \max(x, y),$$

$$\text{NOT}(x) = 1 - x.$$

Many alternative choices are possible.
One may instead use the product t-norm

$$\text{AND}(x, y) = xy,$$

or the Lukasiewicz operator

$$\text{AND}(x, y) = \max(0, x + y - 1).$$

Nothing about the surrounding computation changes.
Each node still performs a deterministic function.
The circuit topology remains identical.
Only the local operator associated with each node has changed.
Spherepop therefore separates topology from semantics.
The graph determines how information flows.
The primitive operators determine how information transforms.

23 Circuits as Operator Networks

This viewpoint suggests representing every computation as an operator network.
Each node stores

$$(v_i, f_i),$$

where

$$v_i$$

is the current value and

$$f_i$$

is the local transformation.
Evaluation simply becomes

$$v_i = f_i(v_{j_1}, \ldots, v_{j_k}).$$

Neither Boolean logic nor fuzzy logic is fundamental.
Both correspond to different choices for the family

$$\{f_i\}.$$

One network might contain Boolean gates.
Another might contain differentiable functions.
Another might contain neural network activations.
Another might contain symbolic rewrite operators.
The composition engine remains unchanged.

24 Blending Logical Systems

Because every node merely applies a function, different logical systems may coexist within the same graph.

For example,
one region of a program might use Boolean operators,
another fuzzy operators,
another probabilistic operators,
and another ordinary arithmetic.
The outputs of one region become the inputs of another.

The graph does not need to know what ”kind” of logic is being used.

It only propagates values according to the local functions.

This makes logical systems modular rather than global.

Logic becomes a property of operators rather than of the execution engine.

25 Continuous Admissibility

Within Spherepop, fuzzy truth can be interpreted as continuous admissibility.

Instead of saying

$$x \in A$$

or

$$x \notin A,$$

one assigns an admissibility score

$$\alpha_A(x) \in [0, 1].$$

The score represents the degree to which a continuation satisfies the intended constraints.

Refusal therefore becomes gradual rather than binary.

Instead of

$$\mathsf{refuse}$$

or

$$\mathsf{pop},$$

one may define

$$\mathsf{refuse}_\lambda$$

where

$$\lambda \in [0, 1]$$

measures confidence or admissibility.

Hard refusals become the limiting case

$$\lambda = 0.$$

Complete acceptance becomes

$$\lambda = 1.$$

Intermediate values represent uncertain, partial, or repairable continuations.

This interpretation connects naturally with repair theory.

Repair no longer transforms only invalid histories into valid ones.

Instead, repair increases admissibility continuously until an operational threshold is reached.

26 Differentiable Circuits

An even more interesting consequence is that fuzzy circuits become differentiable whenever the primitive operators are differentiable.

Suppose each node computes

$$v_i = f_i(v_{j_1}, \ldots, v_{j_k}),$$

where every

$$f_i$$

is continuously differentiable.

The entire graph then defines a differentiable function

$$F : \mathbb{R}^n \to \mathbb{R}^m.$$

Gradients propagate through exactly the same composition graph.

Backpropagation is therefore not a fundamentally different computation.

It is another traversal of the same graph using different local operators.

The forward pass computes values.

The backward pass computes derivatives.

Both are compositions over the same underlying history.

27 Composition as the Universal Simulator

This illustrates a broader principle.

Once a language possesses sufficiently expressive function composition, almost every computational formalism can be simulated by changing only the primitive operators.

Boolean circuits become one operator family.

Fuzzy circuits become another.

Neural networks become another.

Probabilistic graphical models become another.

Constraint solvers become another.

Even symbolic theorem provers may be viewed as graphs whose nodes perform rewrite operations instead of arithmetic.

The composition engine itself remains remarkably small.

Its only responsibilities are to maintain the graph, propagate values, preserve evaluation order, and record histories.

Everything traditionally viewed as "the logic of the language" becomes merely a library of composable operators.

This is one of the strongest arguments for treating composition rather than typing as the true computational primitive. The same compositional substrate can faithfully simulate an enormous range of logical and computational systems, while types, proofs, and semantic disciplines remain optional layers that help humans organize and verify those computations rather than enabling them.

28 Parameterized Logical Operators

The preceding discussion suggests that Boolean logic and fuzzy logic should not be viewed as separate computational paradigms. They are merely different parameterizations of the same compositional substrate.

Let every logical operator belong to a parameterized family

$$F_\theta : [0,1]^n \to [0,1],$$

where the parameter θ determines the interpretation of conjunction, disjunction, implication, negation, or any other connective.

Boolean logic appears as one particular choice of θ.

The minimum t-norm appears as another.

The product t-norm appears as another.

The Lukasiewicz operators appear as another.

Nothing about the evaluation algorithm changes.

Only the operator tables are replaced.

Consequently, a Spherepop runtime need not contain a privileged notion of logical truth. It need only evaluate parameterized operators attached to nodes in the computation graph.

The distinction between logical systems therefore migrates from the execution engine to the operator library.

29 Logic as an Algebra of Operators

From this viewpoint, a logical system is simply an algebra generated by a collection of primitive operators.

Suppose

$$\mathcal{O} = \{f_1, f_2, \ldots, f_n\}$$

is a finite collection of functions.

The closure of these functions under composition generates an algebra

$$\langle \mathcal{O} \rangle.$$

Boolean logic corresponds to one generating set.

Fuzzy logic corresponds to another.

Arithmetic corresponds to another.

Linear algebra corresponds to another.

Symbolic rewriting corresponds to another.

The composition engine never changes.

Only the generating algebra changes.

This observation considerably simplifies language implementation.

Instead of designing a separate evaluator for every computational paradigm, one constructs a single composition engine capable of evaluating arbitrary operator graphs.

Every ”language” then becomes a particular choice of primitive operators.

30 Truth Values as Data

Traditional logic often gives the impression that truth values possess a privileged ontological status.

Operationally, they do not.

A truth value is simply another piece of data flowing through a graph.

Whether that datum happens to be

$$0,$$

$$1,$$

$$0.734,$$

or even an interval

$$[0.62, 0.81]$$

is irrelevant to the composition mechanism itself.

Each node receives data.

Each node computes another datum.

Composition proceeds.

Seen this way, Boolean logic is simply the special case in which the carrier set contains only two values.

Fuzzy logic enlarges the carrier set.

Probabilistic reasoning enlarges it further by allowing distributions.

Interval reasoning replaces numbers with sets.

Nothing fundamental changes computationally.

31 Learning New Logics

Because logical operators are ordinary functions, they need not be fixed in advance.

Suppose each primitive operator possesses adjustable parameters

$$f(x; \theta).$$

Instead of selecting the operator manually, one may optimize

$$\theta$$

from observations.

The system therefore learns its own logical connectives.

For example, conjunction may gradually evolve from

$$\min(x, y)$$

toward

$$xy$$

or toward some entirely different differentiable operator that better models empirical data.

Traditional fuzzy systems usually choose a particular t-norm before execution begins.

A compositional system may instead discover one.

This transforms logic itself into a learnable object.

The computation graph remains fixed.

Only the local operators evolve.

32 Sequential Evaluation Without Logical Commitment

One consequence of the history-first viewpoint is that evaluation need not imply logical commitment.

A node may compute

$$v = 0.43$$

without asserting that any proposition has truth value 0.43.

The value merely represents an intermediate state within a larger computation.

Logical interpretation may be postponed until much later.

Indeed, the same numerical history may receive multiple semantic interpretations.

One observer may interpret

0.43

as a fuzzy truth value.

Another may interpret it as a confidence estimate.

Another as a probability.

Another as an admissibility score.

Another as available computational resources.

The execution history is identical.

[OOnly the interpretation changes.

Spherepop therefore distinguishes operational histories from semantic interpretations.

Histories are objective computational artifacts.

Logical meanings are attached afterward.

33 A Universal Operator Machine

The discussion naturally suggests a minimal abstract machine.

Each node stores

$$(\text{operator}, \text{inputs}, \text{outputs}, \text{history}).$$

Execution consists only of repeatedly selecting a node whose predecessors have completed and evaluating its associated operator.

Formally,

$$v_i = f_i(v_{j_1}, \ldots, v_{j_k}).$$

No distinction is made between

Boolean operators,

arithmetic,

symbolic rewriting,

string manipulation,

matrix multiplication,

probabilistic inference,

or fuzzy logic.

Every one of them is simply another operator.

The machine therefore becomes universal in a stronger sense than a conventional virtual machine.

It does not execute instructions belonging to one language.

It executes arbitrary operator compositions.

Programming languages become surface syntaxes describing operator graphs.

Logic becomes an operator library.

Types become annotations over graphs.

Proofs become certificates that certain graph properties hold.

Optimization becomes graph rewriting.

Compilation becomes graph transformation.

Execution becomes graph traversal.

All of these become different views of the same underlying compositional object.

34 Composition as the Primitive Ontology

The resulting ontology is strikingly economical.

Primitive operators exist.

Operators compose into graphs.

Graphs generate histories.

Histories accumulate observations.

Semantic systems classify histories.

Logical systems constrain histories.

Proof systems certify histories.

Type systems summarize histories.

Everything above composition is descriptive rather than generative.

This inversion explains why so many apparently different computational paradigms can simulate one another. They all share the same compositional foundation. Their differences arise primarily from the choice of primitive operators, the interpretation assigned to intermediate values, and the constraints imposed upon admissible compositions.

Spherepop therefore places composition at the bottom of its hierarchy. Every richer logical system—including Boolean logic, fuzzy logic, probabilistic reasoning, dependent type theory, and the calculus of constructions—may be understood as

successive semantic enrichments of an underlying algebra of composable histories rather than as fundamentally distinct computational mechanisms.

35 Logical Gates as Ordinary Functions

A useful consequence of the compositional viewpoint is that logical gates cease to occupy a privileged position. An AND gate is not fundamentally different from addition, matrix multiplication, or string concatenation. Each is simply a function with a specified arity.

Formally, let

$$f : X_1 \times X_2 \times \cdots \times X_n \to Y$$

be any computable function. A node in a Spherepop graph merely stores the function symbol together with references to the histories supplying its arguments.

Evaluation consists of replacing each node by the result of applying its associated function to the outputs of its predecessors.

Consequently, an AND gate is merely

$$f_{\text{AND}} : \{0,1\}^2 \to \{0,1\},$$

while multiplication is

$$f_{\times} : \mathbb{R}^2 \to \mathbb{R},$$

and a fuzzy conjunction is

$$f_{\wedge} : [0,1]^2 \to [0,1].$$

All three are identical from the perspective of the execution engine. They differ only by the local function attached to the node.

This observation removes the distinction between "logic circuits" and "computational graphs." Every circuit is a computational graph whose nodes happen to implement a particular collection of primitive operators.

36 Truth Tables as Lookup Functions

Even classical truth tables need not be regarded as logical primitives.

A truth table is simply a finite function.

For example,

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

is merely a lookup table implementing

$$f_{\wedge}(x, y).$$

The execution engine never reasons about conjunction itself.

It performs a table lookup.

Likewise, fuzzy conjunction may be implemented by a floating-point function, an interpolation table, a polynomial approximation, or even a trained neural network.

The surrounding graph remains unchanged.

Thus the distinction between symbolic logic and numerical computation largely disappears. Both become methods for implementing local operators.

37 Higher-Order Operators

Because operators are themselves ordinary values, they may be composed exactly as data are composed.

Suppose

$$f, g, h$$

are operators.

One may define another operator

$$H(f, g) = f \circ g,$$

which constructs a new operator by composition.

Similarly,

$$M(f, \theta)$$

may produce a modified version of

$$f$$

parameterized by

$$\theta.$$

This allows programs to manipulate their own logical connectives.

For example, a learning algorithm may receive an operator implementing conjunction, optimize its parameters, and return an improved conjunction.

Nothing special is required from the language.

Operators are simply first-class computational objects.

This is closely related to the way Lisp treats functions as ordinary values, except that Spherepop naturally extends the idea to arbitrary operator graphs rather than merely symbolic expressions.

38 Building Fuzzy Systems from Primitive Composition

A complete fuzzy inference engine may therefore be assembled from surprisingly few primitives.

First, a collection of membership functions converts raw observations into degrees of membership.

For each linguistic category

$$A_i,$$

one computes

$$\mu_{A_i}(x) \in [0, 1].$$

Next, rule nodes compose these membership values using fuzzy conjunctions and disjunctions.

A rule such as

IF temperature is hot AND humidity is high

becomes another operator node receiving two numerical inputs and producing one numerical output.

Rule outputs are then aggregated,

perhaps by maximum,

perhaps by summation,

perhaps by another learned operator.

Finally, a defuzzification operator maps the resulting fuzzy set back into an executable numerical decision.

At no point does the execution engine require special knowledge of fuzzy logic.

Every stage consists only of evaluating ordinary functions arranged into a directed graph.

39 Replacing Hard Branches with Continuous Composition

Traditional programming languages often rely upon hard branching.

One writes

$$\texttt{if } x > 0 \texttt{ then } A \texttt{ else } B.$$

Execution follows exactly one path.

Fuzzy computation allows a different interpretation.

Instead of choosing a single branch, one computes degrees of participation.

Suppose

$$\alpha \in [0, 1]$$

measures confidence in branch A.

The complementary confidence

$$1 - \alpha$$

weights branch B.

The output becomes

$$y = \alpha A(x) + (1 - \alpha) B(x).$$

The graph no longer forks into mutually exclusive histories.

Instead, both histories contribute continuously.

Only at a later stage, if necessary, does one collapse the computation into a discrete decision.

From the perspective of Spherepop this is particularly attractive because collapse becomes an explicit operation rather than an implicit consequence of branching syntax.

40 Histories Before Decisions

This suggests interpreting fuzzy computation as delayed commitment.

Rather than deciding immediately whether a proposition is true or false, the history records progressively accumulating evidence.

Each operator transforms the current evidence into richer evidence.

Only after sufficient information has accumulated does a collapse occur.

Operationally,

```
pop
```

opens possibilities,

composition propagates evidence,

repair adjusts inconsistencies,

and

```
collapse
```

commits to a final decision.

Boolean logic appears as the limiting case in which collapse occurs immediately after every operation.

Fuzzy logic postpones collapse until the end of a much longer computational history.

In this sense, fuzzy reasoning is not fundamentally a different logic. It is a different scheduling policy for commitment.

The underlying compositional machinery is identical.

41 A Spectrum of Computational Semantics

The same operator graph may therefore support many semantic interpretations simultaneously.

One execution might interpret intermediate values as Boolean truth values.

Another as fuzzy memberships.

Another as probabilities.

Another as confidence scores.

Another as admissibility measures.

Another as repair potentials.

Another as available computational resources.

The graph computes exactly the same sequence of compositions in every case.

Only the semantic interpretation of the values changes.

This reinforces the central philosophical claim of Spherepop. Composition is the invariant computational substrate. Logical systems, type systems, proof systems, and semantic theories are different coordinate systems placed upon the same underlying histories of composed operations.

42 Formal Operational Semantics of Spherepop

The philosophical position developed throughout this essay is that composition is the primitive computational act. The purpose of this section is to make that claim mathematically precise. Rather than beginning with types, terms, or syntactic judgments, we begin with the smallest possible operational substrate and derive increasingly rich computational structures from it.

The guiding design principle is simplicity. Every execution should ultimately be reducible to repeated applications of primitive operators acting on histories. The execution engine should not distinguish between Boolean reasoning, arithmetic, symbolic rewriting, fuzzy inference, neural computation, parsing, or proof construction. These are all regarded as different interpretations of the same underlying compositional process.

42.1 Primitive Computational Objects

Let

$$\mathcal{V}$$

denote the universe of computational values.

No assumptions are made about the internal structure of these values. They may represent

integers,
floating point numbers,
strings,
graphs,
proof objects,
operators,
circuits,
or entire programs.

The execution engine treats every member of

$$\mathcal{V}$$

uniformly.

Likewise, let

$$\mathcal{O}$$

denote the collection of primitive operators.

Each operator possesses finite arity

$$n \geq 0$$

and is simply a computable mapping

$$f : \mathcal{V}^n \to \mathcal{V}.$$

Nothing in this definition refers to logical truth, typing, syntax, or semantics.

An operator merely consumes values and produces another value.

This observation is considerably more general than the traditional notion of instruction execution.

An instruction is merely one particular operator.

A logical gate is another.
A parser combinator is another.
A neural activation function is another.
A theorem-proving rewrite rule is another.
The runtime therefore requires only one notion of execution.
Every computational step consists of evaluating an operator.

42.2 Composition Graphs

Programs are represented as finite directed graphs

$$G = (N, E),$$

where

$$N$$

is the collection of operator nodes and

$$E$$

records the flow of values between them.
Each node

$$n \in N$$

contains

$$n = (f, I, O),$$

where

$$f \in \mathcal{O}$$

is the operator,

$$I$$

is the ordered collection of incoming edges,
and

$$O$$

is the collection of outgoing edges.

Execution never manipulates source code directly.

Instead, execution traverses this operator graph.

Surface syntax exists only as a convenient method for constructing graphs.

The runtime itself need never know whether a graph originated from

Lisp,

Spherepop,

C,

Python,

a hardware netlist,

a theorem prover,

or a visual programming language.

All are reduced to the same operational representation.

42.3 Histories

The graph describes possible computation.

A history records actual computation.

Formally, let

$$\mathsf{Hist}$$

denote the collection of finite execution histories.

Each history is an ordered sequence

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

where every

$$e_i$$

records one primitive computational event.

Rather than storing only produced values, histories preserve the operational structure of computation itself.

Typical events include

$$\mathsf{pop}(v),$$

opening a new computational object,

$$\mathsf{bind}(f),$$

constructing a new composition,

$$\mathsf{collapse}(v),$$

committing an evaluation,
and

$$\mathsf{refuse}(r),$$

rejecting an attempted continuation.

Additional events may be introduced without changing the execution engine because histories merely accumulate records.

They do not prescribe the interpretation of those records.

42.4 The Evaluation Relation

Evaluation is defined as a transition relation

$$\Longrightarrow$$

on histories.
If

$$h$$

is the current history,
then

$$h \Longrightarrow h'$$

means that one primitive computational event has occurred.

Repeated execution produces

$$h_0 \Longrightarrow h_1 \Longrightarrow h_2 \Longrightarrow \cdots \Longrightarrow h_n.$$

Unlike conventional operational semantics, the history is never discarded.

Every computational event remains available for later inspection, optimization, repair, explanation, or proof generation.

