Building a Type Calculus and the Calculus of Constructions in Spherepop

A Historical Foundation for Dependent Type Theory

Flyxion
Independent Researcher

June 2026

Abstract

Spherepop proposes an event-oriented foundation for constructive mathematics in which irreversible historical events, rather than mutable states or extensional set membership, constitute the primitive objects of reasoning. This paper develops a dependent type calculus directly from these principles and demonstrates that a complete Calculus of Constructions may be reconstructed within an event-native kernel. Instead of adopting the traditional viewpoint that contexts are unordered collections of assumptions, Spherepop interprets every derivation as a growing history of constructive events. Structural rules become transformations of histories, substitution becomes provenance-preserving historical substitution, equality becomes historical equivalence, and proof objects become immutable event traces that may be replayed to reconstruct their logical origin.

Beginning from a minimal event algebra, we derive historical typing judgments, dependent products, cumulative universes, polymorphism, equality types, inductive families, and a bidirectional type checking algorithm. We then establish the relationship between the historical calculus and the conventional Calculus of Constructions by introducing a history-erasure functor that recovers ordinary dependent type theory as a conservative abstraction. The resulting kernel unifies programming, proof construction, provenance, and event sourcing within a single mathematical framework while remaining sufficiently compact to serve as the trusted foundation of the Spherepop language.

Contents

1 Introduction

The design of a programming language is ultimately determined by the primitive objects from which it chooses to build its mathematics. Every proof assistant, compiler, and formal verification system inherits assumptions about identity, computation, and construction from its underlying logical calculus. These assumptions are often so deeply embedded that they become invisible. Variables are introduced into unordered contexts, functions are regarded as the primitive computational objects, and evaluation is understood as the reduction of expressions until a normal form is obtained. The surrounding logical machinery is then constructed upon these foundations.

Spherepop begins from a different observation. Computation does not merely transform symbolic expressions; it records the irreversible sequence of constructive events by which those expressions came into existence. Every meaningful object possesses a provenance. Every proof has a history. Every program is not merely a function from inputs to outputs but an accumulated record of admissible constructions whose order cannot, in general, be ignored.

The central hypothesis of this paper is therefore that historical construction, rather than state transformation or extensional membership, should be regarded as the primitive notion from which dependent type theory is derived. Instead of treating histories as metadata attached to an otherwise conventional proof calculus, Spherepop elevates histories to the status of first-class mathematical objects. Contexts become histories. Structural rules become operations upon histories. Equality becomes equivalence of constructive provenance. The familiar machinery of the lambda calculus is recovered as a disciplined method for composing historical constructions rather than as the primitive ontology of computation.

This change of perspective is subtle but significant. Conventional dependent type theories begin with judgments of the form

$$\Gamma \vdash t : A,$$

where Γ denotes a context consisting of assumptions. The context is typically regarded as an unordered collection satisfying structural rules such as weakening, exchange, and contraction. The semantics of the language are then determined by substitution and normalization within this context.

Spherepop instead begins with judgments of the form

$$H \vdash t : A,$$

where H is not an unordered context but an irreversible constructive history. A history is an ordered sequence of admissible events whose temporal structure contributes directly to the identity of every derived object. Consequently,

$$e_1; e_2 \neq e_2; e_1$$

unless the two events are demonstrably independent. Ordering therefore becomes an intrinsic mathematical property rather than a merely operational artifact.

This historical viewpoint aligns naturally with the philosophy that underlies the remainder of the Spherepop language. Programs are histories rather than mutable states. Data structures are accumulated constructions rather than containers whose contents are repeatedly overwritten. Deletion is represented by explicit refusal events rather than physical removal. Quotients become recorded collapse events. Parallel computation is represented through constructive melding of compatible histories rather than through implicit synchronization over mutable memory.

These observations suggest that the conventional structural rules of type theory should themselves be reinterpreted. Weakening corresponds to extending an admissible history without invalidating earlier derivations. Exchange becomes conditional upon historical independence. Substitution records provenance rather than merely replacing variables syntactically. Equality becomes a relation between histories instead of a primitive axiom imposed upon extensional objects.

Remarkably, these reinterpretations preserve the expressive power of ordinary dependent type theory. The dependent product remains the fundamental constructor from which ordinary functions, polymorphism, higher-order type operators, and dependent families all arise. Universes retain their cumulative hierarchy. Curry–Howard continues to identify propositions with types and proofs with programs. Beta reduction remains the computational engine of evaluation. What changes is not the computational content of the calculus but the ontology from which that content is derived.

Throughout this paper we therefore pursue two complementary objectives. The first objective is constructive. Beginning with a minimal algebra of historical events, we develop a complete dependent type calculus capable of supporting polymorphism, inductive families, equality types, and verified computation. The

second objective is comparative. We demonstrate that the resulting historical calculus forms a conservative extension of the traditional Calculus of Constructions through an explicit history-erasure functor that recovers ordinary dependent type theory whenever provenance is intentionally ignored.

The resulting system occupies an intermediate position between programming language semantics, proof theory, and event-sourced computation. Every theorem possesses both logical meaning and constructive provenance. Every computation simultaneously evaluates a term and extends a history. Every proof object becomes replayable, allowing logical correctness and historical origin to be reconstructed from the same immutable artifact.

From this perspective, the familiar lambda calculus is no longer the beginning of the story but one chapter within a broader theory of historical construction. The remaining sections develop this foundation systematically, beginning with the algebra of events from which every subsequent notion of typing, computation, and proof will be derived.

2 The Historical Ontology of Spherepop

The development of a formal calculus begins by specifying its primitive ontology. Classical set theory takes membership as primitive. Category theory begins with objects and morphisms. The lambda calculus begins with abstraction and application. Conventional dependent type theories adopt contexts, variables, and substitution as their irreducible notions. Spherepop instead begins with constructive events.

An event is not simply an operation performed during execution. It is the fundamental unit from which mathematical identity is assembled. Every object exists because some finite history of admissible events produced it. Consequently, the kernel of the language is concerned not with describing static mathematical objects but with describing the irreversible accumulation of constructive history.

This perspective deliberately separates identity from representation. Two values may possess identical observable behaviour while nevertheless differing in the histories that produced them. Conversely, two apparently distinct constructions may later become identified by an explicit historical collapse. Identity therefore becomes a dynamic notion governed by constructive provenance rather than an extensional predicate assumed a priori.