Execution therefore constructs a persistent operational record rather than merely producing a final value.

This reflects the central philosophy of Spherepop that histories are primary objects rather than temporary artifacts.

42.5 Node Evaluation

Suppose a node

$$n$$

contains operator

$$f$$

and predecessors

$$p_1, \ldots, p_k.$$

If all predecessor values have already been computed,

the node becomes executable.

Its output is defined by

$$v = f(v_1, \ldots, v_k),$$

where

$$v_i$$

is the value produced by predecessor

$$p_i.$$

The history extends by appending

$$\mathsf{collapse}(v),$$

or by recording a richer evaluation event if additional provenance information is desired.

Notice that this rule is completely independent of the nature of

$$f.$$

The runtime never distinguishes arithmetic operators from logical operators.

Every primitive computation is evaluated by exactly the same rule.

The only variation lies inside the local operator itself.

42.6 Readiness

Not every node is executable at every moment.

A node becomes ready precisely when all of its dependencies have been satisfied.

Define

$$\mathrm{Pred}(n)$$

to be the predecessors of node

$$n.$$

Then

$$\mathrm{Ready}(n, h)$$

holds whenever every predecessor has already produced an output recorded in the history

$$h.$$

The scheduler therefore requires no semantic understanding of the computation.

It merely determines which nodes have become operationally available.

Execution order emerges from graph structure rather than from programming-language syntax.

In the special case where the graph is acyclic, readiness induces a topological ordering.

For cyclic graphs, additional mechanisms such as delay operators or fixed-point iteration will be introduced in a later section.

42.7 Composition as the Only Primitive Rule

The semantics developed here deliberately avoids introducing separate evaluation rules for every computational paradigm.

Instead, there exists only one primitive computational principle.

A node may execute whenever its inputs have become available.

Everything traditionally regarded as computation—

Boolean reasoning,

symbolic manipulation,

numeric calculation,

type checking,

proof verification,

parsing,

optimization,

or fuzzy inference—

reduces to repeated application of this single rule.

The execution engine therefore contains remarkably little domain-specific knowledge.

It performs composition.

Every richer computational phenomenon emerges from the choice of primitive operators and the topology of the composition graph rather than from specialized execution mechanisms.

42.8 Composition as Universal Evaluation

The preceding semantics may appear almost trivial. This simplicity is intentional. The objective is not to build a specialized execution engine for each style of computation, but to identify the smallest operational core capable of expressing all of them.

Suppose that every primitive operator belongs to the collection

$$\mathcal{O} = \{f_1, f_2, \ldots\}.$$

Execution then consists entirely of repeated applications of the rule

$$(v_1, \ldots, v_n) \overset{f}{\longmapsto} f(v_1, \ldots, v_n).$$

The runtime neither knows nor cares what the operator represents.

If

$$f$$

implements floating-point multiplication, multiplication occurs.

If it implements a Boolean conjunction, logical inference occurs.

If it implements a parser combinator, parsing occurs.

If it implements a graph rewrite, symbolic computation occurs.

If it implements a neural activation function, machine learning occurs.

The execution mechanism is therefore invariant under changes of computational interpretation.

This invariance is one of the principal motivations for treating composition rather than syntax as the primitive computational object.

42.9 Operator Libraries

Rather than viewing programming languages as collections of syntactic constructs, Spherepop views them as collections of operator libraries.

Formally, let

$$\mathcal{L} \subseteq \mathcal{O}$$

be a finite operator library.

Different computational paradigms correspond to different choices of

$$\mathcal{L}.$$

For example,

$$\mathcal{L}_{\text{Bool}} = \{\text{AND}, \text{OR}, \text{NOT}, \text{XOR}\}$$

generates ordinary Boolean circuits.

Likewise,

$$\mathcal{L}_{\text{Fuzzy}} = \{\min, \max, 1 - x\}$$

generates one family of fuzzy inference systems.

An arithmetic library might contain

$$+, -, \times, \div, \sin, \exp .$$

A symbolic rewriting system contains rewrite operators.

A theorem prover contains inference operators.

A neural network contains affine transformations and nonlinear activation operators.

The execution engine remains identical.

Only the selected library changes.

Programming languages therefore become standardized methods for organizing operator libraries rather than fundamentally different execution models.

42.10 The Independence of Topology and Semantics

An important consequence of this formulation is the separation of graph topology from operator semantics.

The graph specifies only

$$\text{who depends upon whom.}$$

The operators specify only

$$\text{how values are transformed.}$$

These two structures are mathematically independent.

One may replace every Boolean gate in a circuit with fuzzy operators while leaving the graph completely unchanged.

One may replace arithmetic operators with symbolic operators.

One may replace deterministic operators with probabilistic operators.

In every case the topology is preserved.

This separation allows the same computational graph to be interpreted under multiple semantic regimes.

Indeed, one may even execute the same graph repeatedly under different operator libraries to compare different computational interpretations.

The graph therefore captures the structural organization of the computation, while the operators determine its semantic behavior.

42.11 Composition and Universality

The classical Church–Turing thesis identifies many apparently different models of computation as computationally equivalent.

Lambda calculus,

Turing machines,

recursive functions,

Post systems,

and combinatory logic

all compute the same class of functions.

Spherepop suggests a complementary perspective.

Rather than emphasizing equivalent formal systems, it emphasizes a common operational substrate.

Every sufficiently expressive computational model possesses

a collection of primitive operators,

a method of composing them,

and an execution strategy.

The details differ enormously.

The underlying pattern does not.

Consequently, universality may be viewed less as a property of any particular formalism than as a consequence of unrestricted composition.

Whenever a language permits sufficiently general operator composition, universal computation becomes possible regardless of whether the surface language resembles assembly code, Lisp, lambda calculus, Prolog, hardware description languages, or visual data-flow programming.

42.12 Persistent Histories

Traditional operational semantics often discard intermediate computations after producing the final result.

Spherepop deliberately rejects this assumption.

Every execution event becomes part of a persistent history.

Suppose

$$h_n = (e_1, e_2, \ldots, e_n).$$

Rather than replacing

$$h_n$$

by

$$h_{n+1},$$

Spherepop regards

$$h_{n+1} = (e_1, e_2, \ldots, e_n, e_{n+1})$$

as an extension of the previous history.

Nothing is forgotten.

This persistent representation supports several important capabilities.

First, explanations become straightforward because every computational decision remains available for inspection.

Second, repair becomes natural because alternative histories may branch from any previous point without reconstructing lost information.

Third, optimization may analyze historical execution patterns rather than only current program state.

Finally, proof generation becomes possible because derivations are recorded directly rather than reconstructed retrospectively.

Histories therefore function simultaneously as execution traces, provenance records, debugging information, proof objects, and optimization resources.

42.13 Execution Without Commitment

One of the more subtle consequences of persistent histories is the distinction between evaluation and commitment.

Ordinary programming languages frequently identify these concepts.

Once an expression is evaluated, the computation proceeds as though the result were final.

Spherepop separates these stages.

Evaluation produces information.

Commitment records that information as operationally authoritative.

Between these stages, additional operators may inspect,

repair,

transform,

annotate,

or even reject the computed value.

Only after these possibilities have been exhausted does a

collapse

event permanently commit the result.

This distinction becomes particularly useful for symbolic reasoning, theorem proving, fuzzy inference, and interactive computation, where tentative results often require further refinement before becoming final.

Execution therefore becomes a process of progressively increasing commitment rather than a sequence of irreversible state transitions.

42.14 The Minimal Spherepop Machine

The operational semantics developed in this section may be summarized by an extremely small abstract machine.

The machine consists of

$$(G, H, S),$$

where

$$G$$

is the composition graph,

$$H$$

is the persistent execution history,

and

S

is a scheduler selecting executable nodes.

Each iteration performs exactly four steps.

The scheduler selects a ready node.

The associated operator is evaluated.

The resulting event is appended to the history.

Successor nodes whose dependencies have become satisfied are added to the scheduler.

No additional computational machinery is fundamentally required.

Everything traditionally associated with programming languages—control flow, logical inference, symbolic rewriting, parsing, type checking, proof construction, optimization, fuzzy reasoning, neural computation, and arithmetic—appears only through the operators attached to graph nodes and the histories accumulated during execution.

The remarkable economy of this machine illustrates the central thesis of this essay. Composition is not merely one computational principle among many. It is the primitive from which the others may be systematically reconstructed.

42.15 Control Flow as Graph Structure

Conventional programming languages devote considerable machinery to describing control flow. Sequential execution, conditional branching, iteration, recursion, exceptions, and function calls are usually introduced as distinct language constructs, each requiring its own operational semantics.

From the perspective developed here, these distinctions are largely superficial.

Control flow is not a primitive computational object.

It is a geometric property of the composition graph.

A sequence of instructions is simply a path through the graph.

A conditional branch is simply a node possessing multiple outgoing continuations.

A loop is simply a directed cycle.

A function call is a subgraph whose outputs become inputs elsewhere.

Parallel execution corresponds to disconnected or independent subgraphs whose dependencies do not overlap.

The scheduler therefore never executes statements.

It traverses graph topology.

This greatly reduces the conceptual complexity of the execution model. What appear syntactically as many different control structures become special cases of graph traversal.

42.16 Branching Without Special Syntax

Consider the familiar conditional

$$\texttt{if } c \texttt{ then } A \texttt{ else } B.$$

Traditional operational semantics introduces separate inference rules for evaluating the condition, selecting one branch, and discarding the other.

Spherepop requires no such primitive.

Instead, the graph contains

a node producing the condition,

two candidate continuation subgraphs,

and a selection operator

$$\text{Select}(c, A, B).$$

The selection operator is itself merely another primitive function.

For Boolean computation,

$$\text{Select}(1, A, B) = A,$$

and

$$\text{Select}(0, A, B) = B.$$

For fuzzy computation,

$$\text{Select}$$

may evaluate both branches simultaneously and combine their outputs continuously.

For probabilistic computation,

the same operator may sample one branch according to a distribution.

For symbolic computation, [I it may preserve both branches as unevaluated possibilities.

Nothing about the graph changes.

Only the semantics of the local operator changes.

Thus conditional execution is not fundamentally different from arithmetic addition or logical conjunction.

It is another operator.

42.17 Iteration as Feedback

Iteration is traditionally presented as a primitive language feature.

Statements such as

```
while,
```

```
for,
```

and

```
repeat
```

appear to require their own semantic machinery.

Graphically, however, each represents the same structural phenomenon.

The output of a computation eventually returns to one of its own inputs.

Iteration is therefore feedback.

The distinction between iteration and recursion becomes largely syntactic.

Both produce cyclic dependency structures.

Execution of such graphs requires only that the scheduler understand delayed dependencies or fixed-point evaluation.

No fundamentally new computational mechanism has appeared.

The graph has merely ceased to be acyclic.

42.18 Functions as Graph Fragments

Functions likewise admit a simple graph-theoretic interpretation.

A function is not fundamentally a block of syntax.

It is a reusable graph fragment.

Its formal parameters correspond to designated input vertices.

Its return values correspond to designated output vertices.

Invoking the function does not require a special execution rule.

One simply connects the outputs of one graph to the inputs of another.

Function composition therefore becomes ordinary graph composition.

Inlining corresponds to graph substitution.

Partial evaluation corresponds to replacing portions of the graph by already-computed constants.

Memoization corresponds to remembering previously evaluated subgraphs.

Again, these techniques require no language-specific semantics.

They are graph transformations.

42.19 Programs as Persistent Graph Objects

An important consequence of this formulation is that programs themselves become ordinary computational values.

A graph may construct another graph.

A graph may modify itself.

A graph may generate specialized versions of itself.

A graph may optimize another graph.

This is difficult to express naturally when programs are regarded primarily as syntactic documents.

It becomes almost unavoidable once programs are represented directly as graph objects.

Suppose

$$G_1$$

is an optimizer and

$$G_2$$

is a target program.

Execution of

$$G_1(G_2)$$

produces another graph

$$G_2'.$$

The optimizer is therefore not a separate meta-language.

It is another operator graph acting upon graph values.

This unifies interpretation,

compilation,

optimization,

program transformation,

and code generation.

Each becomes computation over graph objects.

42.20 Surface Syntax as Serialization

The graph-first viewpoint also changes the role of programming language syntax.

Conventionally, syntax is regarded as the primary representation of a program.

The compiler then attempts to discover its computational structure.

Spherepop reverses this relationship.

The computational graph is primary.

Source code is merely one serialization of that graph.

Visual node editors provide another serialization.

Abstract syntax trees provide another.

S-expressions provide another.

Binary executable formats provide another.

All represent the same underlying compositional object.

This inversion explains why programs can often be translated between radically different syntactic forms while preserving identical computational behavior.

The syntax changes.

The graph does not.

Consequently, parsers become translators between external representations and internal graph objects rather than creators of computational structure.

The structure already exists.

The parser merely recovers it.

42.21 Toward Graph Normal Forms

Once computation is represented entirely by composition graphs, many apparently different programs become visibly identical.

Variable names disappear.

Formatting disappears.

Syntactic sugar disappears.

Evaluation order becomes explicit.

Dependency structure becomes explicit.

This suggests the existence of graph normal forms.

Two programs are operationally equivalent whenever they reduce to isomorphic normalized composition graphs together with equivalent operator libraries.

Such a characterization is substantially more semantic than ordinary textual equality.

It identifies computations by what they do rather than by how they were originally written.

The next section develops this idea further by showing that compilation itself can be understood as a sequence of graph rewriting operations. Under this interpretation, optimization is not a mysterious collection of compiler heuristics but the systematic transformation of one composition graph into another that computes the same history more efficiently.

43 Compilation as Graph Rewriting

The graph-oriented semantics developed in the previous section suggests a fundamentally different view of compilation. Conventionally, a compiler is described as a sequence of syntactic transformations. Source code is parsed into an abstract syntax tree, transformed into one or more intermediate representations, optimized, and finally translated into machine instructions.

From the perspective of Spherepop, these intermediate stages are convenient engineering choices rather than distinct computational principles. The compiler is more naturally understood as a graph rewriting system. Every phase of compilation preserves, approximates, or refines a composition graph while changing its representation.

The central object is therefore not the syntax tree but the underlying dependency

graph. Different intermediate representations merely expose different structural properties of that graph.

43.1 Programs as Rewrite Objects

Let

$$G = (N, E)$$

be a composition graph.

Compilation is defined as the application of a finite sequence of rewrite operators

$$R_1, R_2, \ldots, R_k,$$

producing

$$G \Longrightarrow R_1(G) \Longrightarrow R_2(R_1(G)) \Longrightarrow \cdots \Longrightarrow R_k \cdots R_1(G).$$

Each rewrite transforms one graph into another.

Importantly, the graph itself remains the object being manipulated.

The compiler does not manipulate source text except during parsing.

After parsing, every optimization, lowering step, specialization, or code generation phase becomes a graph transformation.

This viewpoint unifies a remarkable variety of compiler techniques that are often presented separately.

43.2 Correctness of Rewriting

A rewrite is meaningful only if it preserves the intended computation.

Let

$$\mathcal{H}(G)$$

denote the collection of observable histories generated by executing graph

$$G.$$

A rewrite

$$R$$

is operationally correct whenever

$$\mathcal{H}(R(G)) \cong \mathcal{H}(G),$$

where

$$\cong$$

denotes observational equivalence.

The exact definition of observational equivalence depends upon the execution model.

For deterministic computation it may require identical outputs.

For probabilistic computation it may require identical output distributions.

For symbolic computation it may require equivalent normal forms.

For repair-aware computation it may require equivalent admissible histories.

The rewrite engine therefore need not understand the internal meaning of every operator.

It need only preserve the operational behavior of the graph under the chosen equivalence relation.

43.3 Inlining as Graph Substitution

Function inlining is one of the simplest graph rewrites.

Suppose a node

$$f$$

invokes a function represented by subgraph

$$G_f.$$

Inlining removes the call node and substitutes

$$G_f$$

directly into the surrounding graph.

Formally,

$$G = C[f]$$

becomes

$$C[G_f],$$

where

$$C$$

denotes the surrounding graph context.

No semantic rule for function calls is required.

Inlining is simply graph substitution.

The distinction between ”calling” a function and ”copying” a graph fragment becomes an implementation choice rather than a conceptual one.

43.4 Dead Graph Elimination

Many programs contain computations whose results never influence any observable output.

Graphically, these correspond to nodes whose outputs are unreachable from the designated output vertices.

Define

$$\text{Reach}(G)$$

to be the set of nodes reachable from the outputs by traversing dependency edges in reverse.

Every node outside

$$\text{Reach}(G)$$

may be removed without affecting observable behavior.

This generalizes dead code elimination.

The compiler does not remove textual statements.

It removes disconnected portions of the composition graph.

The rewrite is therefore entirely structural.

43.5 Constant Propagation

Suppose an operator receives only constant inputs.

Its output may be computed during compilation rather than execution.

Graphically,

$$f(c_1, \ldots, c_n)$$

is replaced by a single constant node

$$c.$$

The surrounding graph remains unchanged.

Only one local region has been simplified.

Repeated application of this rewrite gradually collapses large portions of the graph into constants.

This process may itself expose additional opportunities for rewriting.

Constant propagation is therefore naturally viewed as graph reduction rather than symbolic evaluation.

43.6 Common Subgraph Elimination

Duplicate computation appears as repeated isomorphic subgraphs.

Suppose two regions compute

$$f(g(x), h(y)).$$

Rather than evaluating both independently, the compiler may construct the shared subgraph once and redirect all consumers to its output.

Graphically,

two copies become one shared node.

Unlike textual common subexpression elimination, this formulation makes sharing explicit.

The optimization is simply graph merging.

Nothing about the execution semantics changes.

The graph has become smaller while preserving the same observable histories.

43.7 Graph Rewriting as the Compiler's Primitive

These examples illustrate a recurring theme.

Inlining,

constant propagation,

dead code elimination,

common subexpression elimination,

partial evaluation,

strength reduction,

loop invariant motion,

and many other optimizations differ only in the local graph rewrite being applied.

The compiler therefore requires only three conceptual ingredients.

First, it must recognize graph patterns.

Second, it must replace one pattern by another.

Third, it must verify that the replacement preserves observational equivalence.

Everything else follows from repeated application of these three principles.

This greatly simplifies the conceptual architecture of compilation.

Instead of memorizing dozens of unrelated optimization techniques, one studies a single algebra of graph transformations whose individual rewrite rules happen to produce familiar compiler optimizations as special cases.