Definition 2.1 (Historical Event)**.** A *historical event* is an indivisible constructive

action that extends an admissible history. If H denotes an existing history and e denotes an event, then

$$H' = H; e$$

is called the historical extension of H by e.

Unlike state transitions, events are irreversible. Once an event has entered the historical record it is never removed. Later events may reinterpret, collapse, refine, or refuse earlier constructions, but they do not erase them. Histories therefore possess a monotone structure.

Axiom 2.2 (Historical Monotonicity)**.** For every admissible history H and every admissible event e,

$$H \subseteq H; e.$$

Every extension preserves the complete provenance of the previous history.

This monotonicity distinguishes historical reasoning from mutable computation. Traditional operational semantics overwrite machine states repeatedly during execution. Spherepop instead records every admissible transformation. Evaluation therefore accumulates information rather than replacing it.

The existence of a permanent historical record immediately changes the meaning of mathematical identity.

Suppose two objects x and y have respective construction histories

$$\text{History}(x) \quad \text{and} \quad \text{History}(y).$$

Spherepop regards these histories as primary.

Definition 2.3 (Historical Identity)**.** Two objects are historically identical whenever their constructive histories are equivalent:

$$x \equiv_H y \iff \text{History}(x) \simeq \text{History}(y),$$

where $\simeq$ denotes admissible historical equivalence.

Notice that this definition differs from extensional equality. Extensional equality compares completed objects. Historical equality compares the constructive processes that generated those objects. Extensional equality therefore appears only as a derived notion after historical equivalence has been established.

The distinction becomes particularly important for proof objects. Two proofs may establish precisely the same theorem while exhibiting entirely different constructions. Conventional proof assistants frequently erase this distinction by normalizing both proofs to equivalent terms. Spherepop instead preserves both derivations as independent historical artifacts whose provenance remains available for later inspection, optimization, or replay.

A history itself is defined recursively.

Definition 2.4 (Constructive History)**.** The set of histories is generated inductively by

$$H ::= \varepsilon \mid H; e,$$

where ε denotes the empty history and e denotes an admissible event.

The recursive definition is intentionally minimal. No assumptions have yet been made regarding the nature of admissible events. Those operations will be introduced systematically in the following section. At this stage the only structure imposed upon histories is sequential composition.

Sequential composition is associative.

$$(H; e_1); e_2 = H; (e_1; e_2),$$

allowing parentheses to be omitted whenever no ambiguity arises. In contrast, commutativity generally fails.

$$H; e_1; e_2 \neq H; e_2; e_1.$$

The order of construction contributes directly to the mathematical identity of every derived object. Only under carefully specified independence conditions will later sections permit historical exchange.

This asymmetry has important logical consequences. Conventional structural proof theory often treats contexts as multisets whose ordering is irrelevant. Spherepop rejects this assumption at the foundational level. Dependencies are not merely relations among variables but among historical events. Every variable introduced into the system possesses a definite origin, and subsequent constructions inherit that provenance.

The ontology therefore distinguishes between two kinds of mathematical object. The first consists of observable terms, types, proofs, and programs. The

second consists of the histories that generated those observable constructions. Ordinary dependent type theory concerns itself almost exclusively with the former. Spherepop treats both as equally fundamental.

One may therefore regard the observable language as the visible projection of a richer historical substrate. Computation manipulates observable terms while simultaneously extending the hidden but persistent historical record. Proof checking validates both the logical correctness of a derivation and the admissibility of its constructive history. Every theorem consequently possesses two inseparable components: its mathematical content and its provenance.

The remaining development of the calculus proceeds by enriching this minimal ontology with an algebra of primitive historical operations. These operations will serve the role traditionally occupied by structural rules, quotient constructions, mutable updates, and synchronization primitives, while preserving the central principle that every admissible computation is represented by an irreversible extension of constructive history.

3 The Event Algebra

Having established histories as the primitive mathematical objects of the Spherepop kernel, we now specify the elementary operations by which histories are extended. These operations form the constructive alphabet from which every program, proof, derivation, and data structure is assembled. They occupy a role analogous to the primitive inference rules of proof theory or the primitive combinators of the lambda calculus. Unlike those systems, however, the operations of Spherepop manipulate histories rather than isolated symbolic expressions.

The objective is not merely to define a programming language. Rather, the event algebra specifies the smallest collection of irreversible constructive actions from which the remainder of the type calculus may be derived. Every higher-level language feature is ultimately translated into combinations of these primitive historical events.

3.1 Primitive Historical Operations

Let

$$H$$

denote an admissible history. Every primitive operation accepts one or more histories and produces another admissible history.

The kernel introduces five primitive historical constructors,

$$\mathsf{Pop}, \qquad \mathsf{Refuse}, \qquad \mathsf{Bind}, \qquad \mathsf{Collapse}, \qquad \mathsf{Meld}.$$

Additional language constructs are regarded as derived operations obtained by composition of these primitives.

3.2 The Pop Operation

The first primitive operation records constructive commitment.

During computation there frequently exist several admissible continuations. A choice among these possibilities should itself become part of the permanent historical record. Rather than silently selecting one branch, Spherepop records the decision explicitly.

Definition 3.1 (Pop)**.** Suppose

$$b$$

is an admissible continuation from history

$$H.$$

The operation

$$\mathsf{Pop}(H, b) = H'$$

extends the history by recording that branch

$$b$$

has been constructively selected.

The operation does not merely evaluate a branch. It records the commitment to that branch as part of the mathematical identity of all future constructions.

Consequently,

$$\mathsf{Pop}(H, b_1) \neq \mathsf{Pop}(H, b_2)$$

whenever

$$b_1 \neq b_2.$$

Even if both branches later compute identical observable values, their provenance remains distinct until an explicit collapse event identifies them.

3.3 The Refuse Operation

Selection alone is insufficient to describe constructive computation. Equally important is the permanent rejection of alternatives.

Traditional programming languages usually represent rejection implicitly through failed conditions or discarded execution paths. Spherepop instead promotes refusal to a primitive historical operation.

Definition 3.2 (Refuse)**.** Suppose

$$a$$

is an admissible alternative under history

$$H.$$

Then

$$\mathsf{Refuse}(H, a) = H'$$

extends the history by permanently recording that

$$a$$

has been constructively rejected.

Unlike logical negation, refusal is historical rather than truth-functional.

Negation merely states that a proposition does not hold.

Refusal records the constructive act by which a possibility was intentionally excluded.

Future reasoning may therefore inspect not only what was chosen but also what was deliberately abandoned.