43.8 Compilation as Progressive Refinement

The traditional compiler pipeline is often presented as a sequence of unrelated phases.

One begins with lexical analysis,

then parsing,

then semantic analysis,

then optimization,

then instruction selection,

then register allocation,

and finally code generation.

While this organization is useful from an engineering perspective, it obscures a deeper mathematical unity.

Every stage is simply another graph transformation.

The graph produced by the parser is generally very close to the source language.

The graph produced by the optimizer is computationally simpler.

The graph produced by the backend is closer to the target architecture.

Nevertheless, each graph represents essentially the same computation.

Compilation therefore becomes a process of progressive refinement.

Each rewrite removes accidental structure while preserving operational structure.

The graph evolves toward forms that are increasingly efficient for the chosen execution substrate.

The underlying computation remains invariant.

43.9 Multiple Intermediate Representations

Modern compilers frequently employ numerous intermediate representations.

One representation may emphasize control flow.

Another data flow.

Another static single assignment.

Another machine scheduling.

Spherepop interprets these not as different computational models but as different coordinate systems on the same graph.

Each representation preserves certain structural information while suppressing other information.

An abstract syntax tree emphasizes syntactic hierarchy.

A control-flow graph emphasizes possible execution paths.

A dependence graph emphasizes producer-consumer relationships.

A hardware netlist emphasizes physical connectivity.

An SSA graph emphasizes unique value production.

The compiler continually changes representations because different optimization problems become easier in different coordinate systems.

The computation itself never changes.

Only the chosen description changes.

This is analogous to geometry.

Cartesian coordinates,

polar coordinates,

and homogeneous coordinates describe the same geometric object while making different calculations convenient.

Intermediate representations play precisely the same role for computation.

43.10 Instruction Selection as Operator Replacement

Eventually every compiler must target a concrete execution device.

Conventionally this stage is called instruction selection.

Graphically, however, instruction selection is simply another rewrite.

Suppose a source graph contains

$$f(x, y).$$

If the target processor contains a native instruction implementing

$$f,$$

the graph node is relabeled with the corresponding machine operator.

If no direct instruction exists,

the node is replaced by a subgraph implementing the same computation.

For example,

integer division on one processor may correspond to a single machine instruction,

while on another processor it expands into a sequence of shifts,

subtractions,

comparisons,

and branches.

The surrounding graph is unchanged.

Only the local implementation differs.

Instruction selection therefore becomes operator substitution.

This interpretation applies equally well to CPUs,

GPUs,

FPGAs,

neural accelerators,

and distributed execution engines.

Every backend simply provides a different operator library.

43.11 Compilation Across Computational Paradigms

One of the most attractive consequences of the graph formulation is that compilation between apparently unrelated paradigms becomes conceptually straightforward.

A Boolean circuit may be rewritten into an arithmetic circuit.

An arithmetic circuit may be rewritten into tensor operations.

Tensor operations may be rewritten into GPU kernels.

GPU kernels may be rewritten into hardware netlists.

Likewise,

a symbolic rewrite system may be translated into a data-flow graph,

or a functional program into a hardware circuit.

These transformations are often presented as fundamentally different compiler problems.

Spherepop instead regards them as graph homomorphisms between operator libraries.

The graph topology is preserved as much as possible.

The local operators are translated into equivalent operators available on the target substrate.

Thus compilation becomes independent of programming language.

It is instead a translation between computational substrates.

43.12 Optimization by Local Equivalence

Many compiler optimizations derive from surprisingly small local identities.

For example,

$$x + 0 = x,$$

$$x \times 1 = x,$$

$$x \wedge 1 = x,$$

$$x \vee 0 = x.$$

Each identity replaces one graph fragment with a smaller graph fragment.

Nothing about the surrounding computation changes.

Repeated application of local equivalences gradually simplifies the entire graph.

This observation suggests viewing optimization as analogous to algebraic simplification.

Instead of simplifying equations,

the compiler simplifies composition graphs.

The rewrite rules become computational identities.

Optimization becomes the search for shorter or more efficient representatives of an operational equivalence class.

43.13 Self-Optimizing Programs

Once programs are represented explicitly as graph objects,

optimization need no longer remain exclusively a compile-time activity.

Programs may inspect their own graphs,

measure execution histories,

identify expensive regions,

and rewrite themselves dynamically.

A frequently executed subgraph may be specialized.

A branch that is never taken may be removed.

Several operators may be fused into a single composite operator.

Entire regions may migrate from CPU execution to GPU execution.

Because optimization itself is merely graph transformation,

the optimizer need not occupy a privileged role outside the computational system.

It becomes another graph acting upon graph values.

This naturally unifies

interpreters,

compilers,

optimizers,

just-in-time compilation,

adaptive execution,

and reflective programming.

Each becomes a specialized graph rewriting process operating on persistent

computational histories.

43.14 Compilation Without Languages

Perhaps the most radical consequence of this viewpoint is that compilation no longer fundamentally concerns programming languages.

Programming languages become one source of computational graphs.

Visual editors become another.

Machine learning systems may generate graphs directly.

Interactive proof assistants may produce graphs.

Optimization systems may synthesize graphs.

Even biological or physical processes may be abstracted as operator graphs.

The compiler therefore need not ask,

"What language was this written in?"

Instead it asks,

"What graph has been presented?"

and

"Into what graph should it be transformed?"

This shifts the emphasis away from syntax and toward computation itself.

Languages become interfaces.

Graphs become the computational reality.

The next section builds upon this foundation by considering cyclic graphs, recursion, persistent state, and fixed-point computation. Whereas directed acyclic graphs capture feed-forward computation, cycles introduce memory, iteration, and self-reference. Rather than requiring an entirely new execution model, these phenomena emerge as natural extensions of the same compositional semantics developed throughout this essay.

44 Cycles, Recursion, Feedback, and Fixed Points

Thus far we have concentrated primarily on directed acyclic graphs. This restriction is pedagogically convenient because every computation admits a unique dependency ordering. Once the predecessors of a node have been evaluated, the node itself may be evaluated exactly once, and execution proceeds monotonically toward the outputs.

Real computation, however, is rarely so simple.

Programs contain loops.

Digital circuits contain feedback.

Operating systems maintain persistent state.

Interactive applications respond continuously to external events.

Compilers invoke themselves recursively.

Theorem provers repeatedly refine partial proofs.

Artificial neural networks may contain recurrent connections.

These phenomena appear very different when viewed syntactically. Graphically, however, they share a common feature.

Each introduces one or more directed cycles into the composition graph.

The problem therefore is not how to invent an entirely new execution model. The problem is how to extend the compositional semantics developed earlier so that cycles become meaningful computational objects.

44.1 The Meaning of a Cycle

Consider a graph

$$G = (N, E)$$

containing a directed path

$$n_1 \to n_2 \to \cdots \to n_k \to n_1.$$

Unlike an acyclic graph, no topological ordering exists.

Each node appears to depend upon itself indirectly.

This apparent paradox has often motivated special language constructs such as

`while`,

`for`,

recursive function definitions,

state machines,

or process calculi.

Spherepop instead treats the cycle itself as the primitive object.

A cycle simply records that information produced during one stage of execution may later become an input to another stage of the same computation.

Nothing fundamentally mysterious has occurred.

Composition has simply become self-referential.

44.2 Time as Successive Histories

The difficulty with cycles disappears once histories become explicit.

A value does not feed back into the operator that originally produced it.

Rather,

a value produced during one history participates in the construction of a later history.

Suppose

$$h_0, h_1, h_2, \ldots$$

are successive execution histories.

A feedback edge should therefore be interpreted as

$$h_t \longrightarrow h_{t+1},$$

rather than

$$h_t \longrightarrow h_t.$$

The graph itself remains cyclic,

but the histories generated by repeated execution are linearly ordered.

The apparent circularity therefore disappears.

The graph describes structural dependence.

The history describes temporal realization.

Separating these two notions greatly simplifies the semantics.

44.3 Delay Operators

Many computational systems introduce explicit delay operators.

Digital circuits contain registers.

Signal processing systems contain unit delays.

Control systems contain integrators.

Functional reactive languages contain stream delays.

These all perform the same mathematical role.

A delay operator breaks an immediate dependency by replacing it with a dependency upon the previous history.

Formally,

let

$$D$$

be the delay operator.

Then

$$D(x_t) = x_{t-1}.$$

The current evaluation therefore depends upon the previously committed history rather than the partially constructed present history.

Operationally,

the scheduler no longer attempts to evaluate a cycle instantaneously.

Instead,

each iteration consumes values committed during earlier histories.

The graph remains unchanged.

Only the interpretation of the feedback edge changes.

44.4 Iteration as Repeated Graph Evaluation

This viewpoint leads to a remarkably simple interpretation of loops.

Rather than repeatedly executing statements,

the runtime repeatedly evaluates the same graph over successive histories.

Suppose

$$G$$

is the graph representing one iteration.

Execution becomes

$$G(h_0) \to h_1,$$

$$G(h_1) \to h_2,$$

$$G(h_2) \to h_3,$$

and so forth.
Nothing inside

$$G$$

changes.
The graph is constant.
Only the history supplied to it changes.
Iteration therefore becomes repeated application of one compositional object.
This interpretation naturally generalizes to simulations,
interactive systems,
stream processors,
and operating systems,
all of which repeatedly transform one history into another.

44.5 Recursion as Self-Composition

Recursive functions receive a similarly direct interpretation.
Suppose

$$f$$

calls itself.
Conventionally,
one introduces special semantic rules describing recursive invocation.
Graphically,
no such distinction is necessary.
The function graph simply contains an edge returning to its own input interface.
Recursive evaluation becomes repeated self-composition.
For example,

$$f^{(0)}(x) = x,$$

$$f^{(1)}(x) = f(x),$$

$$f^{(2)}(x) = f(f(x)),$$

and more generally

$$f^{(n)}(x) = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ times}}(x).$$

Termination corresponds to reaching a history in which further self-composition is no longer required.

From this viewpoint,

recursion is not a separate computational primitive.

It is composition applied repeatedly to the same graph.

44.6 Fixed Points as Stable Histories

Cycles introduce the possibility that repeated graph evaluation eventually stabilizes. Rather than producing an endless sequence of distinct histories,

$$h_0,\ h_1,\ h_2,\ \ldots,$$

the execution may converge to a history satisfying

$$G(h^*) = h^*.$$

Such a history is called a fixed point of the computation.

From the traditional perspective, fixed points are often introduced through recursive definitions or domain-theoretic semantics. In Spherepop they arise naturally from repeated composition.

One repeatedly applies the same graph to the current history until no further operational changes occur.

The emphasis therefore shifts from solving equations to observing the evolution of histories.

A fixed point is simply a history that reproduces itself under further admissible composition.

This interpretation is particularly natural for interactive systems.

An operating system never truly terminates.

Instead, it repeatedly evolves toward locally stable configurations before responding to new external events.

Likewise, many optimization algorithms terminate when successive histories become indistinguishable according to some operational criterion.

The notion of convergence is therefore not restricted to numerical analysis.

It becomes a general property of compositional evolution.

44.7 Persistent State as Committed History

Programming languages frequently introduce variables as primitive storage locations.

Graphically this is unnecessary.

Persistent state may instead be viewed as previously committed history.

Suppose a computation produces

$$v_t$$

during history

$$h_t.$$

If that value remains available during

$$h_{t+1},$$

the system has state.

Nothing fundamentally different has been added to the execution engine.

The history has simply become persistent.

Variables,

registers,

memory cells,

database records,

and object fields

all become particular mechanisms for exposing previously committed histories to future computations.

This interpretation removes the apparent distinction between memory and execution.

Memory is simply computation that has already been committed.

Execution is memory that is currently under construction.

The boundary between the two is the

collapse

operation.

44.8 Streams Rather Than Variables

This perspective also suggests replacing mutable variables with streams of historical values.

Instead of writing

$$x := x + 1,$$

one may regard

$$x$$

as the sequence

$$x_0,\, x_1,\, x_2,\, \ldots.$$

Each element belongs to a different history.

The update rule becomes

$$x_{t+1} = x_t + 1.$$

No value has been overwritten.

Every historical state remains available for inspection.

Many programming paradigms have independently rediscovered versions of this idea.

Functional reactive programming emphasizes streams.

Signal processing emphasizes time series.

Hardware design emphasizes clocked signals.

Version control systems preserve historical revisions.

Event sourcing records immutable events rather than mutable state.

Spherepop unifies these observations by treating every mutable object as an evolving history.

Mutation becomes an interpretation placed upon successive historical values rather than a primitive computational operation.

44.9 Repair Within Cyclic Computation

The explicit representation of histories provides an elegant account of repair in recursive systems.

Suppose repeated evaluation generates

$$h_0, h_1, \ldots, h_k.$$

If an inconsistency appears during

$$h_k,$$

there is no need to reconstruct earlier computation.

The previous histories remain available.

Repair may therefore proceed by branching from any earlier history,

modifying a local region,

and continuing execution.

The original history is preserved.

The repaired history becomes an alternative continuation.

This resembles branching in version control systems, but it occurs at the level of operational histories rather than source code.

Repair therefore becomes a first-class computational operation.

Instead of discarding failed executions,

Spherepop records them,

explains them,

and allows new histories to emerge from them.

44.10 Recurrence Without Language Constructs

One consequence of the graph interpretation is that many familiar programming constructs become optional.

Loops,

recursive functions,

iterators,

state machines,

coroutines,

generators,

and reactive systems

all express repeated interaction between successive histories.

Rather than introducing each construct separately,

Spherepop provides only composition,

persistent histories,

feedback edges,

and scheduling.

The higher-level constructs emerge as common graph patterns.

For example,

a finite-state machine is simply a graph whose current committed history determines which subgraph becomes active during the next history.

A coroutine is a graph whose execution is suspended after producing a partial history and resumed during a later history.

A generator repeatedly extends its history by producing one additional value.

These concepts differ operationally only in how histories are scheduled and connected.

The underlying compositional semantics remains unchanged.

44.11 Histories as the Fundamental Temporal Object

Traditional operational semantics often begins with program state and derives execution traces.

Spherepop reverses this relationship.

Execution histories are fundamental.

State is a projection extracted from history.

Memory is committed history.

Time is the ordering of histories.

Recursion is repeated self-composition across histories.

Feedback is the transmission of committed history into future composition.

Iteration is repeated graph evaluation over successive histories.

Fixed points are histories that become invariant under further composition.

Viewed in this way, temporal computation requires remarkably little additional machinery beyond the acyclic semantics developed earlier.

Cycles do not force the introduction of a fundamentally different execution model.

They merely require histories to persist across successive evaluations of the same composition graph.

The next section exploits this observation to show that Boolean logic, fuzzy logic, probabilistic reasoning, symbolic rewriting, neural computation, and other computational paradigms can all be interpreted as different operator libraries executing over the same history-preserving compositional substrate.

45 Simulating Computational Paradigms Through Operator Libraries

One of the strongest consequences of the preceding development is that the execution engine no longer embodies any particular computational paradigm. Once computation has been reduced to the evaluation of operator graphs over persistent histories, the distinction between programming languages, logical systems, and computational models shifts almost entirely into the operator library.

This inversion is important. Traditionally one asks whether a language supports logic programming, symbolic computation, neural computation, probabilistic reasoning, or fuzzy inference. From the perspective developed here, the more fundamental question is much simpler.

Which operators are available?

The runtime remains unchanged.

Only the collection of primitive operators changes.

Consequently, Spherepop should not be regarded as another programming language competing with existing paradigms. Rather, it is a compositional substrate upon which many paradigms may coexist simultaneously.

45.1 Boolean Computation

Boolean computation represents perhaps the simplest operator library.

The carrier space is

$$B = \{0, 1\},$$

and the primitive operators may be chosen as

$$\mathcal{L}_B = \{\wedge, \vee, \neg, \oplus\}.$$

Every combinational circuit becomes a finite composition graph over these operators.

Evaluation proceeds exactly as described in the previous sections.

Each node waits until its predecessors have produced values.

The operator associated with the node is applied.

The resulting value becomes available to successor nodes.

Nothing in the execution engine recognizes that these values represent logical truth.

They are simply members of the carrier space

$$B.$$

This observation immediately removes the distinction between logical circuits and ordinary computational graphs.

Boolean logic becomes one particular operator algebra.

45.2 Fuzzy Computation

Fuzzy logic enlarges the carrier space.

Instead of

$$\{0, 1\},$$

one works over

$$[0, 1].$$

The graph topology is unchanged.

Only the operators differ.
For example,

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

$$x \vee y = \max(x, y),$$

and

$$\neg x = 1 - x.$$

Alternative operator families are equally valid.
One may choose product t-norms,
Lukasiewicz operators,
Gödel operators,
or continuously parameterized conjunctions.
The execution scheduler remains completely unaffected.
It continues evaluating nodes according to graph dependencies.
Only the local transformation performed at each node has changed.
This illustrates one of the central principles of Spherepop.
Logical systems are operator libraries.
They are not execution engines.

45.3 Probabilistic Computation

The same reasoning extends naturally to probabilistic computation.
Instead of propagating truth values,
nodes propagate probability distributions.
The carrier space now becomes

$$\mathcal{P}(X),$$

the collection of probability measures over some domain

$$X.$$

Primitive operators perform familiar probabilistic transformations.

Some combine distributions.

Others condition upon observations.

Others perform Bayesian updates.

Others sample from distributions.

Operationally nothing changes.

Each node still receives inputs.

Each node still produces outputs.

The scheduler still traverses the graph.

Probability therefore appears not as a separate computational paradigm but as another operator algebra acting upon a different carrier space.

45.4 Symbolic Computation

Symbolic systems appear, at first glance, completely different from numerical computation.

Instead of numbers,

they manipulate expressions,

equations,

proofs,

or syntax trees.

Yet the compositional viewpoint reveals the same underlying structure.

Each node simply performs a rewrite.

Suppose

$$R$$

is a rewrite operator.

Then

$$R : E \to E,$$

where

$$E$$

is the space of symbolic expressions.

The graph itself remains identical to the graphs used for arithmetic or logic.

Only the local operators now perform substitutions,

normalizations,

simplifications,

or theorem-proving steps.

The runtime neither knows nor needs to know that symbolic manipulation is occurring.

It merely evaluates operator nodes.

45.5 Neural Computation

Artificial neural networks provide another illustration.

Each neuron computes

$$y = \sigma\left(\sum_i w_i x_i + b\right),$$

where

$$\sigma$$

is an activation function.