This distinction becomes particularly valuable during proof search, backtracking, optimization, and distributed computation, where the rejected alternatives may themselves carry useful provenance.

3.4 The Bind Operation

Dependency is introduced through historical binding.

Suppose

$$x : A$$

and

$$f : A \to B.$$

Instead of viewing the application of

$$f$$

to

$$x$$

as an isolated symbolic substitution, Spherepop records the dependency itself.

Definition 3.3 (Bind)**.** The historical binding operation is

$$\mathsf{Bind}(H, x, f) = H'$$

where

$$H'$$

records that every subsequent construction of

$$f(x)$$

depends historically upon

$$x.$$

This dependency graph becomes part of the provenance of every future construction.

Consequently, substitutions performed later in the type checker preserve not only values but also dependency histories.

3.5 The Collapse Operation

Many mathematical constructions eventually identify objects that were initially constructed independently.

Set theory introduces quotient sets.

Type theory introduces definitional equality.

Category theory introduces coequalizers.

Spherepop instead records explicit historical collapse.

Definition 3.4 (Collapse)**.** Given two constructions

$$a$$

and

$$b,$$

the operation

$$\mathsf{Collapse}(H, a, b) = H'$$

extends the history with the constructive assertion that

$$a$$

and

$$b$$

are thereafter regarded as historically equivalent.

Notice that no previous provenance is erased.

Both construction histories remain available for inspection.

The collapse merely records an additional event identifying them.

Consequently,

$$\text{History}(a)$$

and

$$\text{History}(b)$$

remain individually recoverable even after the collapse has occurred.

This explicit treatment of identification will later provide the basis for historical equality types and quotient constructions.

3.6 The Meld Operation

Constructive histories need not evolve sequentially.

Independent computations may proceed concurrently before eventually being combined.

Rather than representing concurrency through mutable shared state, Sphere-pop introduces a primitive historical merge operation.

Definition 3.5 (Meld)**.** Suppose

$$H_1$$

and

$$H_2$$

are compatible histories.

Their constructive merge is

$$\mathsf{Meld}(H_1, H_2),$$

which produces a history containing the provenance of both components while preserving every dependency relation.

Compatibility is intentionally left abstract at this stage.

Later chapters define admissibility conditions under which two histories may be melded without violating dependency constraints.

Operationally, Meld generalizes event sourcing, distributed logs, version control, and concurrent proof construction.

Mathematically, it resembles categorical pushouts while remaining entirely constructive.

3.7 Closure Under Historical Composition

The primitive operations introduced above are not isolated commands but constructors of a historical algebra.

Every finite computation is represented by repeated application of these constructors.

Thus the collection of admissible histories is generated inductively by

$$H ::= \varepsilon \mid H; e \mid \mathsf{Pop}(H, b) \mid \mathsf{Refuse}(H, a) \mid \mathsf{Bind}(H, x, f) \mid \mathsf{Collapse}(H, a, b) \mid \mathsf{Meld}(H_1, H_2).$$

This grammar should not be interpreted merely as syntax.

Each constructor possesses operational meaning, typing rules, normalization behavior, and historical semantics that will be developed throughout the remainder of this paper.

Taken together, these five primitive operations form the constructive substrate upon which the Spherepop kernel is built. Conventional notions such as mutable assignment, quotienting, synchronization, substitution, branching, and proof construction emerge as derived concepts within this historical algebra rather than appearing as independent primitives. The subsequent section introduces the notion of historical contexts and demonstrates how the event algebra replaces the unordered assumptions of conventional dependent type theory.

4 Historical Contexts

The introduction of the event algebra permits a fundamental reexamination of the role played by contexts in dependent type theory. Conventional presentations of the lambda calculus and the Calculus of Constructions begin with a typing context

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

whose purpose is to record the assumptions under which a judgment is derived. Although the notation suggests an ordering, most systems subsequently introduce

structural rules showing that the order is largely irrelevant. Weakening, exchange, and contraction together imply that contexts behave much like finite multisets of assumptions.

Spherepop adopts a different interpretation.

A context is not merely a collection of assumptions. It is an irreversible construction history. Every declaration represents an event whose position within the historical record contributes to the identity of all later derivations. Consequently, contexts cease to be unordered environments and instead become temporally ordered histories generated by the event algebra of the previous section.

4.1 Historical Judgments

The fundamental typing judgment of the Spherepop kernel has the familiar syntactic appearance

$$H \vdash t : A,$$

but its interpretation differs substantially from the conventional reading.

Rather than asserting that the term t has type A under a collection of assumptions, the judgment states that the construction of t as an inhabitant of A is admissible after the historical sequence H has occurred.

Thus a typing derivation simultaneously establishes two facts.

First, the term is well typed.

Second, the historical evolution that produced the term is itself admissible.

Logical correctness and constructive provenance therefore become inseparable.

4.2 Historical Formation

The empty history is denoted by

$$\varepsilon.$$

It represents the absence of prior constructive events.

Every non-empty history is generated recursively by historical extension,

$$H' = H; e,$$

where e denotes an admissible event.

Accordingly,

$$\varepsilon \subseteq H \subseteq H; e \subseteq H; e; e' \subseteq \cdots$$

forms a monotone chain of historical growth.

No operation of the kernel removes events from an existing history.

Every theorem therefore possesses a complete constructive provenance extending back to the empty history.

4.3 Historical Declarations

Variable declarations become historical events.

Instead of viewing

$$x : A$$

as a static assumption inserted into an environment, Spherepop interprets it as an irreversible declaration event,

$$e_x = \text{Declare}(x : A).$$

Appending the declaration extends the history,

$$H \longmapsto H; \text{Declare}(x : A).$$

Variables therefore acquire historical identity.

Every subsequent reference to x may be traced uniquely to the declaration event that introduced it.

Two variables having identical observable types but distinct declaration events remain historically distinguishable.

4.4 Historical Scope

Lexical scope acquires a historical interpretation.

A variable is available precisely when its declaration occurs earlier than every event depending upon it.

If

$$e_x < e_t,$$

where e_t denotes the construction of a term t, then

$$t$$

may legitimately reference

$$x.$$

Conversely, attempting to reference a declaration that has not yet occurred violates historical admissibility.

This interpretation replaces the traditional notion of environments with an explicit dependency ordering embedded directly into the event history.

4.5 Historical Reachability

Every object possesses a unique construction path through the history.