Graphically,

every neuron is simply another operator node.

The weighted connections become edges.

Layers become collections of nodes.

Entire networks become operator graphs.

The forward pass is one traversal of the graph.

Backpropagation is another traversal over essentially the same topology using derivative operators instead of activation operators.

Thus neural computation does not require a fundamentally different execution engine.

It merely requires another operator library together with a second graph traversal for learning.

The compositional substrate remains identical.

45.6 Arithmetic Computation

Ordinary arithmetic appears even simpler.

Primitive operators include

$$+, -, \times, \div, \sqrt{\ }, \sin, \cos, \exp, \log .$$

The carrier space becomes

$$\mathbb{R},$$

or perhaps arbitrary precision integers,

complex numbers,

or matrices.

Again,

the graph does not change.

Only the interpretation of each operator changes.

Arithmetic therefore occupies exactly the same conceptual status as Boolean logic or fuzzy reasoning.

It is another operator algebra.

45.7 The Uniformity Principle

The repeated appearance of the same execution pattern suggests a general principle.

Theorem 45.1 (Uniformity Principle)**.** *Let*

$$\mathcal{G}$$

be a composition graph and let

$$\mathcal{L}$$

be any operator library whose operators act upon a carrier space

$$X.$$

Then the execution algorithm depends only upon

$$\mathcal{G},$$

while the computational interpretation depends only upon

$$\mathcal{L}.$$

The proof is immediate.

The scheduler examines graph dependencies alone.

Operator semantics are invoked only after a node has become executable.

Therefore replacing one operator library by another leaves the scheduling algorithm unchanged.

This theorem captures the central architectural separation advocated throughout this essay.

Topology determines computation.

Operators determine meaning.

Neither determines the other.

A single compositional substrate may therefore execute Boolean circuits,

fuzzy inference systems,

Bayesian networks,

symbolic theorem provers,

arithmetic expressions,

or neural networks without changing its operational semantics.

The differences arise entirely from the operators attached to graph nodes rather than from the execution mechanism itself.

45.8 Hybrid Computational Graphs

Perhaps the most important consequence of the operator-library viewpoint is that there is no longer any reason to insist that an entire program belong to a single computational paradigm.

Traditional language design often forces such a commitment.

One language is called functional.

Another logical.

Another probabilistic.

Another symbolic.

Another differentiable.

Another hardware-oriented.

These classifications arise because the language itself has been identified with a particular collection of primitive operators.

Spherepop deliberately separates these notions.

The graph is universal.

Only the local operators vary.

Consequently, different regions of the same graph may employ entirely different computational semantics.

One subgraph may perform Boolean reasoning,

another fuzzy inference,

another symbolic rewriting,

another matrix multiplication,

another probabilistic sampling,

and another neural inference.

The outputs of each region become ordinary values consumed by the next region.

No boundary exists between computational paradigms except the choice of local operators.

This permits hybrid computational systems to be constructed naturally rather than through awkward foreign-function interfaces or language embeddings.

45.9 Operators as First-Class Values

An even stronger consequence follows if operators themselves are permitted to flow through the graph.

Suppose the carrier space contains not only ordinary values but operators,

$$\mathcal{O} \subseteq \mathcal{V}.$$

Then an operator may become the output of one computation and the input to another.

Formally, one may define higher-order operators

$$M : \mathcal{O} \to \mathcal{O},$$

or

$$C : \mathcal{O} \times \mathcal{O} \to \mathcal{O},$$

which modify or compose existing operators.

Examples include

operator fusion,

automatic differentiation,

partial application,

currying,

symbolic simplification,

or optimization.

The execution engine again remains unchanged.

The only novelty is that some values now happen to represent executable operators.

This is entirely analogous to the treatment of functions as first-class values in functional programming, except that Spherepop generalizes the idea from functions to arbitrary computational operators.

45.10 Learning New Operators

If operators are ordinary values, they may also be constructed during execution.

Suppose an optimization procedure observes many execution histories,

$$h_1, h_2, \ldots, h_n.$$

Rather than merely optimizing parameters,

the system may synthesize an entirely new operator

$$f^*$$

whose behavior approximates a frequently occurring subgraph.

Operationally,

the original graph

$$G$$

contains a large repeated region

$$H.$$

After sufficient observation,

that region may be replaced by a single synthesized node

$$f^*.$$

The graph becomes smaller,

evaluation becomes faster,

and the learned operator itself becomes part of the available operator library.

This process resembles compiler optimization,

library construction,

and machine learning simultaneously.

From the graph perspective, all three are instances of operator synthesis.

45.11 Universality Through Composition Rather Than Instruction Sets

Traditional computer architecture places great emphasis on instruction sets.

Processors are classified according to the primitive instructions they provide.

Spherepop suggests that this emphasis may be somewhat misleading.

The truly fundamental property is not the instruction set itself but unrestricted composition.

Suppose two systems possess different primitive operators,

$$\mathcal{O}_1$$

and

$$\mathcal{O}_2.$$

If every operator in

$$\mathcal{O}_1$$

can be expressed as a finite composition of operators from

$$\mathcal{O}_2,$$

then every graph over

$$\mathcal{O}_1$$

may be rewritten into an equivalent graph over

$$\mathcal{O}_2.$$

Conversely,

if the reverse translation also exists,

the two operator libraries are computationally equivalent.

This shifts the notion of universality away from particular machines.

Universality becomes a property of compositional expressiveness.

Instruction sets,

logical connectives,

language primitives,

and circuit components all become different generating sets for the same algebra of compositions.

45.12 Simulation as Graph Translation

Simulation now admits a particularly elegant formulation.

Suppose one wishes to simulate Boolean computation using fuzzy operators.

One does not modify the execution engine.

Instead,

one constructs a graph translation

$$T : G_B \to G_F,$$

mapping each Boolean operator into an equivalent fuzzy subgraph.

Likewise,

a symbolic rewrite system may be translated into graph reductions,

a finite automaton into a feedback graph,

or a neural network into ordinary arithmetic operators.

Every simulation becomes a graph homomorphism preserving observable histories.

The remarkable diversity of computational paradigms therefore reflects the diversity of available operator libraries rather than diversity in the underlying mechanism of computation.

45.13 The Ontological Status of Logical Systems

These observations suggest a different philosophical interpretation of logic itself.

Logic is often regarded as a foundational description of computation.

Spherepop instead places logic one level higher.

Primitive composition comes first.

Operator libraries come second.

Logical systems arise when particular operator libraries are interpreted as truth-preserving transformations.

Boolean logic,

fuzzy logic,

modal logic,

temporal logic,

probabilistic logic,

and intuitionistic logic

are therefore not different computational universes.

They are different semantic organizations imposed upon the same compositional substrate.

The computation does not know that it is performing logic.

It merely composes operators.

Logic emerges only when an observer interprets particular operators as logical connectives and particular values as truth-like objects.

This inversion is central to the philosophy developed throughout this essay.

Composition generates computation.

Operator libraries generate computational behavior.

Logical systems generate interpretation.

Each level explains the one below without being required for its existence.

The next section develops perhaps the strongest consequence of this inversion by arguing that syntax, grammars, type systems, and even abstract syntax trees should themselves be viewed as human-oriented descriptive layers rather than primitive computational objects.

46 Construction Without Types or Syntax

The previous sections have argued that many apparently distinct computational paradigms differ only in their operator libraries. The same argument may be pushed one step further. Not only are logical systems secondary to composition, but much of what programming language theory traditionally regards as foundational is likewise secondary.

Syntax, grammars, type systems, abstract syntax trees, namespaces, modules, and even programming languages themselves should be understood as descriptive technologies. They help human beings construct, organize, verify, communicate, and maintain computations. They are not the computation itself.

The computation is the composition graph.

Everything else is an interface to that graph.

46.1 Programs Before Languages

It is common to speak as though a program is "written in" some programming language.

This expression is convenient but philosophically misleading.

A program is not fundamentally a sequence of characters.

It is a computational object.

The textual representation merely provides one convenient method for communicating that object between humans and machines.

The same computation may be represented by

ordinary source code,

a visual node editor,

an abstract syntax tree,

an S-expression,

a hardware schematic,

a serialized graph,

or even a handwritten diagram.

These representations differ enormously.

The underlying composition graph need not.

Consequently, programming languages should be regarded as external notations rather than computational substrates.

They describe graphs.

They do not create them.

46.2 The Secondary Role of Grammar

Formal grammars occupy a central position in compiler theory.

A grammar specifies which strings belong to a language.

Parsing transforms those strings into an internal representation suitable for execution.

From the graph perspective, grammars perform only one task.

They recover graph structure from linear notation.

Human beings naturally communicate sequentially.

Speech unfolds over time.

Writing unfolds along a line.

Computers, however, need not execute linearly.

Their internal computations are overwhelmingly graph structured.

The parser therefore performs a dimensional expansion.

It converts a one-dimensional representation into a higher-dimensional dependency graph.

Nothing computationally fundamental has occurred.

The parser has reconstructed relationships that were implicit in the textual serialization.

The graph already contained the computation.

The grammar merely made it communicable.

46.3 Abstract Syntax Trees as Transitional Objects

Abstract syntax trees occupy an interesting intermediate position.

They are substantially closer to computation than source code, yet they remain shaped by syntax.

Every node corresponds to a grammatical construct.

Parent-child relationships largely reflect textual nesting.

This makes ASTs extremely useful for parsing and semantic analysis.

Nevertheless, they should not be mistaken for the computational object itself.

The computational graph frequently differs from the syntax tree.

Common subexpressions become shared nodes.

Evaluation dependencies cross syntactic boundaries.

Optimization introduces new structures absent from the original syntax.

Data-flow edges become explicit.

Control dependencies become explicit.

Eventually the syntax tree disappears almost entirely.

What remains is the operator graph.

ASTs therefore function as transitional representations.

They bridge human syntax and computational topology.

They are neither the beginning nor the end of the compilation process.

46.4 Why Types Feel Fundamental

Type systems often create the impression that they define the nature of computation.

One cannot apply an integer as though it were a function.

One cannot add strings to matrices without defining suitable operators.

One cannot access fields that do not exist.

These observations are entirely correct.

What they demonstrate, however, is not that types generate computation.

Rather, they demonstrate that types summarize regularities within families of computations.

Suppose an operator

$$f$$

expects two floating-point numbers.

Nothing in the execution engine requires the existence of a floating-point type.

The operator merely expects certain inputs.

The type system records this expectation in a form convenient for reasoning.

In other words,

types are compressed descriptions of operator compatibility.

The runtime ultimately checks only whether an operator can be evaluated.

The type checker predicts this before execution begins.

Its purpose is explanatory and preventative rather than generative.

46.5 Types as Regions of the Composition Graph

The graph viewpoint suggests an alternative interpretation of types.

Instead of regarding a type as a set of values,

one may regard it as a region within the graph where particular operators may interact safely.

Suppose a family of operators repeatedly appears together.

The type system recognizes this regularity and assigns it a name.

The resulting type is therefore not an intrinsic property of the values themselves.

It is a property of admissible compositions.

Viewed this way,

typing is fundamentally relational.

It concerns compatibility between operators rather than ontology of objects.

The same value may legitimately participate in many different typed regions depending upon which operators surround it.

This interpretation aligns naturally with Spherepop's emphasis on histories and admissibility rather than intrinsic object classifications.

46.6 Construction as Human Compression

One may push the argument even further.

Programming itself may be viewed as an act of compression.

Instead of manipulating millions of primitive operator nodes directly,

human beings construct

functions,

classes,

modules,

packages,

libraries,

and programming languages.

Each level suppresses detail while exposing reusable structure.

These abstractions are extraordinarily valuable.

Without them,

large software systems would be impossible to understand.

Their value, however, lies in cognitive organization rather than computational necessity.

The execution engine ultimately expands every abstraction back into compositions of primitive operators.

Abstraction therefore performs the inverse role of compilation.

Compilation expands human descriptions into executable graphs.

Programming compresses executable graphs into human-manageable descriptions.

Both processes manipulate the same underlying computational object from opposite directions.

This observation explains why programming language theory and compiler theory mirror one another so closely.

They study opposite transformations between graph topology and human representation.

46.7 Programming Languages as Views Rather Than Realities

The graph-first interpretation suggests a useful analogy with geometry.

A three-dimensional object may be viewed from many directions.

Each projection reveals certain features while hiding others.

No single projection is the object itself.

Programming languages play an analogous role.

Lisp presents computation as nested symbolic expressions.

C presents computation as imperative procedures over memory.

Haskell presents computation through typed functional composition.

Prolog presents computation as logical inference.

Verilog presents computation as interconnected hardware modules.

Python emphasizes readability and dynamic composition.

Each language encourages a particular way of viewing computation.

None of them constitutes computation itself.

They are coordinate systems on an underlying compositional object.

The same computational graph may therefore possess many equally valid textual representations.

Translating between languages becomes analogous to changing coordinates.

The underlying graph remains unchanged.

Only its description changes.

This viewpoint explains why apparently different languages can often express exactly the same algorithms with surprisingly small semantic differences.

46.8 The Illusion of Sequential Programs

Most programming languages encourage the programmer to think sequentially.

[OStatements appear one after another.

Functions occupy contiguous blocks of text.

Control appears to flow from the beginning of the file toward the end.

This sequential appearance is almost entirely a consequence of human communication.

The underlying computation is rarely sequential.

Even a simple arithmetic expression

$$(a + b)(c + d)$$

contains two independent additions that may execute simultaneously.

Likewise,

large software systems contain enormous amounts of implicit parallelism.

The compiler spends considerable effort recovering this parallel structure from sequential syntax.

Spherepop proposes beginning with the parallel structure directly.

The graph already records true computational dependence.

Sequential execution becomes merely one scheduling policy among many.

This inversion removes a significant amount of accidental complexity introduced solely by textual programming languages.

46.9 Naming as Compression

Names also deserve reinterpretation.

Variables,

functions,

classes,

modules,

packages,

and namespaces all assign symbolic names to portions of a graph.

Nothing computationally requires these names.

The execution engine could assign anonymous identifiers to every node.

Human beings, however, are extremely poor at remembering large anonymous structures.

Naming therefore acts as another compression mechanism.

Instead of repeatedly reconstructing a complicated subgraph,

one simply writes

$$f,$$

or

$$sort,$$

or

$$parse.$$

The name becomes a pointer into the much larger composition graph.

This explains why naming conventions play such a large role in software engineering while playing essentially no role in execution semantics.

Names are cognitive tools.

They are not computational primitives.

46.10 Syntax as a Lossy Projection

Textual syntax possesses another important property.

It is inherently lossy.

A graph explicitly records every dependency.

Linear text cannot.

Many important relationships become implicit.

For example,

data dependencies,

parallel evaluation,

sharing,

resource usage,

and communication structure

must often be reconstructed through sophisticated analysis.

This reconstruction occupies much of modern compiler theory.

The compiler repeatedly attempts to recover information that was lost when the graph was serialized into text.

From the Spherepop perspective,

this effort is largely unnecessary.

If the graph remains the primary representation,

dependency information is never discarded.

Surface syntax becomes merely an import and export format rather than the canonical computational object.

46.11 Reasoning Directly About Graphs

Once graphs become primary,

many traditional analyses become considerably simpler.

Reachability becomes ordinary graph traversal.

Dependency analysis becomes edge inspection.

Parallel scheduling becomes graph partitioning.

Dead code elimination becomes removal of disconnected subgraphs. [I Inlining becomes graph substitution.

Common subexpression elimination becomes graph merging.

Loop analysis becomes cycle analysis.

Function composition becomes graph composition.

The remarkable diversity of compiler algorithms therefore reflects different questions asked about one mathematical object.

Instead of studying separate theories of parsing,

optimization,

control flow,

and data flow,

one studies a single theory of graph transformations.

This conceptual economy is one of the strongest motivations for adopting a

graph-first computational ontology.

46.12 The Spherepop Philosophy of Construction

The central philosophical claim of this section may now be stated succinctly.

Human beings construct descriptions.

Machines construct histories.

Descriptions consist of

syntax,

grammars,

types,

modules,

proofs,

and abstractions.

Histories consist of composed operator evaluations.

Descriptions are optimized for understanding.

Histories are optimized for execution.

Compilation transforms descriptions into histories.

Program comprehension transforms histories back into descriptions.

Neither direction is more fundamental.

Both operate upon the same underlying composition graph.

Spherepop therefore refuses to identify computation with any particular descriptive framework.

Languages,

type systems,

and proof calculi remain indispensable engineering achievements.

They dramatically improve reliability,

maintainability,

verification,

and communication.

Yet they should be understood as sophisticated human interfaces to a simpler computational reality.

That reality consists only of operators,

their compositions,

and the histories generated by their execution.

The next section turns from representation to optimization. Once computation is understood as graph transformation, optimization itself becomes a geometric process of rewriting composition graphs into observationally equivalent forms that reduce cost, increase parallelism, improve locality, or simplify subsequent transformations.

47 Optimization as Graph Transformation

The graph-first interpretation developed throughout this essay naturally changes the meaning of optimization. Conventionally, optimization is viewed as an assortment of compiler techniques whose purpose is to improve execution speed, reduce memory consumption, or decrease code size. Individual optimizations are usually presented independently, each possessing its own algorithms and correctness arguments.

From the perspective of Spherepop, these distinctions are secondary.

Optimization is fundamentally the search for a better representative of an operational equivalence class.

The graph being optimized is not changed because its computation is incorrect.

It is changed because the same history can be generated more efficiently.

Every optimization therefore becomes a graph transformation preserving observable computational behavior while improving some resource measure.

47.1 Optimization Objectives

Suppose a graph

$$G$$

possesses observable behavior

$$\mathcal{H}(G).$$

Rather than speaking simply of ”the” optimized graph, we introduce an objective function

$$J(G),$$

which measures computational cost.

Typical objectives include

execution time,

memory usage,

communication cost,

power consumption,

latency,

parallel efficiency,

or numerical stability.

Optimization then becomes the search for

$$G^* = \arg\min_{G'} J(G')$$

subject to

$$\mathcal{H}(G') \cong \mathcal{H}(G).$$

Different execution environments therefore produce different optimal graphs even when they compute exactly the same histories.

Optimization is relative to computational objectives rather than absolute.

47.2 Observational Equivalence Classes

Every computation determines not one graph but an entire family of equivalent graphs.

Define

$$[G] = \{G' \mid \mathcal{H}(G') \cong \mathcal{H}(G)\}.$$

This equivalence class contains every graph producing the same observable computation.

Optimization never leaves this class.

Instead,

it searches within it.

This interpretation immediately unifies many apparently unrelated compiler techniques.

Inlining,

constant propagation,

instruction selection,

operator fusion,

parallel scheduling,

and graph partitioning

all remain inside the same equivalence class.

They merely move toward different representatives.

The compiler therefore behaves less like a translator than like a navigator moving through a space of equivalent composition graphs.

47.3 Graph Geometry

This suggests assigning a geometry to computational graphs.

Suppose each rewrite operator

$$R_i$$

acts locally upon a graph.

The collection of admissible rewrites generates a graph of graphs,

whose vertices are computational graphs

and whose edges represent elementary rewrites.

Optimization becomes a path through this higher-order graph.

Rather than asking

"How do we optimize this program?"

one asks

"What path through graph space leads to a lower-cost representative?"

This viewpoint connects naturally with search algorithms,

local optimization,

simulated annealing,

genetic programming,

and machine-learned optimizers.

All become methods for navigating graph space.

47.4 Operator Fusion

One particularly natural graph rewrite is operator fusion.

Suppose two adjacent nodes compute

$$f$$

followed immediately by

$$g.$$

Rather than evaluating them separately,
one constructs a composite operator

$$h = g \circ f.$$

Graphically,
two nodes become one.
The surrounding topology remains unchanged.
Fusion may reduce memory traffic,
improve locality,
or expose additional optimization opportunities.
This technique appears throughout modern computation.
Neural network compilers fuse activation functions with matrix multiplication.
Signal processing systems fuse filter stages.
Database engines fuse query operators.
Functional languages fuse list transformations.
These examples differ operationally only in the operators being composed.
The graph transformation itself is identical.

47.5 Graph Partitioning

Not every optimization reduces graph size.
Some improve execution by reorganizing topology.
Suppose

$$G$$

contains several weakly interacting regions.
One may partition

$$G = G_1 \cup G_2 \cup \cdots \cup G_k,$$

assigning each region to a different processor,

machine,

or hardware accelerator.

The computation remains unchanged.

Only the physical realization differs.

Distributed computing,

GPU scheduling,

hardware synthesis,

and cloud deployment

all become instances of graph partitioning.

The scheduler simply evaluates different regions on different computational substrates while preserving dependency relationships.

47.6 Optimization as Compression

An alternative interpretation views optimization as information compression.

Large graphs often contain repeated structures.

Equivalent operators.

Identical subgraphs.

Symmetric branches.

Repeated computational motifs.

Optimization discovers these regularities and replaces them with shorter descriptions.

This is precisely what compression algorithms accomplish for data.

Optimization therefore compresses computation rather than information.

The optimized graph represents the same operational histories using fewer computational resources.

This observation connects compiler optimization with minimum description length,

Kolmogorov complexity,

and program induction.

Each seeks shorter representations of equivalent computational behavior.

47.7 Learning Cost Functions

Traditional compilers employ manually designed heuristics.

Graph-first computation suggests a more general approach.

Suppose repeated executions produce histories

$$h_1, h_2, \ldots, h_n.$$

Performance measurements extracted from these histories define empirical costs.

Rather than fixing

$$J(G)$$

in advance,

the system may learn it.

Optimization then becomes adaptive.

Graphs that perform well become preferred representatives.

Graphs performing poorly are gradually abandoned.

This transforms optimization into a continuous learning problem.

The graph rewrite system remains unchanged.

Only the objective function evolves.

Thus optimization itself becomes another operator acting upon persistent histories.

47.8 Optimization as Repair

Within the broader philosophy of Spherepop, optimization is most naturally interpreted as a special case of repair.

Traditionally these subjects are separated.

Correctness concerns whether a program computes the intended result.

Optimization concerns whether it computes the result efficiently.

The graph viewpoint suggests that these are points on a continuum rather than fundamentally different activities.

Suppose

$$G$$

is a composition graph.

If

$$G$$

produces an incorrect observable history, the graph requires semantic repair.

If

$$G$$

produces the correct history but consumes excessive resources, the graph requires performance repair.

In both cases the underlying operation is identical.

One graph is rewritten into another while preserving some collection of desired properties.

Repair therefore becomes the more general concept.

Correctness repair,

performance repair,

energy repair,

latency repair,

parallelization,

and hardware specialization

all become different optimization objectives over the same graph.

This observation further unifies compiler theory with the broader repair-theoretic framework developed in earlier essays.

47.9 Resource Geometry

Optimization is often described qualitatively.

Programs become ”faster.”

Algorithms become ”more efficient.”

From the graph perspective these notions acquire a geometric interpretation.

Each node possesses an associated execution cost

$$c(n).$$

Each edge possesses a communication cost

$$w(e).$$

The total computational cost becomes

$$J(G) = \sum_{n \in N} c(n) + \sum_{e \in E} w(e).$$

Different hardware platforms induce different cost geometries.

On one architecture,

computation may be expensive while communication is cheap.

On another,

the opposite may hold.

Consequently,

optimization is not performed in an abstract vacuum.

It is performed relative to a particular computational geometry.

The same composition graph may therefore admit many different optimal realizations depending upon the resource landscape through which it is embedded.

47.10 Graph Curvature and Bottlenecks

Certain regions of a computation naturally resist parallelization.

Many independent paths converge,

communication becomes concentrated,

or sequential dependencies accumulate.

These regions behave as computational bottlenecks.

One may regard them as regions of high graph curvature.

The exact mathematical definition of curvature may vary,

but the intuition is straightforward.

Flat regions admit many equivalent execution schedules.

Highly curved regions constrain scheduling severely.

Optimization therefore attempts to flatten the graph wherever possible.

Operator fusion,

common subgraph elimination,

pipeline restructuring,

and dependency reduction

all reduce effective curvature by enlarging the space of admissible execution schedules.

This interpretation connects optimization with the broader geometric language used throughout the repair-theoretic program.

47.11 Scheduling as Geometry Rather Than Time

Execution order is often treated as an intrinsic property of a program.

The graph perspective suggests otherwise.

A graph determines only partial order.

Many nodes may become executable simultaneously.

A scheduler chooses one particular realization of that partial order.

Suppose

$$S$$

is a scheduler.

Then execution becomes

$$(G, S) \longrightarrow H,$$

where

$$H$$

is the resulting history.

Different schedulers may produce different histories while remaining observationally equivalent.

This observation is important.

Time is not fundamentally encoded within the graph.

Only dependency is.

Time emerges from the scheduler's traversal of dependency structure.

Sequential execution,

parallel execution,

distributed execution,

speculative execution,

and lazy evaluation

become different geometric traversals of the same computational object.

47.12 Lazy and Eager Evaluation

Lazy evaluation and eager evaluation illustrate this principle particularly clearly.

In eager evaluation,

every node is executed as soon as its predecessors become available.

In lazy evaluation,

evaluation is postponed until some successor actually requires the result.

Graphically,

the composition graph is identical.

Only the scheduling policy changes.

The graph therefore represents potential computation.

The scheduler determines when that potential becomes actual history.

Evaluation strategy is not part of the computation itself.

It is a property of graph traversal.

This observation removes another apparent distinction between programming paradigms.

Strict functional languages,

lazy functional languages,

data-flow systems,

and demand-driven systems

all execute the same underlying graph under different scheduling policies.

47.13 The Optimization Hierarchy

The various optimization techniques discussed throughout this section may now be organized into a remarkably simple hierarchy.

At the lowest level,

operators are rewritten into equivalent operators.

Above this,

small graph fragments are rewritten into simpler fragments.

Above this,

entire subgraphs are reorganized,

partitioned,

or fused.

Above this,

schedulers choose efficient traversals of unchanged graphs.

Above this,

operator libraries themselves may be specialized,

learned,

or synthesized.

Every optimization therefore acts upon one of five levels:

operator optimization,
local graph rewriting,
subgraph transformation,
execution scheduling,
operator-library evolution.

Despite their apparent diversity,

each level manipulates the same underlying compositional object.

The only difference lies in the scale at which graph transformations are applied.

This hierarchy reinforces the central thesis of the present essay.

Optimization is not an auxiliary discipline attached to computation.

It is computation acting upon itself.

The optimizer is simply another composition graph whose outputs happen to be improved composition graphs.

The distinction between compiler,

optimizer,

program transformer,

repair engine,

and learning system therefore becomes largely one of emphasis rather than of fundamental computational mechanism.

48 Relationship to Existing Computational Foundations

The preceding sections have deliberately developed Spherepop from the bottom upward. Rather than beginning with an established formalism and extending it,

we have begun with primitive composition and asked what structures necessarily emerge. It is therefore useful to compare this perspective with several classical foundations of computation.

The objective of this section is not to argue that previous foundations are incorrect. On the contrary, each captures an important aspect of computation. The claim is instead that they may all be understood as different descriptive layers over a common algebra of composition graphs and persistent histories.

48.1 Lambda Calculus

The lambda calculus is perhaps the best-known mathematical foundation of functional computation.

Its primitive operations are variable abstraction,

$$\lambda x.t,$$

and application,

$$t\,u.$$

Beta reduction,

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

defines computation through substitution.

Spherepop agrees that abstraction and application are extraordinarily expressive.

However, it interprets them differently.

A lambda abstraction is not regarded as a primitive ontological object.

It is a reusable composition graph possessing designated input and output interfaces.

Application is not fundamentally substitution.

It is graph composition.

Beta reduction is therefore one particular graph rewrite among many.

The lambda calculus emerges as one convenient symbolic notation for constructing composition graphs rather than as the primitive source of computation itself.

This interpretation also explains why many computational models equivalent to the lambda calculus nevertheless possess radically different syntax.

The underlying composition graph is invariant.

Only its symbolic presentation changes.

48.2 Combinatory Logic

Combinatory logic goes one step further than the lambda calculus by eliminating variables entirely.

Primitive combinators such as

$$S, \qquad K, \qquad I$$

generate arbitrary computation through repeated composition.

In many respects this philosophy is remarkably close to Spherepop.

Both systems emphasize composition rather than symbolic substitution.

The principal difference concerns representation.

Combinatory logic focuses upon algebraic expressions.

Spherepop focuses upon explicit composition graphs.

A combinator expression therefore corresponds naturally to a particular graph.

Repeated combinator reduction becomes graph rewriting.

From this perspective,

combinatory logic may be regarded as one textual coordinate system for graph computation.

48.3 Lisp

Lisp occupies a particularly interesting position.

Unlike many programming languages,

its fundamental data structure is extraordinarily simple.

Repeated application of

`cons`

constructs lists,

trees,

symbolic expressions,

and eventually entire programs.

This observation strongly influenced the present development.

Spherepop adopts the same minimalist philosophy while shifting attention from symbolic list construction toward arbitrary graph construction.

One may view

`cons`

as a primitive graph-building operator.

Repeated composition then generates progressively richer computational objects.

The principal difference is therefore geometric.

Lisp naturally emphasizes trees.

Spherepop begins directly with graphs.

Trees become one particularly important special case.

This small shift substantially enlarges the natural representational power of the underlying substrate while preserving Lisp's remarkable conceptual economy.

48.4 Turing Machines

The Turing machine provides another classical foundation.

Its primitive components include

a tape,

a finite control,

a read-write head,

and a transition function.

Execution proceeds sequentially by repeatedly updating machine state.

Spherepop does not dispute the universality of this model.

Instead,

it regards the Turing machine as a particular realization of a composition graph.

The transition function becomes an operator.

Tape updates become persistent histories.

Machine states become regions of the graph.

Sequential execution becomes one scheduling policy.

The familiar Turing model therefore appears as a highly specialized execution strategy embedded within a much more general compositional framework.

This observation also explains why many computational systems possessing no obvious tape structure nevertheless remain computationally universal.

Universality depends upon composition,

not upon tapes.

48.5 Dataflow Systems

Among existing computational paradigms,

dataflow programming comes perhaps closest to the graph-first interpretation advocated here.

Nodes perform computations.

Edges transmit values.

Execution proceeds when inputs become available.

Spherepop inherits this basic intuition.

The principal additions are persistent histories,

explicit repair,

operator libraries,

and the separation between graph topology and semantic interpretation.

A dataflow graph ordinarily specifies one computational paradigm.

A Spherepop graph specifies only composition.

The computational paradigm emerges from the operator library attached to the nodes.

This distinction permits Boolean, symbolic, probabilistic, fuzzy, and neural computation to coexist naturally within the same graph.

48.6 Category Theory

Category theory is frequently presented as one of the most abstract foundations of computation. Objects represent mathematical entities, while morphisms represent structure-preserving maps between them. Composition of morphisms satisfies associativity, and every object possesses an identity morphism.

Spherepop shares category theory's emphasis on composition but differs in what it treats as primitive.

Category theory begins with objects and morphisms.

Spherepop begins with operators and histories.

Objects emerge as relatively stable regions of persistent computational histories rather than existing independently of computation.

Likewise, morphisms need not be interpreted solely as static mathematical mappings.

They become operational events embedded within evolving execution histories.

Composition therefore acquires a dynamic interpretation.

Instead of asking only whether two morphisms compose,

Spherepop asks how the resulting composition is evaluated,

what history it produces,

how that history may be repaired,

and how it may subsequently be transformed.

Category theory therefore provides a valuable algebra of composition.

Spherepop attempts to supply an operational semantics for that algebra.

48.7 The Calculus of Constructions

The calculus of constructions represents one of the richest known foundations for proof-producing computation.

Dependent products,

universes,

identity types,

and propositions-as-types permit programs and mathematical proofs to inhabit the same formal system.

Nothing developed in the present essay contradicts this achievement.

Instead, the relationship is inverted.

The calculus of constructions is interpreted as one particularly sophisticated descriptive layer over composition graphs.

Dependent types become constraints on admissible compositions.

Proof terms become histories demonstrating that certain graph transformations preserve specified properties.

Universes classify families of graph constructions.

Normalization becomes graph reduction.

Type checking becomes prediction that a proposed graph composition will execute coherently before execution actually occurs.

The expressive power of the calculus remains intact.

Only its ontological status changes.

Rather than generating computation,

it describes computation.

This distinction explains why computers remain capable of computation even when no type checker is present.

The execution engine requires only composition.

The calculus of constructions assists human beings in organizing and verifying those compositions.

48.8 Graphs as the Common Foundation

These comparisons reveal a recurring pattern.

Lambda calculus emphasizes substitution.

Combinatory logic emphasizes variable-free composition.

Lisp emphasizes recursive symbolic construction.

Turing machines emphasize sequential state transitions.

Dataflow systems emphasize dependency.

Category theory emphasizes compositional algebra.

The calculus of constructions emphasizes verification and proof.

Each framework illuminates an important aspect of computation.

None appears to require abandoning the graph-first interpretation.

Instead, each becomes a different projection of the same underlying compositional substrate.

The common mathematical object is neither the syntax tree,

nor the tape,

nor the proof term,

nor the lambda expression,

nor the category.

It is the composition graph together with the histories generated by its execution.

Different theories simply choose different features of this object as their primary subject of study.

48.9 A Layered View of Computational Foundations

The resulting picture suggests a layered organization of computational theory.

At the lowest level lies unrestricted operator composition.

From repeated composition emerge execution graphs.

Execution of those graphs produces persistent histories.

Scheduling determines one realization of those histories.

Operator libraries determine computational behavior.

Optimization transforms graphs while preserving observable histories.

Only after these operational structures exist do higher descriptive layers naturally appear.

Programming languages provide textual views of graphs.

Grammars serialize and recover graph structure.

Type systems summarize admissible patterns of composition.

Logical systems assign semantic interpretations to operator behavior.

Proof calculi certify that graph transformations preserve desired properties.

This ordering differs significantly from the traditional presentation found in programming language theory, where syntax and typing often precede operational semantics.

Spherepop reverses the direction of explanation.

Operational composition comes first.

Description follows afterward.

48.10 The Principle of Descriptive Emergence

The comparisons developed throughout this section motivate what may be regarded as the central philosophical principle of this essay.

> Every sufficiently expressive formal system describing computation may be understood as a descriptive organization imposed upon an underlying algebra of composed operators and persistent histories.

The purpose of this principle is not to diminish the importance of type theory,

logic,

category theory,

or programming languages.

On the contrary,

their extraordinary success reflects the effectiveness with which they compress,

organize,

verify,

and communicate complex computational structures.

The claim is only that these descriptive systems should not be confused with the operational substrate from which computation itself arises.

The substrate is composition.

Everything else emerges by progressively enriching our description of composed histories.

This perspective naturally leads to the practical question addressed in the next section: if computation is fundamentally graph composition, what should a Spherepop runtime and virtual machine actually look like?

49 Runtime Architecture and the Spherepop Virtual Machine

The theoretical development of the preceding sections naturally raises a practical question. If composition graphs and persistent histories constitute the primitive computational objects, what would an implementation actually execute?

The answer is surprisingly simple.

Unlike conventional virtual machines, whose execution models are inherited from sequential processors, a Spherepop virtual machine executes graphs directly. The runtime does not interpret statements, evaluate expressions, or execute instructions in textual order. Instead, it maintains a persistent composition graph together with the history generated by traversing that graph.

Everything else is layered above this minimal substrate.

49.1 The Minimal Runtime

The smallest useful Spherepop runtime consists of only five components.

The first component is the composition graph

$$G = (N, E),$$

which stores operators together with their dependency relationships.

The second component is the operator library

$$\mathcal{O},$$

containing the primitive computational transformations associated with graph nodes.

The third component is the execution history

$$H,$$

which records every committed computational event.

The fourth component is the scheduler

$$S,$$

which determines which nodes become executable.

The fifth component is the graph store

$$M,$$

which maintains reusable graph fragments, synthesized operators, imported libraries, and persistent computational objects.

The runtime state is therefore simply

$$R = (G, \mathcal{O}, H, S, M).$$

Remarkably little additional machinery is fundamentally required.

49.2 The Execution Cycle

Execution proceeds through repeated application of a single operational cycle.

The scheduler examines the current graph and identifies all nodes whose dependencies have already been satisfied.

One executable node is selected according to the scheduling policy.

Its associated operator is evaluated.

The resulting value is attached to the corresponding node.

A new event is appended to the execution history.

Successor nodes whose dependencies have now become satisfied are inserted into the scheduler.

The cycle repeats until no executable nodes remain or until execution is intentionally suspended.

Notice that this procedure makes no reference to programming-language syntax.