Definition 4.1 (Historical Reachability)**.** Let

$$H = e_1; e_2; \cdots ; e_n.$$

A construction c is historically reachable if there exists a finite sequence of admissible dependencies

$$e_{i_1} \prec e_{i_2} \prec \cdots \prec e_{i_k}$$

whose final event produces c.

Reachability therefore replaces the informal notion of "being in scope."

Objects are available precisely because their construction histories are reachable from the current historical state.

This perspective aligns naturally with the broader philosophy of Spherepop, in which accessibility is determined by constructive reachability rather than membership in an abstract environment.

4.6 Dependency Graphs

Although histories are recorded sequentially, their internal dependency structure forms a directed acyclic graph.

Each historical event contributes one or more dependency edges,

$$e_i \longrightarrow e_j,$$

indicating that the construction represented by

$$e_j$$

depends upon the earlier event

$$e_i.$$

The sequential ordering guarantees acyclicity, while the dependency graph captures the finer structure of constructive provenance.

Thus every history admits two complementary descriptions.

The first is the chronological sequence

$$e_1; e_2; \cdots ; e_n.$$

The second is the induced dependency graph

$$(H, \text{Depends}).$$

The sequential representation preserves temporal order, whereas the graph representation exposes logical dependence.

Both descriptions encode the same historical object.

4.7 Observable Contexts as Projections

Conventional typing contexts may now be recovered as projections of historical contexts.

Define the projection operator

$$\pi_\Gamma : H \to \Gamma$$

to erase every event except variable declarations and type assumptions.

The resulting context

$$\Gamma = \pi_\Gamma(H)$$

is precisely the environment expected by an ordinary dependent type checker.

Consequently, traditional contexts are not discarded but derived.

They represent observable summaries of richer historical structures whose additional provenance information has been intentionally forgotten.

This observation foreshadows one of the principal results of the present paper: ordinary dependent type theory will emerge as the image of the historical calculus under an appropriate erasure functor.

4.8 Historical Well-Formedness

Not every finite sequence of events constitutes a valid history.

A history must satisfy several admissibility conditions.

Every dependency must reference an earlier event.

Every declaration must precede its use.

Every collapse must identify previously constructed objects.

Every meld must preserve dependency consistency.

These conditions collectively define the class of well-formed histories.

Definition 4.2 (Well-Formed History)**.** A history H is well formed if every event occurring in H satisfies the admissibility constraints imposed by the event algebra and all dependency relations are acyclic.

The remainder of the calculus assumes that every typing judgment is interpreted relative to a well-formed historical context.

This assumption replaces the conventional requirement that typing contexts be well formed, while simultaneously encoding considerably richer structural information.

Having established histories as the environments in which every derivation takes place, we are now prepared to reconsider the structural rules of dependent type theory themselves. The next section shows that weakening, exchange, contraction, and substitution are no longer primitive logical rules but emerge as admissible transformations of constructive histories.

5 Structural Rules as Historical Transformations

One of the most distinctive features of conventional type theory is its reliance upon structural rules. These rules determine how contexts may be manipulated independently of the logical connectives themselves. Weakening allows unused assumptions to be introduced. Exchange permits assumptions to be reordered. Contraction identifies duplicate assumptions. Substitution propagates completed constructions throughout a derivation.

Historically, these rules originate in Gentzen's structural proof theory, where contexts are regarded as collections of assumptions rather than constructive objects in their own right. Since assumptions possess no intrinsic temporal ordering, they may be duplicated, permuted, or discarded without altering the meaning of a proof.

The historical interpretation developed in the preceding sections fundamentally changes this situation.

A Spherepop context is an irreversible construction history. Every declaration, branch selection, collapse, refusal, and dependency contributes to the identity of the derivation. Consequently, structural rules can no longer be regarded as axiomatic manipulations of an unordered environment. Instead, they become properties of admissible historical transformations.

This shift from syntactic manipulation to historical transformation constitutes one of the principal conceptual differences between the Spherepop kernel and conventional dependent type theory.

5.1 Historical Extension

The most basic structural operation is historical growth.

Suppose

$$H$$

is a well-formed history and

$$e$$

is an admissible event independent of an existing derivation.

Appending

$$e$$

cannot invalidate constructions that have already been completed.

This observation gives rise to the historical analogue of weakening.

Theorem 5.1 (Historical Weakening)**.** *If*

$$H \vdash t : A$$

and

$$e$$

is an admissible extension that introduces no dependency upon the derivation of t*, then*

$$H; e \vdash t : A.$$

Proof. The derivation of

$$t$$

depends only upon events already contained in

$$H.$$

Since the appended event introduces no new dependencies into the construction of

$$t,$$

the dependency graph of the original derivation remains unchanged. Historical monotonicity therefore preserves the typing judgment.

□

Unlike ordinary weakening, this theorem is not simply an axiom of context formation. It is a consequence of the monotonic nature of historical construction.

5.2 Conditional Exchange

Conventional proof theory assumes that assumptions may be reordered freely.

For example,

$$x : A,\ y : B$$

is interchangeable with

$$y : B,\ x : A.$$

Spherepop rejects unrestricted exchange.

If one declaration depends upon another, reversing their order destroys the construction history that produced the derivation.

Instead, exchange becomes conditional upon historical independence.

Definition 5.2 (Historical Independence)**.** Two events

$$e_i$$

and

$$e_j$$

are historically independent if neither depends directly or indirectly upon the other.

We write

$$e_i \perp e_j.$$

Only under this condition may their order be exchanged.

Theorem 5.3 (Conditional Exchange)**.** *Suppose*

$$e_i \perp e_j.$$

Then

$$H; e_i; e_j \equiv H; e_j; e_i.$$

Thus exchange becomes a derived theorem rather than a primitive structural rule.

The dependency graph determines precisely when commutation is admissible.

This interpretation resembles Mazurkiewicz traces and partially ordered event structures, where concurrency is characterized by commutation of independent events rather than by arbitrary permutation.

5.3 Historical Contraction

Ordinary contraction identifies duplicate assumptions.

For example,

$$x : A,\ x : A$$

may be replaced by a single occurrence.

Such a rule is appropriate when contexts merely describe logical assumptions.

Historical contexts, however, distinguish between repeated assumptions and repeated construction events.

Two declaration events may introduce observably identical objects while possessing distinct provenance.

Consequently,

$$H; e; e$$

is generally not equivalent to

$$H; e.$$

Instead, contraction is replaced by explicit historical collapse.