The runtime neither knows nor cares whether the graph originated from Spherepop,

Lisp,

Python,

Verilog,

a neural-network compiler,

or an interactive graph editor.

Every source language ultimately supplies the same internal representation.

49.3 The Scheduler

One of the more interesting components of the runtime is the scheduler.

Traditional virtual machines often embed a specific execution order directly into the instruction stream.

Spherepop separates dependency from scheduling.

The graph determines only which evaluations are permissible.

The scheduler determines which permissible evaluation actually occurs next.

This distinction allows a variety of execution strategies.

A sequential scheduler executes one node at a time.

A parallel scheduler executes all independent nodes simultaneously.

A distributed scheduler partitions the graph across several machines.

A speculative scheduler evaluates likely future branches before they become necessary.

A lazy scheduler postpones evaluation until values are demanded.

The graph remains unchanged under every one of these policies.

Only the traversal changes.

Execution order therefore becomes an implementation decision rather than a semantic property of the computation itself.

49.4 Histories as First-Class Runtime Objects

Unlike conventional execution traces,

the history maintained by the Spherepop runtime is not merely a debugging aid.

It is an active computational object.

Optimization algorithms inspect histories.

Repair operators branch from histories.

Learning systems discover recurring subgraphs by analyzing histories.

Proof generators extract derivations from histories.

Visualizers display histories.

Interactive development environments replay histories.

Consequently,

the runtime never discards historical information simply because execution has progressed.

The history becomes another persistent data structure available to every computational subsystem.

This persistent representation also makes execution reproducible.

Rather than storing only the current state,

the runtime preserves the sequence of transformations that produced that state.

Computation therefore acquires intrinsic provenance.

Every value carries with it the history of its construction.

49.5 Operators as Loadable Modules

The separation between execution and semantics naturally suggests a modular architecture for operator libraries.

Rather than hard-coding arithmetic,

logic,

probability,

or symbolic reasoning into the runtime,

operators are loaded dynamically.

A library contributes a collection of primitive operators together with metadata describing their arity,

resource requirements,

determinism,

and optimization rules.

The runtime need not understand their mathematical meaning.

It needs only to know how to invoke them.

Consequently,

new computational paradigms may be introduced simply by loading new operator libraries.

No modification of the execution engine is required.

The runtime therefore resembles a small operating system whose primary responsibility is coordinating compositions rather than interpreting programming languages.

49.6 Graph Objects as Runtime Values

Perhaps the most distinctive feature of the Spherepop runtime is that graphs themselves are ordinary runtime values.

A node may consume a graph.

A node may produce a graph.

A graph may specialize another graph.

A graph may optimize itself.

A graph may synthesize new operators.

A graph may construct entirely new computational pipelines during execution.

This capability largely eliminates the traditional distinction between compile time and run time.

Compilation becomes graph transformation.

Optimization becomes graph transformation.

Program generation becomes graph transformation.

Execution itself becomes graph transformation.

All four activities manipulate the same underlying computational object.

The runtime therefore supports reflection and metaprogramming without introducing separate semantic mechanisms.

Graphs compute over graphs exactly as numbers compute over numbers.

49.7 The Runtime as an Operating System for Computation

The graph-oriented architecture suggests a useful reinterpretation of the virtual machine itself.

Rather than viewing the runtime as an interpreter for one programming language, it is more accurate to regard it as an operating system for computations.

Traditional operating systems schedule processes.

Spherepop schedules operator graphs.

Traditional operating systems allocate memory.

Spherepop allocates graph regions and histories.

Traditional operating systems manage files.

Spherepop manages persistent computational objects.

Traditional operating systems load executable programs.

Spherepop loads operator libraries.

In both cases the underlying responsibility is remarkably similar.

Resources are managed.

Dependencies are respected.

Independent activities are coordinated.

The principal difference is that the primitive object being managed is no longer the process but the composition graph.

49.8 Memory as Persistent Graph Storage

Conventional computer architectures distinguish sharply between memory and computation.

Memory stores data.

Processors perform computation.

The graph interpretation softens this distinction considerably.

Memory becomes the persistent storage of computational objects.

A stored graph is simply a computation that has not yet been scheduled.

Likewise,

an executing graph is merely a stored graph whose operators are currently producing histories.

The boundary between storage and execution therefore becomes fluid.

A graph may remain dormant for years.

At some later time it may be loaded,

specialized,

rewritten,

executed,

partially evaluated,

or embedded within another graph.

Nothing fundamental changes.

The graph itself remains the primary computational object throughout its entire

lifetime.

49.9 Caching as History Preservation

Caching also acquires a particularly natural interpretation.

Suppose a graph fragment

$$G_0$$

has already produced a history

$$H_0.$$

If exactly the same fragment later receives the same historical inputs,
there is no need to execute it again.
The previous history may simply be reused.
Traditional memoization,
dynamic programming,
common subexpression elimination,
and execution caching
all become manifestations of the same principle.
Previously constructed histories need not be reconstructed.
The runtime therefore treats histories as reusable computational resources.
Rather than asking
"Has this function already been called?"
one asks
"Has this graph already generated an equivalent history?"

The second formulation is substantially more general because it applies equally well to arbitrary graph fragments rather than only named functions.

49.10 Persistence Across Executions

Because histories are first-class objects,

the runtime need not begin from an empty state every time a program executes.

Instead,

graphs and histories may persist indefinitely.

Learning systems accumulate experience.

Optimization systems remember successful rewrites.

Repair systems remember previous corrections.

Proof systems retain completed derivations.

Interactive environments preserve computational context.

Execution therefore becomes cumulative.

Each new computation begins with the histories generated by previous computations.

This perspective differs markedly from the traditional process model,

where each execution begins with newly initialized memory.

Spherepop instead treats computation as a continuously evolving ecosystem of persistent graph objects.

49.11 Reflection as Ordinary Computation

Reflection is often regarded as an advanced language feature.

Programs inspect themselves,

modify themselves,

or generate new programs.

Within the present framework,

reflection requires no special semantic machinery.

Because graphs are ordinary runtime values,

one graph may simply receive another graph as input.

Inspection becomes graph traversal.

Modification becomes graph rewriting.

Generation becomes graph construction.

Self-modification becomes graph rewriting applied to the currently executing graph.

The execution engine remains entirely unchanged.

Reflection therefore ceases to be a privileged capability.

It is an ordinary computation over graph values.

49.12 Distributed Computation

The graph-first architecture also provides a remarkably clean foundation for distributed execution.

Suppose

$$G = G_1 \cup G_2 \cup \cdots \cup G_n$$

is partitioned across several machines.

Each machine evaluates its assigned subgraph.

Communication occurs only when dependency edges cross partition boundaries.

Nothing else changes.

Every machine executes precisely the same scheduler,

the same operator semantics,

and the same history construction rules.

Distributed execution therefore becomes graph partitioning followed by coordinated scheduling.

The runtime need not distinguish between local computation and remote computation.

Distance simply becomes another edge cost within the graph geometry.

49.13 A Computational Ecosystem

Taken together,

these observations suggest that the Spherepop runtime should not be viewed merely as another virtual machine.

It is better understood as an ecosystem supporting the continuous evolution of computational graphs.

Graphs are created.

Graphs execute.

Graphs generate histories.

Histories produce optimizations.

Optimizations rewrite graphs.

Graphs synthesize new operators.

New operators generate richer graphs.

The computational substrate therefore evolves continuously rather than merely

executing static programs.

This recursive relationship between graphs,

histories,

repair,

optimization,

and operator synthesis captures one of the central ambitions of Spherepop.

The runtime is not simply a machine that executes programs.

It is an environment within which computational structures develop,

specialize,

repair themselves,

and accumulate operational knowledge over time.

The final section of this essay draws these threads together, arguing that composition should be regarded as the primitive computational ontology from which programming languages, type theories, logical systems, proof calculi, and execution architectures all emerge as progressively richer descriptive layers.

50 Conclusion and Future Directions

The central claim of this essay has been deliberately simple.

Composition is the primitive computational act.

Everything else is descriptive.

Beginning from this premise, we have reconstructed a remarkably broad collection of computational ideas without requiring them as primitives. Directed acyclic graphs emerged as feed-forward composition. Cyclic graphs emerged through persistent histories. Recursion became repeated self-composition. State became committed history. Optimization became graph rewriting. Compilation became graph transformation. Type systems became descriptions of admissible compositions. Logical systems became operator libraries. Proof systems became certificates concerning graph transformations.

This inversion of the traditional hierarchy changes the role of many familiar concepts.

Programming languages no longer define computation.

They describe computation.

Type systems no longer generate admissibility.

They summarize admissibility.

Proof calculi no longer create correctness.

They certify correctness.

Execution engines no longer interpret syntax.

They traverse composition graphs.

Each level remains valuable, but each derives its significance from the compositional substrate beneath it.

One advantage of this viewpoint is conceptual economy.

The same execution engine may evaluate Boolean circuits,

fuzzy inference systems,

probabilistic graphical models,

symbolic rewrite systems,

neural networks,

stream processors,

interactive applications,

hardware descriptions,

or theorem provers.

No change to the operational semantics is required.

Only the operator library changes.

Likewise,

the same graph may be executed sequentially,

in parallel,

lazily,

speculatively,

or across a distributed system simply by changing the scheduler.

The graph remains invariant.

Scheduling becomes another interpretation of the same computational object.

Persistent histories provide another unifying principle.

Rather than treating execution traces as disposable artifacts,

Spherepop elevates them to first-class computational objects.

Histories support explanation,

repair,

optimization,

learning,

proof extraction,

visualization,

and provenance simultaneously.

The traditional separation between execution,

debugging,

verification,

and optimization therefore begins to disappear.

Each becomes a different analysis performed over the same accumulated history.

Perhaps the most significant philosophical consequence concerns the relationship between computation and description.

Human beings rarely manipulate raw computational graphs.

Instead,

we invent progressively richer descriptive systems.

We create variables,

functions,

modules,

packages,

grammars,

type systems,

logical calculi,

proof assistants,

and programming languages.

These inventions are extraordinarily successful because they compress enormous computational structures into forms that can be understood,

communicated,

verified,

and maintained.

Their success should not be mistaken for ontological priority.

They exist because computation is difficult for humans to reason about directly.

The composition graph exists independently of the particular descriptive language chosen to present it.

This perspective suggests several promising directions for future work.

The first is the development of a complete formal semantics for history-preserving graph execution, including proofs of normalization, confluence where appropriate,

observational equivalence, and correctness of graph rewriting.

The second is the construction of a practical Spherepop virtual machine implementing the architecture described in this essay. Such a runtime would treat graphs, histories, operators, and schedulers as first-class objects rather than embedding them beneath the abstractions of conventional programming languages.

The third direction concerns adaptive operator libraries. Because operators are ordinary computational values, they may be synthesized, optimized, specialized, or learned directly from execution histories. This suggests computational systems capable not merely of learning parameters but of gradually extending their own primitive computational vocabulary.

A fourth direction concerns repair-oriented execution. If histories remain persistent throughout execution, repair need no longer be viewed as exceptional error handling. Instead, computational systems may branch from previous histories, preserve unsuccessful attempts, compare alternative repairs, and accumulate operational experience over time.

Finally, there remains the broader question of computational ontology. Throughout this essay we have argued that graphs and histories provide a simpler foundation than syntax, typing, or logical systems. Whether this perspective ultimately proves fruitful will depend not upon philosophical elegance alone but upon its ability to simplify implementations, unify apparently unrelated computational paradigms, and support new forms of adaptive computation.

If successful, the resulting picture is surprisingly economical.

Primitive operators compose.

Composition generates graphs.

Graphs generate histories.

Histories admit repair.

Repair induces optimization.

Optimization reshapes graphs.

Descriptions summarize graphs.

Proofs certify transformations.

Languages communicate descriptions.

Everything above composition serves to organize, explain, or constrain what the underlying computational substrate is already capable of expressing.

From this perspective, computation is neither fundamentally symbolic nor numerical, neither functional nor imperative, neither logical nor probabilistic.

It is the progressive construction and transformation of persistent histories through the composition of operators.

All richer computational structures emerge from that single principle.

Appendices

A The Algebra of Composition Graphs

The body of this essay has argued that composition graphs constitute the primitive computational objects from which programming languages, logical systems, and proof calculi emerge. This appendix develops the mathematical structure underlying that claim.

Rather than beginning with syntax, we begin directly with graphs equipped with computational operators.

A.1 Composition Graphs

Definition A.1 (Composition graph)**.** *A composition graph is a quadruple*

$$G = (N, E, \mathcal{O}, \ell),$$

where

- N *is a finite set of nodes,*
- $E \subseteq N \times N$ *is a directed edge relation,*
- $\mathcal{O}$ *is a collection of primitive operators,*
- $\ell : N \to \mathcal{O}$ *assigns one operator to each node.*

Edges describe computational dependence rather than temporal order.

Notice that nothing in this definition refers to variables, syntax trees, statements, types, or programming languages.

Only composition is primitive.

A.2 Interfaces

Every graph possesses an interface.

Definition A.2 (Input interface)**.** *The input interface of a graph is the collection*

$$I(G) = \{n \in N : \mathrm{pred}(n) = \varnothing\},$$

where

$$\text{pred}(n) = \{m : (m, n) \in E\}.$$

Likewise,

Definition A.3 (Output interface)**.** *The output interface is*

$$O(G) = \{n \in N : \text{succ}(n) = \varnothing\},$$

where

$$\text{succ}(n) = \{m : (n, m) \in E\}.$$

These interfaces play exactly the role that function arguments and return values play in conventional programming languages.

The difference is that interfaces are defined geometrically rather than syntactically.

A.3 Graph Composition

The fundamental operation of Spherepop is graph composition.

Suppose

$$G_1 = (N_1, E_1),$$

and

$$G_2 = (N_2, E_2),$$

are two composition graphs whose interfaces are compatible.

Their composition,

written

$$G_2 \circ G_1,$$

is obtained by identifying the output interface of

$$G_1$$

with the input interface of

$$G_2$$

and preserving every remaining edge.

Operationally,

the outputs generated by

$$G_1$$

become the inputs consumed by

$$G_2.$$

No additional semantic machinery is required.

A.4 Associativity

Composition satisfies the expected associativity property.

Theorem A.4. *Whenever all interface connections are defined,*

$$(G_3 \circ G_2) \circ G_1 = G_3 \circ (G_2 \circ G_1).$$

Proof. Both constructions identify exactly the same interfaces and preserve exactly the same dependency edges.

The only difference is the order in which interface identifications are performed.

Since graph union together with interface identification is associative up to graph isomorphism,

the resulting composition graphs are isomorphic.

□

Associativity is perhaps the single most important algebraic property of the entire framework.

It allows arbitrarily large computations to be assembled hierarchically from smaller graph fragments without changing observable behavior.

Functions,

modules,

libraries,

and complete programs all become repeated applications of this one operation.

A.5 Identity Graphs

Composition also possesses identity elements.

Definition A.5. *For every compatible interface*

$$I,$$

the identity graph

$$\mathrm{Id}_I$$

contains only interface nodes connected by identity operators.

Composing any graph with

$$\mathrm{Id}_I$$

leaves the graph unchanged up to graph isomorphism.

Identity graphs play the same algebraic role as identity morphisms in category theory or identity functions in functional programming.

They contribute no new computation while preserving compositional structure.

Together with associativity, they show that composition graphs form an algebra under graph composition, providing the mathematical foundation for the graph-first philosophy developed throughout the essay.

A.6 Subgraphs

The notion of a subgraph plays a central role throughout Spherepop.

Functions,

modules,

libraries,

optimization regions,

compiler passes,

and even entire programming languages may all be interpreted as distinguished subgraphs embedded within larger computational graphs.

Definition A.6 (Subgraph)**.** *Let*

$$G = (N, E)$$

be a composition graph.

A graph

$$H = (N_H, E_H)$$

is called a subgraph of

$$G$$

whenever

$$N_H \subseteq N,$$

and

$$E_H = E \cap (N_H \times N_H).$$

Every subgraph therefore inherits both its operators and its dependency structure from the larger computation.

This inheritance makes graph rewriting particularly simple.

Optimization rarely constructs computation from nothing.

Instead,

it replaces one subgraph with another.

A.7 Boundary Interfaces

Every subgraph possesses a boundary separating internal computation from the surrounding graph.

This boundary completely characterizes how the subgraph interacts with its environment.

Definition A.7 (Boundary)**.** *The boundary of a subgraph*

$$H$$

consists of

$$\partial H = I(H) \cup O(H),$$

where

$$I(H)$$

contains every node receiving an edge from outside

$$H,$$

and

$$O(H)$$

contains every node sending an edge outside

$$H.$$

The remarkable consequence is that every optimization,

every function call,

every module,

and every compiler transformation acts only through this boundary.

The internal organization may change dramatically while the surrounding graph remains completely unaware of those changes.

A.8 Graph Equivalence

Two graphs need not possess identical node sets to represent the same computation.

Instead,

they must generate equivalent observable histories.

This motivates the following definition.

Definition A.8 (Operational equivalence)**.** *Two graphs*

$$G_1$$

and

$$G_2$$

are operationally equivalent,

written

$$G_1 \simeq G_2,$$

whenever they produce observationally equivalent histories for every admissible input history.

This equivalence relation is considerably stronger than textual equality.

Different programming languages,

different optimization strategies,

and different hardware implementations may all produce operationally equivalent graphs despite possessing entirely different internal structures.

Operational equivalence therefore replaces syntactic identity as the fundamental notion of sameness.

A.9 Rewrite Systems

Optimization and compilation both rely upon graph rewriting.

Formally,

a rewrite rule is simply an ordered pair

$$(L, R),$$

where

$$L$$

and

$$R$$

are operationally equivalent subgraphs.

Whenever

$$L$$

appears inside a larger graph,

it may be replaced by

$$R.$$

Graphically,

$$C[L] \Longrightarrow C[R],$$

where

$$C$$

denotes the surrounding graph context.

This notation closely resembles term rewriting,

but the rewritten object is now an arbitrary composition graph rather than a symbolic expression.

A.10 The Locality Principle

One of the most important properties of graph rewriting is locality.

Most optimizations modify only a tiny fraction of the graph.

The overwhelming majority of nodes remain untouched.

Formally,

suppose

$$G' = R(G).$$

Then typically

$$|N(G) \triangle N(G')| \ll |N(G)|.$$

Only a small neighborhood surrounding the rewritten subgraph changes.