If two constructions are intended to become identical, the kernel records

$$\mathsf{Collapse}(H, a, b),$$

thereby preserving both original histories while introducing a new event that constructively identifies them.

Duplicate constructions are therefore never silently erased.

They become historically related through explicit evidence.

5.4 Historical Erasure

Conventional proof systems frequently remove unused assumptions.

Within the Spherepop kernel such removal cannot occur internally because every event contributes to historical provenance.

Instead, removal becomes an external projection.

Define the history-erasure operator

$$\mathcal{E} : H \to \Gamma,$$

which forgets every event except the observable declarations required for type checking.

This operator is not part of the trusted kernel.

Rather, it is an abstraction used when comparing the historical calculus with ordinary dependent type theory.

Consequently, information may be forgotten by external observers without ever being destroyed inside the constructive history itself.

5.5 Structural Preservation

Because structural transformations are interpreted as historical operations, their admissibility follows from the dependency graph.

Proposition 5.4. *Every admissible structural transformation preserves the dependency graph up to historical equivalence.*

Proof. Weakening extends the graph without modifying existing edges.

Conditional exchange merely permutes incomparable vertices.

Historical collapse introduces an identification event while leaving previous vertices intact.

History erasure removes observable information only after construction has been completed.

Each transformation therefore preserves the dependency relation modulo the appropriate notion of historical equivalence.

□

This proposition replaces the informal intuition that contexts may be manipulated "without affecting the proof." In the historical calculus every transformation has explicit constructive meaning.

5.6 Structural Rules as Derived Computation

The reinterpretation developed in this section has an important philosophical consequence.

Traditional proof theory separates logical rules from structural rules.

Logical rules manipulate propositions.

Structural rules manipulate contexts.

Spherepop removes this distinction.

Contexts themselves are mathematical objects generated by constructive events.

Consequently, every structural rule becomes another computational process acting upon histories.

Weakening is historical extension.

Exchange is conditional commutation.

Contraction is constructive collapse.

Erasure is external projection.

None of these operations exists independently of the historical semantics.

The familiar structural calculus therefore emerges as a special case of a more general algebra of constructive histories. Having established this reinterpretation, we next examine substitution itself. Rather than treating substitution as purely syntactic replacement, the Spherepop kernel will redefine it as a provenance-preserving transformation that transports both values and their construction histories simultaneously.

6 Historical Substitution

Substitution occupies a central position in every formulation of dependent type theory. The lambda calculus derives its computational behavior almost entirely from beta reduction, while dependent types rely upon substitution to instantiate families of types and propositions. Nearly every metatheoretic result, including preservation, normalization, and subject reduction, ultimately depends upon the properties of substitution.

In conventional presentations, substitution is regarded as a purely syntactic operation. Given a term

$$t,$$

a variable

$$x,$$

and another term

$$a,$$

one simply writes

$$t[x := a],$$

meaning that every free occurrence of

$$x$$

is replaced by

$$a.$$

The correctness of the operation depends only upon avoiding variable capture. Once substitution has been performed, the origin of the substituted object plays no further role in the formal system.

Spherepop adopts a richer interpretation.

A substituted object is not merely a symbolic expression but a completed historical construction. Replacing one variable by another therefore transports not only observable syntax but also the constructive provenance responsible for that syntax. Historical substitution is consequently an operation upon histories rather than upon expressions alone.

6.1 Historical Environments

Suppose

$$H \vdash a : A$$

has already been established.

The derivation of

$$a$$

is itself represented by a finite dependency graph embedded inside the history

$$H.$$

Introducing

$$a$$

into another derivation should therefore preserve this dependency structure rather than discarding it.

Accordingly, substitutions are represented by historical environments

$$\sigma = \{x_1 \mapsto (a_1, H_1),\ x_2 \mapsto (a_2, H_2),\ \ldots\},$$

where every substituted object is paired with the history that constructed it.

The environment records both semantic value and constructive provenance.

6.2 Historical Substitution

Instead of the ordinary notation

$$t[x := a],$$

the Spherepop kernel employs

$$t[H, x \mapsto a],$$

which denotes substitution relative to a specific constructive history.

The notation emphasizes that substitution is never history-free.

Every replacement occurs within an admissible historical context that determines the provenance inherited by the resulting construction.

Definition 6.1 (Historical Substitution)**.** Suppose

$$H \vdash a : A$$

and

$$H; \mathsf{Declare}(x : A) \vdash t : B.$$

The historical substitution

$$t[H, x \mapsto a]$$

is obtained by replacing every free occurrence of

$$x$$

with

$$a,$$

while simultaneously extending every dependency of

$$t$$

to include the construction history of

$$a.$$

Unlike syntactic substitution, this operation modifies both the observable term and its underlying dependency graph.

The resulting object therefore possesses a richer provenance than either of its components considered separately.

6.3 Dependency Preservation

Historical substitution satisfies a fundamental invariance property.

Theorem 6.2 (Dependency Preservation)**.** *Let*

$$D(t)$$

denote the dependency graph of

$$t.$$

If

$$t' = t[H, x \mapsto a],$$

then

$$D(t') = D(t) \cup D(a),$$

together with the dependency edges introduced by the substitution itself.

Proof. Every dependency previously attached to

$$t$$

remains unchanged.

Every dependency required for constructing

$$a$$

must also remain available because the substituted object is now part of the construction of

$$t'.$$

The dependency graph is therefore the union of the original graphs together with the edges connecting the substitution site to the substituted construction.

□

Thus substitution never destroys provenance.

It enlarges the constructive explanation carried by the resulting term.

6.4 Compositionality

Multiple substitutions compose naturally.

Suppose

$$\sigma_1 = \{x \mapsto (a, H_a)\},$$

and

$$\sigma_2 = \{y \mapsto (b, H_b)\}.$$

Then

$$t[\sigma_1][\sigma_2]$$

is equivalent to

$$t[\sigma_2 \circ \sigma_1]$$

provided the substitutions satisfy the usual capture-avoidance conditions and their dependency graphs are compatible.

Composition therefore preserves both semantic correctness and historical provenance.

6.5 Substitution and Beta Reduction

The historical interpretation modifies the understanding of beta reduction without changing its observable computational result.

Ordinary beta reduction is written

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

Spherepop instead interprets the reduction as

$$(H, (\lambda x.t)\, a) \longrightarrow (H', t[H, x \mapsto a]),$$

where

$$H'$$

extends

$$H$$

by recording the substitution event itself.

Evaluation therefore becomes simultaneously

(i) construction of a new observable term,

(ii) construction of a new historical event documenting how that term arose.

Every beta reduction therefore contributes permanently to constructive provenance.

6.6 Historical Capture Avoidance

Traditional substitution must avoid accidental variable capture.

The historical calculus imposes an additional requirement.

Not only must variables remain distinct, but historical identities must remain distinct as well.

Suppose two variables possess identical observable names while originating from different declaration events,

$$e_x \neq e'_x.$$

The substitution algorithm distinguishes them by their provenance rather than by their textual spelling alone.

Consequently, alpha conversion becomes a historical renaming operation acting upon declaration events instead of merely upon symbols.

This interpretation naturally accommodates hygienic macros, proof replay, distributed proof construction, and incremental compilation because every bound variable carries a globally unique historical identity.

6.7 Historical Substitution Lemma

The ordinary substitution lemma extends directly to the historical setting.

Theorem 6.3 (Historical Substitution Lemma)**.** *Suppose*

$$H \vdash a : A$$

and

$$H; \mathsf{Declare}(x : A) \vdash t : B.$$

Then

$$H \vdash t[H, x \mapsto a] : B[H, x \mapsto a].$$

Furthermore, the dependency graph of the resulting derivation is precisely the historical composition of the dependency graphs of

$$t$$

and

$$a.$$

This theorem plays the same foundational role in the historical calculus that the ordinary substitution lemma plays in conventional dependent type theory. Virtually every subsequent metatheoretic argument will rely upon it.

The difference is that substitution now preserves considerably more information than observable syntax alone. It transports complete constructive provenance, making proof replay, dependency analysis, incremental recompilation, and history-sensitive verification natural consequences of the kernel rather than external implementation techniques.

With historical substitution established, we are now prepared to introduce the computational core of the language. The next section reconstructs the lambda calculus itself as a calculus of historical construction, showing that abstraction and application emerge naturally from the event algebra rather than serving as primitive ontological notions.

7 The Historical Lambda Calculus

The preceding sections have deliberately postponed discussion of the lambda calculus. This ordering differs from nearly every traditional presentation of type theory. Conventionally, abstraction and application are regarded as the primitive computational operations from which the remainder of the calculus is constructed. Histories, if they appear at all, are viewed merely as operational artifacts generated during evaluation.

The philosophy of Spherepop reverses this relationship.

Histories are primitive.

Lambda abstraction is a derived mechanism for constructing reusable historical transformations.

Functions are therefore understood not as timeless mathematical objects but as persistent construction procedures whose repeated application produces new historical events while preserving their internal provenance.

This interpretation does not alter the observable behavior of the lambda calculus. Instead, it enriches its semantics by embedding every computational step within the irreversible historical record.

7.1 Terms

The historical lambda calculus is generated inductively by the grammar

$$t, u, A, B ::= x \mid \lambda x : A.\, t \mid t\, u \mid \Pi x : A.\, B \mid \mathsf{Prop} \mid \mathsf{Type}_i.$$

This syntax is intentionally almost identical to the ordinary Calculus of Constructions.

The distinction lies not in the observable language but in the interpretation assigned to each constructor.

Every term possesses an associated construction history

$$\mathrm{History}(t),$$

which records the sequence of events by which the term entered the kernel.

Thus the semantic object is more accurately represented by the pair

$$(H, t),$$

rather than by the term

$$t$$

alone.

Observable syntax is therefore regarded as the visible projection of a richer historical construction.

7.2 Historical Variables

Variables are introduced through declaration events.

Suppose

$$e_x = \text{Declare}(x : A).$$

The variable

$$x$$

is thereafter identified not merely by its textual name but by the declaration event that introduced it.

Consequently,

$$x_{e_1} \neq x_{e_2}$$

whenever

$$e_1 \neq e_2,$$

even if both declarations assign the same observable type.

This historical interpretation eliminates accidental ambiguity arising from shadowing, renaming, or repeated declarations.

Variables become globally unique historical references rather than locally scoped textual symbols.

7.3 Historical Abstraction

Lambda abstraction records the construction of a reusable transformation.

Conventionally,

$$\lambda x : A.\, t$$

is interpreted as an anonymous function.

Spherepop instead interprets abstraction as the construction of a historical template capable of generating future histories.

Definition 7.1 (Historical Abstraction)**.** Suppose

$$H; \mathsf{Declare}(x : A) \vdash t : B.$$

Then

$$H \vdash \lambda x : A.\, t : \Pi x : A.\, B$$

constructs a reusable historical transformation whose future applications inherit the dependency graph of the original derivation.

The abstraction therefore captures not only symbolic structure but also constructive provenance.

Applying the function later does not erase this provenance.

Instead, it extends it.

7.4 Historical Application

Application instantiates a historical template with a completed construction.

Suppose

$$H \vdash f : \Pi x : A.\, B$$

and

$$H \vdash a : A.$$

Application produces

$$f\, a,$$

but simultaneously records a new historical event connecting the construction history of

$$a$$

with the dependency graph already contained inside

$$f.$$

Operationally,

$$(H, f\ a)$$

becomes

$$(H', f\ a),$$

where

$$H' = H; \mathsf{Bind}(H, a, f).$$

Application is therefore not merely evaluation.
It is also the explicit construction of a new dependency relation.

7.5 Historical Beta Reduction

Beta reduction remains the computational heart of the calculus.

The observable reduction rule is unchanged,

$$(\lambda x : A.\, t)\ a \longrightarrow t[x := a].$$

The historical semantics, however, are richer.
Reduction is represented by

$$(H, (\lambda x : A.\, t)\ a) \longrightarrow (H', t[H, x \mapsto a]),$$

where

$$H' = H; \mathsf{Bind}(H, a, t); \mathrm{Reduce}\,.$$

The reduction event becomes part of the permanent historical record.

Repeated normalization therefore produces not merely a normal form but also a complete explanation of every computational step that produced it.

7.6 Historical Eta Conversion

Eta conversion expresses extensional equivalence between functions.

Conventionally,

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

Spherepop distinguishes observable behavior from historical provenance.
Observable equality may hold while historical identity does not.
Instead,

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

and

$$f$$

remain distinct historical constructions until an explicit collapse event identifies them.

Thus eta conversion becomes an admissible historical equivalence rather than an automatic syntactic simplification.

This preserves the provenance of optimization steps while allowing extensional reasoning whenever required.

7.7 Historical Normal Forms

Normalization no longer produces only a simplified term.