This locality has several important consequences.

First,

optimization scales well because only small regions require analysis.

Second,

parallel optimization becomes possible because distant graph regions may be rewritten independently.

Third,

correctness proofs become substantially simpler since only the rewritten neighborhood requires verification.

Locality therefore explains much of the practical success of modern compiler technology.

A.11 Graph Factorization

Repeated computational motifs naturally suggest graph factorization.

Suppose a graph contains several identical subgraphs,

$$H_1, H_2, \ldots, H_n,$$

with

$$H_i \simeq H_j$$

for every pair.

Rather than storing each separately,

one may replace them with a single representative

$$H$$

together with multiple interface connections.

Operationally,

this is exactly the same principle underlying

functions,

subroutines,

macros,

libraries,

hardware modules,

and object-oriented methods.

All perform graph factorization.

Instead of repeatedly constructing identical computation,

they reuse existing subgraphs.

This observation provides another example of how familiar programming concepts emerge naturally from graph algebra rather than requiring independent semantic foundations.

A.12 Composition Algebras

The collection of all finite composition graphs together with graph composition forms an algebraic structure.

Composition supplies the binary operation.

Identity graphs supply neutral elements.

Operational equivalence partitions graphs into equivalence classes.

Rewrite rules generate transformations within those classes.

Factorization generates reusable graph fragments.

Graph substitution generates hierarchical composition.

Taken together,

these operations constitute an algebra of computation considerably more general than traditional symbolic algebras.

Instead of manipulating equations,

the algebra manipulates computations themselves.

This appendix therefore provides the mathematical foundation for one of the central claims of the essay.

Programs are not fundamentally texts,

trees,

proofs,

or typed expressions.

They are elements of an algebra whose primitive operation is graph composition and whose observable meaning is determined by the histories generated through execution.

A.13 Execution Histories as Morphisms

The graph algebra developed thus far describes the static organization of computation. It remains to describe how execution itself interacts with this structure.

Rather than viewing a graph as a machine that instantaneously transforms inputs into outputs, Spherepop regards execution as the construction of histories.

Consequently, the primary semantic object is not merely the graph

$$G,$$

but the pair

$$(G, H),$$

where

$$H$$

records the operational history generated by executing

$$G.$$

This distinction separates potential computation from realized computation.
The graph specifies what may occur.
The history records what actually occurred.

A.14 History Extension

Execution proceeds by extending histories.

Let

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

be the current history.
A single execution step produces

$$H_{n+1} = (H_n, e_{n+1}).$$

Notice that no previous event is modified.
Histories are monotone.
They grow by extension.
This monotonicity distinguishes histories from mutable program state.
State may appear to change.
Histories never do.
Every previous computation remains available for later inspection.
This property underlies reproducibility,
repair,
provenance,
and explanation.

A.15 History Composition

Just as graphs compose,

histories compose.

Suppose

$$H_1 = (e_1, \ldots, e_m),$$

and

$$H_2 = (e_{m+1}, \ldots, e_n).$$

Their composition is simply concatenation,

$$H_2 \circ H_1 = (e_1, \ldots, e_n).$$

Operationally,
history composition corresponds to executing one graph after another.
Unlike graph composition,
which combines computational structure,
history composition combines realized computation.
The two notions therefore occupy complementary levels.
Graphs describe possibility.
Histories describe realization.

A.16 The History Functor

Every execution graph naturally generates histories.

This observation may be expressed as a mapping

$$\mathcal{H} : \mathsf{Graph} \longrightarrow \mathsf{History},$$

assigning to each graph the collection of histories obtainable from its execution.
Composition behaves naturally with respect to this mapping.
Whenever graph composition is defined,

$$\mathcal{H}(G_2 \circ G_1) = \mathcal{H}(G_2) \circ \mathcal{H}(G_1),$$

up to observational equivalence.

The significance of this relationship is considerable.

Graph composition predicts history composition.

Operational semantics therefore preserves compositional structure.

Execution does not destroy the algebra developed earlier.

It realizes it.

A.17 Branching Histories

Not every computation produces a single history.

Interactive systems,

probabilistic systems,

symbolic search,

repair,

and speculative execution naturally produce branching histories.

Rather than representing execution by a single sequence,

we obtain a history tree

$$T(H).$$

Each branch corresponds to an alternative continuation.

Importantly,

all branches share their common prefixes.

Only the divergent suffixes differ.

Repair therefore does not require restarting computation.

It merely extends an existing history along a different branch.

Likewise,

backtracking,

proof search,

constraint solving,

and speculative execution all become special cases of branching history construction.

The runtime therefore requires no separate semantic machinery for these paradigms.

Branching histories already provide the necessary structure.

A.18 History Metrics

Because histories are explicit mathematical objects,

they may be compared quantitatively.

Suppose

$$H_1$$

and

$$H_2$$

share a common prefix.

One natural measure of separation is the length of the shortest divergent suffix.

Another compares differing operator events.

Another compares resource consumption.

More generally,

one may introduce a metric

$$d_H(H_1, H_2),$$

measuring operational distance between histories.

Such metrics support numerous computational tasks.

Repair seeks nearby successful histories.

Optimization seeks histories requiring fewer computational resources.

Debugging seeks the earliest point at which two histories diverge.

Learning seeks recurring history fragments.

Thus history geometry becomes as important as graph geometry.

A.19 The Duality Between Graphs and Histories

A recurring theme of this appendix is the duality between graphs and histories.

Graphs describe static possibility.

Histories describe dynamic realization.

Graphs emphasize dependency.

Histories emphasize temporal development.

Graphs compose spatially.

Histories compose temporally.

Optimization primarily transforms graphs.

Learning often analyzes histories.

Compilation constructs graphs.

Execution constructs histories.

Neither object is sufficient by itself.

A graph without histories never executes.

A history without an underlying graph possesses no computational explanation.

Together they provide complementary descriptions of the same computational process.

This duality is perhaps the most distinctive mathematical feature of Spherepop.

Rather than reducing computation entirely to syntax,

or entirely to state transitions,

or entirely to functions,

it represents computation as the continuous interaction between compositional structure and historical realization.

The remaining appendices build upon this duality by developing the algebra of operator libraries and showing formally how Boolean logic, fuzzy logic, probabilistic computation, symbolic rewriting, and differentiable computation arise as different semantic interpretations over the same compositional substrate.

B Operator Algebras and Computational Semantics

The previous appendix developed the algebra of composition graphs independently of any particular notion of computation. No assumptions were made concerning arithmetic, logic, probability, symbolic manipulation, or machine learning. The purpose of the present appendix is to show that these computational paradigms arise entirely from the choice of operator algebra attached to an otherwise unchanged composition graph.

This separation between graph topology and operator semantics is one of the central mathematical principles of Spherepop.

B.1 Operator Algebras

Let

$$\mathcal{V}$$

be a carrier space.

An operator algebra is a pair

$$(\mathcal{V}, \mathcal{O}),$$

where

$$\mathcal{O} = \{f_i\}$$

is a collection of computable mappings

$$f_i : \mathcal{V}^{n_i} \to \mathcal{V}.$$

The execution engine makes no assumptions concerning the internal structure of

$$\mathcal{V}.$$

It may contain

numbers,

Boolean values,

graphs,

proofs,

probability distributions,

functions,

operators,

or arbitrary computational objects.

Likewise,

the runtime imposes no restrictions upon the mathematical interpretation of the operators.

Its only responsibility is to evaluate them once their inputs become available.

B.2 Semantic Independence

The graph

$$G$$

and the operator algebra

$$(\mathcal{V}, \mathcal{O})$$

are logically independent.

The graph determines only

$$\text{dependency},$$

while the operator algebra determines

$$\text{transformation}.$$

Execution therefore consists of evaluating

$$(G, \mathcal{O}),$$

rather than either object individually.

This separation has an important consequence.

The same graph may be evaluated using many different operator algebras without changing its topology.

Conversely,

the same operator algebra may act upon many different graphs.

The runtime therefore decomposes naturally into structural and semantic components.

B.3 Boolean Operator Algebras

Classical digital computation corresponds to the carrier space

$$\mathcal{V} = \{0, 1\},$$

together with operators such as

$$\wedge, \qquad \vee, \qquad \neg, \qquad \oplus.$$

Each graph node computes one Boolean transformation.

Nothing in the graph itself distinguishes these operators from arithmetic or symbolic operators.

They simply happen to act upon a two-element carrier space.

Consequently,

Boolean computation appears as one point within a much larger family of operator algebras.

B.4 Arithmetic Operator Algebras

Likewise,

ordinary numerical computation chooses

$$\mathcal{V} = \mathbb{R},$$

or perhaps

$$\mathbb{Z}, \qquad \mathbb{C}, \qquad M_n(\mathbb{R}),$$

together with familiar operators

$$+, -, \times, \div, \sin, \exp, \log .$$

Again,

the graph topology remains unchanged.

Only the operators differ.

Arithmetic therefore differs from Boolean logic only through its choice of carrier space and primitive transformations.

The execution semantics remain identical.

B.5 Fuzzy Operator Algebras

Fuzzy computation enlarges the Boolean carrier space from

$$\{0, 1\}$$

to

$$[0, 1].$$

Conjunction,

disjunction,

and negation become continuous operators.

For example,

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

$$x \vee y = \max(x, y),$$

$$\neg x = 1 - x.$$

Alternative fuzzy algebras simply replace these local operators.

No graph transformations are required.

The runtime therefore remains entirely unaware that fuzzy reasoning is occurring.

Only the operator library changes.

B.6 Parameterized Operator Families

Many computational systems need not commit to a fixed operator algebra.

Instead,

operators themselves may depend continuously upon parameters.

Suppose

$$f_\theta : \mathcal{V}^n \to \mathcal{V},$$

where

$$\theta$$

belongs to some parameter space.

Changing

$$\theta$$

continuously transforms the computational semantics while preserving graph topology.

Boolean logic,

fuzzy logic,

soft logic,

and differentiable approximations may therefore be viewed as different parameter choices within one continuous operator family.

The graph remains invariant throughout this deformation.

This provides a particularly elegant account of systems that gradually interpolate between discrete and continuous reasoning.

B.7 Closure Under Composition

An operator algebra should be stable under repeated graph construction.

Suppose

$$f, g \in \mathcal{O}.$$

Whenever their interfaces are compatible,

their composition

$$g \circ f$$

again defines a computable operator.

Repeated graph construction therefore generates progressively richer operators from a comparatively small primitive library.

This closure property explains why practical programming languages require surprisingly few primitive constructs.

Large computational systems emerge through repeated composition rather than through enormous collections of unrelated primitives.

Composition therefore functions as a generator of computational complexity.

The operator algebra supplies only the local building blocks.

The composition graph determines how those blocks interact.

Together they generate every computational behavior considered throughout this essay.

B.8 Operators Acting on Operators

One of the advantages of separating operator algebras from graph topology is that operators themselves become legitimate computational values.

Conventionally,

operators are treated as part of the execution engine.

Spherepop instead regards them as ordinary objects inhabiting the computational universe.

Suppose

$$\mathcal{O} \subseteq \mathcal{V}.$$

Then an operator may become the input to another operator.

For example,

$$M : \mathcal{O} \to \mathcal{O}$$

may optimize an operator,

while

$$C : \mathcal{O} \times \mathcal{O} \to \mathcal{O}$$

may compose two operators into a single composite operator.

Likewise,

$$D : \mathcal{O} \to \mathcal{O}$$

may construct the derivative of a differentiable operator,

and

$$R : \mathcal{O} \to \mathcal{O}$$

may generate a repair transformation.

Nothing special is required from the runtime.

The execution engine merely evaluates another operator.

This recursive treatment eliminates the traditional distinction between programs and meta-programs.

Operators manipulate operators exactly as numerical operators manipulate numbers.

B.9 Operator Synthesis

If operators are ordinary computational values,

they may also be synthesized during execution.

Suppose repeated histories reveal a frequently occurring graph fragment

$$H.$$

Rather than repeatedly evaluating

$$H,$$

the runtime may construct a new primitive operator

$$f_H$$

whose behavior is operationally equivalent to the entire fragment.

Graphically,

many nodes collapse into one.

Operationally,

nothing changes.

The synthesized operator simply encapsulates an existing computation.

This process resembles function extraction,

compiler optimization,

hardware synthesis,

and concept formation simultaneously.

Repeated computational patterns gradually become primitive vocabulary.

The operator algebra therefore evolves together with the execution histories that generated it.

B.10 Operator Equivalence

Different primitive operators need not possess different computational meanings.

Suppose

$$f, g \in \mathcal{O}.$$

We say that

$$f \equiv g$$

whenever,

for every admissible collection of input histories,

their observable output histories are operationally equivalent.

Notice that this notion of equivalence is behavioral rather than syntactic.

Two operators may possess entirely different internal implementations while remaining operationally indistinguishable.

This observation plays the same role for operators that observational equivalence played earlier for graphs.

Both emphasize computational behavior rather than representation.

B.11 Minimal Generating Sets

One naturally asks how many primitive operators are actually necessary.

Suppose

$$\mathcal{G} \subseteq \mathcal{O}$$

is a subset whose closure under composition generates every operator in

$$\mathcal{O}.$$

Then

$$\mathcal{G}$$

is called a generating set.

Many familiar computational systems possess surprisingly small generating sets.

Boolean circuits may be generated from NAND alone.

Lambda calculus requires only abstraction and application.

Combinatory logic employs only a handful of primitive combinators.

Lisp builds much of its expressive power from repeated applications of

`cons`,

quotation,

and evaluation.

Spherepop suggests that the search for minimal generating sets should focus not upon syntax but upon operator composition.

The expressive power of a computational substrate depends less upon the number of primitives than upon the richness of the compositions they admit.

B.12 Operator Metrics

Once operators become explicit mathematical objects,

they may be compared quantitatively.

Suppose

$$f$$

and

$$g$$

act upon the same carrier space.

One may introduce a metric

$$d_{\mathcal{O}}(f, g),$$

measuring operational similarity.

Several possibilities naturally arise.

The operators may be compared according to

their observable histories,

their computational cost,

their graph complexity,

their resource usage,

their approximation error,

or the graph distance between their implementations.

Such metrics permit clustering,

library organization,

automatic specialization,

and adaptive synthesis.

The operator algebra therefore acquires its own intrinsic geometry,

complementing the graph geometry developed in Appendix A.

B.13 Changing Semantics Without Changing Structure

Perhaps the most important mathematical consequence of the operator algebra is the complete separation of computational structure from computational interpretation.

Suppose

$$G$$

is fixed.

Replacing one operator algebra

$$(\mathcal{V}, \mathcal{O}_1)$$

with another

$$(\mathcal{W}, \mathcal{O}_2)$$

changes only the local transformations performed at each node.

The dependency graph remains identical.

Consequently,

one may execute precisely the same computational structure as

a Boolean circuit,

a fuzzy inference engine,

a probabilistic graphical model,

a symbolic rewrite system,

a neural network,

or a differentiable computational graph

simply by changing the operator library attached to the nodes.

This theorem-like observation is perhaps the strongest formal justification for the graph-first philosophy developed throughout the essay.

Graph topology represents the invariant organizational structure of computation.

Operator algebras determine its semantic realization.

The execution engine mediates between the two without privileging any particular computational paradigm.

The final appendix exploits this separation to develop a formal theory of graph rewriting, showing that compilation, optimization, repair, specialization, and learning may all be understood as different families of admissible graph transformations preserving appropriate notions of observational equivalence.

C A Calculus of Graph Rewriting

The preceding appendices established two complementary mathematical structures. Composition graphs describe the organization of computation, while operator algebras describe the local semantics attached to graph nodes. We now combine these structures into a formal calculus of graph rewriting.

The purpose of this appendix is to show that compilation, optimization, repair, specialization, learning, and program transformation are all instances of a single mathematical operation: the replacement of one composition graph by another while preserving an appropriate notion of operational behavior.

C.1 Rewrite Rules

The primitive object of the calculus is the rewrite rule.

Definition C.1 (Rewrite Rule)**.** *A rewrite rule is an ordered pair*

$$R = (L, R),$$

where

$$L$$

and

$$R$$

are composition graphs possessing compatible boundary interfaces.
Whenever

$$L$$

appears as a subgraph of a larger graph

$$G,$$

it may be replaced by

$$R.$$

The notation is intentionally suggestive.
The left-hand graph specifies the pattern being recognized.
The right-hand graph specifies the replacement.
Unlike ordinary term rewriting,
both objects are arbitrary directed graphs.

C.2 Matching

Before a rewrite may occur,
the runtime must identify an occurrence of the left-hand graph.

Definition C.2 (Matching)**.** *A match of*

$$L$$

inside

$$G$$

is an injective graph homomorphism

$$\phi : L \to G$$

preserving
nodes,
edges,
operators,
and boundary interfaces.

Operationally,
matching is pattern recognition.
The compiler,
optimizer,
or repair engine searches for graph regions possessing the required structure.
Once found,
the rewrite becomes applicable.
This formulation unifies

pattern-based compiler optimization,

rule-based theorem proving,

symbolic simplification,

and hardware synthesis.

Each consists fundamentally of graph matching followed by graph replacement.

C.3 Replacement

Suppose

$$\phi : L \to G$$

is a successful match.

The rewritten graph

$$G'$$

is obtained by removing

$$\phi(L)$$

from

$$G$$

and inserting a fresh copy of

$$R,$$

connecting every boundary edge exactly as before.

Interior structure may change completely.

Boundary behavior remains unchanged.

The surrounding graph therefore cannot distinguish the rewritten region except through changes in computational cost or internal organization.

This locality property explains why graph rewriting scales effectively even for very large computational systems.

C.4 Admissible Rewrites

Not every rewrite is computationally meaningful.

A rewrite should preserve the intended operational behavior.

Definition C.3 (Admissible Rewrite)**.** *A rewrite*

$$L \Longrightarrow R$$

is admissible whenever

$$L \simeq R,$$

where

$$\simeq$$

denotes operational equivalence.

Different applications require different notions of equivalence.

Compiler optimization usually preserves exact outputs.

Approximate numerical optimization may preserve results within prescribed tolerances.

Probabilistic computation may preserve distributions.

Repair may preserve admissible continuations rather than exact histories.

The rewrite calculus itself remains unchanged.

Only the equivalence relation varies.

C.5 Confluence

Whenever several rewrites are simultaneously applicable,

one naturally asks whether the order of application matters.

Suppose

$$G \Longrightarrow G_1,$$

and

$$G \Longrightarrow G_2.$$

A desirable property is that both rewrite sequences eventually reconverge.

Definition C.4 (Confluence)**.** *A rewrite system is confluent whenever*

$$G \Longrightarrow G_1, \qquad G \Longrightarrow G_2$$

implies the existence of some graph

$$H$$

satisfying

$$G_1 \Longrightarrow^* H, \qquad G_2 \Longrightarrow^* H.$$

Confluence guarantees that optimization does not depend upon arbitrary choices made during rewriting.

Many practical rewrite systems intentionally sacrifice global confluence in exchange for efficiency.

Nevertheless,

local confluence remains an important design objective.

C.6 Termination

Repeated rewriting should not continue indefinitely unless intended.

A rewrite system terminates whenever every rewrite sequence eventually reaches a graph to which no further rewrite applies.

One common proof technique assigns each graph a measure

$$\mu(G),$$

such that every rewrite strictly decreases

$$\mu.$$

Termination then follows immediately because no infinite descending sequence exists.

Different optimization systems choose different measures.

Some minimize graph size.

Others minimize execution cost.

Others minimize resource usage.

The rewrite calculus itself remains independent of the particular objective.

C.7 Critical Pairs

One of the principal difficulties in any rewrite system arises when several rewrite rules may be applied to overlapping regions of the same graph.

Suppose

$$R_1 : L_1 \Longrightarrow R_1,$$

and

$$R_2 : L_2 \Longrightarrow R_2,$$

are simultaneously applicable.

If the matched regions are disjoint,

the order of application is irrelevant.

Each rewrite acts independently.

The situation becomes more interesting when the matched subgraphs overlap.

The two rewrites may compete for the same computational region.

Definition C.5 (Critical Pair)**.** *A critical pair consists of two admissible rewrites whose left-hand sides overlap within a common composition graph.*

Critical pairs identify precisely those locations where the rewrite calculus must determine whether different rewrite orders remain operationally equivalent.

Traditional term rewriting systems devote considerable attention to critical pair analysis.

The graph formulation generalizes these ideas from trees to arbitrary directed dependency structures.

C.8 Commuting Rewrites

Many practical optimizations commute naturally.

Suppose

$$R_1$$

simplifies one subgraph while

$$R_2$$

optimizes another disjoint region.
Then

$$R_1(R_2(G)) = R_2(R_1(G))$$

up to graph isomorphism.
Such rewrites are said to commute.
Commutativity has several practical consequences.
Independent compiler passes may execute in parallel.
Distributed optimization becomes possible.
Large graphs may be partitioned into regions optimized simultaneously.
Reasoning about correctness becomes local rather than global.
The scheduler therefore need not serialize every optimization.
Whenever rewrites commute,
their relative order becomes irrelevant.
This property greatly enlarges the available optimization space.

C.9 Parallel Graph Rewriting

The graph-oriented formulation naturally supports parallel rewriting.
Suppose

$$G = G_1 \cup G_2 \cup \cdots \cup G_n,$$

where the regions are pairwise disjoint.
Each region may undergo rewriting independently.
Formally,

$$R_i : G_i \Longrightarrow G_i',$$

for

$$i = 1, \ldots, n.$$

The rewritten graph becomes

$$G' = G_1' \cup G_2' \cup \cdots \cup G_n'.$$

Because no dependencies cross the rewritten boundaries,

the resulting graph is observationally equivalent to any sequential application of the same rewrites.

Parallel optimization therefore emerges naturally from graph locality rather than requiring special compiler mechanisms.

C.10 Repair as Constrained Rewriting

Repair theory fits particularly naturally within the rewrite calculus.

Suppose a graph

$$G$$

produces an inadmissible history.

Rather than abandoning the computation,

one seeks another graph

$$G'$$

satisfying

$$G \Longrightarrow^{*} G',$$

together with

$$\mathcal{H}(G') \in \mathcal{A},$$

where

$$\mathcal{A}$$

denotes the collection of admissible histories.

Repair therefore differs from ordinary optimization only through its objective.

Optimization minimizes computational cost.

Repair restores admissibility.

Both employ precisely the same graph rewriting machinery.

This observation provides a formal bridge between the present essay and the broader repair-theoretic framework developed elsewhere.

C.11 Learning Rewrite Rules

Traditional compiler optimizations are designed manually.

Spherepop suggests a more adaptive alternative.

Suppose repeated execution histories reveal that one graph fragment repeatedly transforms into another during successful optimization.

The runtime may infer a candidate rewrite

$$L \Longrightarrow R.$$

Subsequent executions test the candidate.

If operational equivalence is preserved,

the rewrite becomes part of the permanent rewrite library.

Learning therefore becomes rewrite discovery.

Rather than merely adjusting numerical parameters,

the computational system gradually expands its algebra of admissible graph transformations.

This viewpoint unifies machine learning,

compiler optimization,

and program synthesis within a common mathematical framework.

C.12 The Fundamental Graph Rewriting Theorem

The preceding development may be summarized by the following principle.

Theorem C.6 (Fundamental Graph Rewriting Theorem)**.** *Let*

$$G$$

be a composition graph,
and let

$$\mathcal{R}$$

be a collection of admissible rewrite rules preserving operational equivalence.
Then every finite sequence of rewrites

$$G \Longrightarrow G_1 \Longrightarrow G_2 \Longrightarrow \cdots \Longrightarrow G_n$$

produces a graph satisfying

$$G_n \simeq G.$$

Consequently,
compilation,
optimization,
repair,
specialization,
hardware lowering,
operator fusion,
partial evaluation,
and program transformation all remain within the same operational equivalence class.

Proof. Each rewrite individually preserves operational equivalence by definition.

Since operational equivalence is transitive,

the composition of finitely many admissible rewrites also preserves operational equivalence.

Therefore

$$G_n \simeq G.$$

□

This theorem provides the mathematical justification for viewing graph rewriting as the primitive mechanism underlying an enormous range of computational transformations.

Rather than introducing separate theories for optimization,

compilation,

repair,

and specialization,

one studies a single algebra of admissible graph rewrites.

C.13 The Graph-Centric View of Computation

The three appendices together establish a unified mathematical picture.

Appendix A introduced the algebra of composition graphs.

Appendix B separated graph topology from operator semantics through operator algebras.

The present appendix has shown that computation evolves through admissible graph rewriting while preserving observable histories.

The resulting hierarchy is remarkably compact.

Composition generates graphs.

Graphs become executable through operator algebras.

Execution generates histories.

Histories motivate graph rewrites.

Graph rewrites produce repair,

optimization,

learning,

specialization,

and compilation.

Higher-level programming languages,

type systems,

logical calculi,

and proof assistants then emerge as increasingly sophisticated descriptive systems for constructing, constraining, and reasoning about these graphs.

This mathematical hierarchy captures the principal thesis of the essay: computation is fundamentally an evolving algebra of composed operators. Everything traditionally regarded as the foundation of programming languages—including syntax, types, logical systems, and proof calculi—should be understood as progressively richer descriptive organizations of that more primitive compositional substrate.

D Reference Implementation of the Spherepop Execution Engine

The preceding appendices developed the mathematical foundations of Spherepop independently of any particular implementation. The purpose of this appendix is to demonstrate that the corresponding runtime architecture is remarkably small. Nearly all of the expressive power of the system resides in graph composition and operator libraries rather than in the execution engine itself.

The implementation described here is intentionally abstract. It should be viewed

as a mathematical specification rather than a particular programming language implementation.

D.1 Runtime State

The complete runtime state consists of the tuple

$$\mathcal{R} = (G, \mathcal{O}, H, Q, M),$$

where

$$G$$

is the current composition graph,

$$\mathcal{O}$$

is the active operator library,

$$H$$

is the persistent execution history,

$$Q$$

is the scheduler queue,
and

$$M$$

stores reusable graph fragments together with synthesized operators.
No program counter is required.
Execution position is determined entirely by graph dependencies.
Likewise,
there is no evaluation stack in the traditional sense.
Dependency structure replaces sequential nesting.

D.2 Initialization

Execution begins by loading a graph together with one or more operator libraries.

Formally,

$$(G, \mathcal{O}) \longmapsto \mathcal{R}_0.$$

The scheduler then computes

$$Q_0 = I(G),$$

where

$$I(G)$$

is the graph's input interface.

Initially,

only nodes possessing no unresolved predecessors are executable.

Every remaining node becomes executable only after its dependencies have produced histories.

D.3 The Evaluation Loop

The operational heart of the runtime consists of a single loop.

At each iteration,

the scheduler removes one executable node

$$n$$

from

$$Q.$$

Suppose

$$\ell(n) = f.$$

The runtime gathers the values produced by every predecessor,

constructing the tuple

$$(v_1, \ldots, v_k).$$

Evaluation produces

$$v = f(v_1, \ldots, v_k).$$

The node output becomes

$$v.$$

The history is extended,

$$H \leftarrow (H, e),$$

where

$$e = (n, f, v).$$

Finally,

every successor whose dependencies have become satisfied is inserted into the scheduler.

No additional semantic rules are required.

Every computational paradigm discussed throughout this essay executes through repeated application of this single loop.

D.4 Refusal During Execution

Spherepop differs from conventional runtimes by treating refusal as an ordinary operational outcome.

Suppose an operator determines that continuation is inadmissible.

Instead of producing an ordinary value,

it may produce

$$\mathsf{refuse}(r),$$

where

$$r$$

records the reason for refusal.

The runtime records this event within the history exactly as it records successful evaluation.

Subsequent operators may

repair the refusal,

propagate it,

branch around it,

or terminate execution.

Refusal therefore does not constitute an exceptional failure of the runtime.

It is simply another computational event.

This treatment considerably simplifies implementations supporting constraint solving,

proof search,

interactive reasoning,

or repair-oriented computation.

D.5 Collapse

Many computations accumulate provisional values before committing them.

Spherepop therefore distinguishes evaluation from commitment.

Suppose a node has produced the provisional value

$$v.$$

The runtime may subsequently perform

$$\mathsf{collapse}(v),$$

recording that the value has become part of the committed computational history.

This distinction permits speculative execution,

fuzzy reasoning,

symbolic evaluation,

and repair to coexist naturally.

Evaluation generates possibilities.

Collapse commits to one realization.

The execution history records both stages explicitly.

D.6 Parallel Execution

Parallel execution requires almost no modification of the runtime.
Instead of selecting a single executable node,
the scheduler selects an antichain

$$A \subseteq Q,$$

whose members possess no mutual dependencies.
Every node in

$$A$$

may execute simultaneously.
After all have completed,
their successor nodes are examined exactly as before.
Consequently,
parallelism is not encoded within the language.
It is discovered from graph topology.
The same graph therefore executes sequentially,
in parallel,
or across distributed hardware without changing its operational semantics.

D.7 Incremental Execution

Many practical systems repeatedly modify only small portions of a graph.
Recomputing the entire graph would be wasteful.
Instead,
the runtime marks only those nodes whose dependencies have changed.
Every unaffected region retains its previous histories.
Execution therefore becomes incremental.
Only computational consequences of the modification are reevaluated.
This mechanism naturally supports
interactive programming,
graphical editors,
continuous simulation,

spreadsheet computation,

and reactive user interfaces.

Incrementality emerges from dependency analysis rather than from specialized programming constructs.

D.8 Summary

The reference runtime described in this appendix is intentionally minimal.

Its responsibilities are limited to

maintaining a composition graph,

loading operator libraries,

scheduling executable nodes,

recording persistent histories,

processing refusal and collapse events,

and applying graph rewrites when requested.

Every richer computational phenomenon discussed throughout this essay—typing, theorem proving, compilation, optimization, fuzzy inference, symbolic reasoning, neural computation, repair, and learning—is realized by supplying different operator libraries and graph transformations rather than by enlarging the execution engine itself.

This economy is the principal engineering consequence of the graph-first philosophy. A small, mathematically uniform runtime can support an unexpectedly broad spectrum of computational paradigms because composition, rather than any particular programming language or logical system, is treated as the primitive computational operation.

E Toward a Composition-First Calculus

The previous appendices have developed the operational machinery of Spherepop using graphs, operator algebras, and histories. A natural question now arises: can these ideas themselves be organized into a formal calculus analogous to the lambda calculus or the calculus of constructions?

The answer proposed here is affirmative.

Rather than taking abstraction, application, or typing as primitive, we propose a composition-first calculus whose primitive judgments concern graph construction and history evolution.

E.1 Primitive Judgments

Most formal calculi begin with judgments such as

$$\Gamma \vdash t : T,$$

or

$$\Gamma \vdash P.$$

Spherepop instead begins with operational judgments.
The most primitive judgment is

$$H \vdash G \Downarrow H',$$

which should be read as

Executing graph

$$G$$

extends history

$$H$$

into history

$$H'.$$

Notice what is absent.
There are no variables.
No typing context.
No logical propositions.
Only graphs and histories.
Everything else will eventually be introduced as additional descriptive structure.

E.2 Primitive Rules

The calculus contains remarkably few primitive inference rules.
The identity rule states that the empty graph leaves history unchanged,

$$\frac{}{H \vdash \mathrm{Id} \Downarrow H.}$$

Composition becomes

$$\frac{H \vdash G_1 \Downarrow H_1 \qquad H_1 \vdash G_2 \Downarrow H_2}{H \vdash G_2 \circ G_1 \Downarrow H_2.}$$

This single rule replaces a large collection of language-specific evaluation rules.
Sequential execution,
function application,
pipeline execution,
module composition,
and hardware composition
all become instances of graph composition.

E.3 Operator Evaluation

Suppose

$$n$$

is an executable node whose operator is

$$f.$$

The evaluation rule becomes

$$\frac{v = f(v_1, \ldots, v_k)}{H \vdash n \Downarrow (H, e),}$$

where

$$e = (n, f, v).$$

Unlike ordinary operational semantics,
the output history explicitly records the computational event.
Evaluation therefore constructs history rather than merely producing values.

E.4 Refusal

Constraint systems,

proof assistants,

repair systems,

and interactive reasoning frequently require the ability to reject proposed continuations.

Spherepop treats refusal as an ordinary inference rule.

$$\frac{r \text{ is a refusal reason}}{H \vdash n \Downarrow (H, \mathsf{refuse}(r)).}$$

Refusal does not terminate the calculus.

Subsequent rules may branch,

repair,

or replace the rejected continuation.

This treatment removes the traditional distinction between successful computation and controlled failure.

Both become ordinary historical events.

E.5 Collapse

Similarly,

the calculus distinguishes evaluation from commitment.

Suppose

$$v$$

has already been produced.

Commitment becomes

$$\frac{v \text{ evaluated}}{H \vdash \mathsf{collapse}(v) \Downarrow (H, \mathsf{collapse}(v)).}$$

This explicit treatment of commitment provides a uniform account of speculative execution,

interactive computation,

symbolic reasoning,

and fuzzy evaluation.

The runtime may evaluate many possibilities while collapsing only selected histories.

E.6 The Conservativity Principle

An important property of the calculus is that richer computational systems extend rather than replace the primitive rules.

Typed systems add typing judgments.

Proof systems add proof judgments.

Dependent type theories add universe judgments.

Program logics add specification judgments.

None of these alters the primitive execution rules developed above.

They merely constrain which graphs are considered admissible.

This observation leads to what may be regarded as the central theorem of the appendix.

Theorem E.1 (Conservativity)**.** *Every extension of the composition-first calculus obtained by adding descriptive judgments while leaving the operational rules unchanged is conservative with respect to execution.*

Consequently,

types,

proofs,

logical systems,

and specification languages enrich computation without enlarging its primitive operational semantics.

This theorem formalizes one of the principal philosophical claims of the entire essay.

Execution precedes description.

Descriptions constrain execution.

They do not generate it.

E.7 The Composition-First Thesis

The resulting hierarchy may finally be stated in mathematical form.

Primitive operators generate graphs.

Graphs generate histories.

Histories admit composition,

repair,

optimization,

and learning.

Only after these operational structures exist do we introduce

syntax,

grammars,

types,

logical systems,

proof calculi,

dependent universes,

and mathematical foundations.

The resulting direction of explanation is therefore

$$\text{composition} \Longrightarrow \text{graphs} \Longrightarrow \text{histories} \Longrightarrow \text{execution} \Longrightarrow \text{optimization} \Longrightarrow \text{description} \Longrightarrow \text{verificati}$$

This ordering is deliberately opposite to that adopted by most presentations of programming language theory.

Rather than deriving execution from syntax,

Spherepop derives syntax from execution.

Rather than deriving computation from type theory,

it derives type theory from admissible patterns of composition.

Rather than deriving programs from formal languages,

it derives formal languages from the need to communicate composition graphs efficiently.

The composition-first calculus therefore serves not as a replacement for existing logical foundations, but as a proposed operational substrate beneath them. Lambda calculi, combinatory logic, type theories, proof assistants, and the calculus of constructions can all be interpreted as progressively richer descriptive systems whose common computational core is the composition of operators into persistent histories.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed., Addison–Wesley, 2007.

[2] J. Backus. Can Programming Be Liberated from the von Neumann Style? *Communications of the ACM*, 21(8):613–641, 1978.

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

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[5] R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, 1998.

[6] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Macmillan, 1976.

[7] R. V. Book and F. Otto. *String-Rewriting Systems*. Springer, 1984.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 4th ed., MIT Press, 2022.

[9] P. Cousot and R. Cousot. Abstract Interpretation. *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1977.

[10] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[11] G. Gonthier. Formal Proof—The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[12] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison–Wesley, 1989.

[13] F. Harary. *Graph Theory*. Addison–Wesley, 1969.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed., Morgan Kaufmann, 2019.

[15] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[16] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed., Addison–Wesley, 2006.

[17] D. E. Knuth. *The Art of Computer Programming*, Vol. 1. Addison–Wesley, 1968.

[18] S. Mac Lane. *Categories for the Working Mathematician*. 2nd ed., Springer, 1998.

[19] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[20] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[21] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[22] G. D. Plotkin. A Structural Approach to Operational Semantics. DAIMI Technical Report FN-19, Aarhus University, 1981.

[23] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. *Information Processing*, 1983.

[24] B. Russell and A. N. Whitehead. *Principia Mathematica*. Cambridge University Press, 1913.

[25] P. Selinger. A Survey of Graphical Languages for Monoidal Categories. In *New Structures for Physics*, Springer, 2011.

[26] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[27] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[28] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[29] P. Wadler. The Essence of Functional Programming. *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1992.

[30] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965.