Instead it computes a pair

$$(H, t) \Downarrow (H', v),$$

where

$$v$$

is the observable normal form and

$$H'$$

is the completed construction history that generated

$$v.$$

The observable computation and its provenance therefore terminate together.

Proof replay simply reconstructs

$$H'$$

and verifies that each recorded reduction is historically admissible.

7.8 Lambda Calculus as Historical Composition

The reinterpretation developed above reveals that abstraction and application are fundamentally operations upon histories.

Abstraction constructs reusable historical templates.

Application composes existing histories.

Beta reduction performs historical substitution.

Eta conversion identifies equivalent historical transformations through explicit collapse.

The ordinary lambda calculus is therefore recovered without alteration of its observable syntax, yet every computational step now carries complete constructive provenance.

This perspective provides a natural bridge between functional programming, proof construction, event sourcing, and dependency analysis. Rather than viewing functions as timeless mappings between values, Spherepop regards them as persistent generators of historical construction. The familiar lambda calculus thus emerges as a special case of a more general calculus of irreversible events.

With the computational interpretation established, we next introduce dependent products and show that the historical calculus naturally reconstructs the full expressive hierarchy of the Calculus of Constructions.

8 Dependent Products and Historical Families

The dependent product is the central constructor of modern dependent type theory. It simultaneously generalizes ordinary function spaces, universal quantification, polymorphism, higher-order type operators, and families of types indexed by values. Nearly the entire expressive power of the Calculus of Constructions arises from repeated application of a single constructor,

$$\Pi.$$

From the perspective developed in the preceding chapters, however, the dependent product is no longer introduced as a primitive logical connective. Instead, it emerges as the canonical method for describing the evolution of constructive histories. A dependent product specifies not merely which values may be constructed, but how future histories are permitted to extend existing ones.

Consequently, a dependent type should be interpreted as a family of admissible historical continuations rather than merely as a family of sets.

8.1 Historical Families

Suppose

$$H \vdash A : \mathsf{Type}_i.$$

A dependent family assigns to every admissible inhabitant

$$a : A$$

a type

$$B(a),$$

whose construction may itself depend upon both the value of

$$a$$

and the history that produced it.

Rather than writing simply

$$B(a),$$

the historical interpretation is more accurately expressed as

$$B(H, a),$$

emphasizing that type formation depends upon constructive provenance as well as observable value.

The explicit reference to

$$H$$

will usually be omitted when the surrounding history is clear from context, but its semantic role remains fundamental throughout the calculus.

8.2 Formation of Dependent Products

Suppose

$$H \vdash A : \mathsf{Type}_i$$

and

$$H; \mathrm{Declare}(x : A) \vdash B : \mathsf{Type}_j.$$

The dependent product is formed by the judgment

$$H \vdash \Pi x : A.\, B : \mathsf{Type}_{\max(i,j)}.$$

This rule differs from the conventional presentation only in its interpretation. The declaration of

$$x$$

is itself a historical event.
The family

$$B$$

therefore describes every admissible continuation of the history following that declaration.

Consequently,

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

classifies historical construction procedures rather than abstract mappings between extensional collections.

8.3 Ordinary Function Spaces

Whenever

$$x$$

does not occur free in

$$B,$$

the dependent product reduces to the ordinary function space

$$A \to B.$$

Observable syntax therefore remains identical to conventional type theory.

Historically, however, even ordinary functions are interpreted as procedures that transform histories.

Thus

$$f : A \to B$$

is more accurately regarded as

$$f : (H, A) \longrightarrow (H', B),$$

where

$$H'$$

extends

$$H$$

through the constructive events performed during evaluation.

Every function therefore carries an implicit history transformer.

8.4 Dependent Construction

The elimination rule for dependent products follows the familiar pattern.

Suppose

$$H \vdash f : \Pi x : A.\, B$$

and

$$H \vdash a : A.$$

Application produces

$$f\ a : B[a/x].$$

Historically this judgment should be interpreted as

$$(H, f) \otimes (H, a) \longmapsto (H', f\ a),$$

where

$$H'$$

contains the merged dependency graphs of both constructions together with the historical substitution event connecting them.

The dependent product therefore governs both logical dependence and historical inheritance simultaneously.

8.5 Historical Universality

Dependent products also recover universal quantification through the Curry–Howard correspondence.

Conventionally,

$$\forall x : A.\, P(x)$$

is represented as

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

Spherepop retains this identification but enriches its meaning.

A proof of

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

constructs not merely a function returning proofs.

It constructs a reusable historical procedure capable of extending every admissible construction of

$$x$$

into a corresponding construction of

$$P(x).$$

Universal quantification therefore becomes universal historical continuation.

8.6 Historical Polymorphism

One of the most remarkable properties of dependent products is their ability to describe polymorphic functions.

The polymorphic identity function possesses the type

$$\Pi A : \mathsf{Type}_0.\, A \to A.$$

Within the historical calculus this judgment states that every admissible type construction determines a corresponding historical identity transformation.

The proof term

$$\lambda A : \mathsf{Type}_0.\, \lambda x : A.\, x$$

contains almost no observable computation.

Nevertheless it records a universal construction template whose future applications inherit the declaration history of every supplied type.

Thus polymorphism itself becomes historical rather than purely syntactic.

8.7 Historical Type Operators

Dependent products also permit types to depend upon other types.

For example,

$$\lambda A : \mathsf{Type}_i.\, A \to A$$

constructs a type operator.

Historically this operator maps one admissible history of type construction to another.

Type-level computation therefore participates in the same event algebra already developed for ordinary terms.

No distinction is required between computation at the value level and computation at the type level.

Both are governed by the same historical semantics.

8.8 Historical Families of Data

Dependent products naturally describe indexed data structures.

Consider the familiar example

$$\Pi n : \mathsf{Nat}.\, \mathsf{Vector}(A, n).$$

Conventional type theory interprets this as a family of vector types indexed by their length.

Spherepop interprets it as a family of admissible construction histories.

Each value of

$$n$$

determines not only the observable shape of the resulting vector but also the historical constraints governing its construction.

Appending an element therefore extends both the vector and the history proving its length.

The index becomes a constructive witness rather than merely an integer carried inside a type.

8.9 The Four Modes of Dependence

The expressive power of the Calculus of Constructions is traditionally described by the four forms of dependence generated by the dependent product.

Within the historical calculus these acquire a unified interpretation.

Terms may depend upon terms because later constructions inherit earlier histories.

Types may depend upon terms because admissible future constructions depend upon previous values.

Terms may depend upon types because historical templates may abstract over earlier type constructions.

Finally, types may depend upon types because histories themselves may classify future histories of type formation.

All four dependency modes therefore become manifestations of a single principle: constructive continuation of admissible histories.

This observation suggests that the dependent product should not be viewed merely as a logical connective but as the universal constructor governing historical extension throughout the Spherepop kernel. Having established this foundation, we next develop the cumulative hierarchy of universes and show how historical construction prevents paradox while permitting unrestricted dependent abstraction across multiple logical levels.

9 Historical Universes

The introduction of dependent products immediately raises the question of how types themselves are to be classified. If types are ordinary objects of the language, then they too require types. Without a disciplined hierarchy the calculus quickly encounters the familiar logical paradoxes arising from self-reference. The historical interpretation adopted by Spherepop does not remove this necessity. Rather, it provides a different understanding of why the hierarchy exists.

Conventional presentations motivate universes as stratifications of collections of types. Spherepop instead interprets universes as successive levels of historical construction. A universe is not merely a container of types. It is a space within which particular kinds of historical constructions are admissible. Moving upward through the universe hierarchy therefore corresponds to enlarging the class of histories that may themselves become mathematical objects.

Consequently, universes should be understood operationally rather than extensionally. They classify modes of construction rather than collections of completed entities.

9.1 The Cumulative Hierarchy

The kernel adopts the standard cumulative hierarchy

$$\mathsf{Prop}, \qquad \mathsf{Type}_0, \qquad \mathsf{Type}_1, \qquad \mathsf{Type}_2, \qquad \cdots$$

together with the judgments

$$\mathsf{Prop} : \mathsf{Type}_0,$$

and

$$\mathsf{Type}_i : \mathsf{Type}_{i+1}.$$

Each universe is therefore itself an inhabitant of the next higher universe.

Unlike the inconsistent universe

$$\mathsf{Type} : \mathsf{Type},$$

no universe contains itself.

The historical semantics leave this observable hierarchy unchanged while providing a constructive interpretation of its significance.

9.2 Historical Interpretation

Suppose

$$H \vdash A : \mathsf{Type}_i.$$

This judgment states more than the ordinary assertion that

$$A$$

is a type.

It asserts that the historical construction responsible for

$$A$$

belongs to the class of admissible constructions represented by

$$\mathsf{Type}_i.$$

Thus universes classify histories of type construction.

Observable types appear only as projections of these richer historical objects.

From this perspective,

$$\mathsf{Type}_i$$

is itself a family of admissible construction procedures.

It describes not static collections but permissible methods of producing future types.

9.3 Historical Growth

One of the central principles established earlier is that histories are monotonic.

Universes inherit the same property.

Suppose

$$H \vdash A : \mathsf{Type}_i.$$

Extending the history cannot invalidate the construction of

$$A.$$

Consequently,

$$H; e \vdash A : \mathsf{Type}_i$$

whenever

$$e$$

is historically admissible.

This historical monotonicity mirrors the weakening theorem developed for contexts while extending it to the universe hierarchy itself.

Universes therefore remain stable under constructive growth.

9.4 Universe Lifting

Because the hierarchy is cumulative, every construction inhabiting a lower universe may be regarded as inhabiting every higher universe.

Formally,

$$\mathsf{Type}_i \hookrightarrow \mathsf{Type}_{i+1}.$$

Historically this inclusion should not be interpreted merely as set-theoretic containment.

Rather, every construction admitted at one historical level remains admissible after enlarging the available space of future constructions.

The inclusion therefore preserves provenance.

No reconstruction of the original object is required.

Only its admissible context of future use is expanded.

9.5 Universe Polymorphism

Many useful constructions should operate uniformly across arbitrary universe levels.

For example, the identity function naturally generalizes to

$$\Pi i.\, \Pi A : \mathsf{Type}_i.\, A \to A.$$

The historical interpretation is particularly natural.

The function abstracts not merely over observable types but over entire classes of historical type constructions.

Consequently, universe polymorphism becomes historical polymorphism.

The resulting proof object is reusable across every admissible level of the construction hierarchy without modification of its internal provenance.

9.6 Historical Closure

Dependent products preserve universe levels according to the familiar formation rule.

Suppose

$$H \vdash A : \mathsf{Type}_i$$

and

$$H, x : A \vdash B : \mathsf{Type}_j.$$

Then

$$H \vdash \Pi x : A.B : \mathsf{Type}_{\max(i,j)}.$$

Historically, this theorem states that composing two admissible construction procedures produces another admissible construction procedure whose complexity is governed by the larger of the two historical levels.

Thus the universe hierarchy is closed under dependent abstraction.

No additional universe constructors are required.

9.7 Historical Reflection

The cumulative hierarchy also supports a limited form of historical reflection.

A history may reason about constructions performed at lower universe levels without becoming identical to them.

Suppose

$$\mathsf{Type}_i : \mathsf{Type}_{i+1}.$$

A construction within

$$\mathsf{Type}_{i+1}$$

may therefore manipulate descriptions of constructions occurring in

$$\mathsf{Type}_i.$$

Reflection consequently becomes stratified.

Each level may observe lower levels while remaining insulated from direct self-reference.

This asymmetry provides the constructive explanation for the consistency of the hierarchy.

9.8 Historical Consistency

The prohibition against

$$\mathsf{Type} : \mathsf{Type}$$

acquires a particularly intuitive interpretation.

If a universe were permitted to classify its own historical construction, then the corresponding dependency graph would necessarily contain a directed cycle.

Such cycles contradict the foundational principle that histories are acyclic, irreversible constructions.

Thus the historical semantics provide an operational explanation for why self-containing universes are forbidden.

Consistency follows not merely from an abstract logical restriction but from the impossibility of constructing cyclic histories within the event algebra.

9.9 Universes as Histories of Histories

The interpretation developed in this section suggests that universes should not be regarded as levels of objects but as levels of historical organization.

Terms inhabit histories.

Types classify terms.

Universes classify histories of type construction.

Higher universes classify increasingly rich modes of historical organization.

The familiar cumulative hierarchy therefore acquires a dynamic interpretation in which every ascent through the hierarchy corresponds to reasoning about a more expressive class of constructive histories.

This viewpoint preserves the metatheoretic advantages of conventional universe systems while integrating them naturally into the event-oriented ontology of Spherepop.

Having established the hierarchy of universes, we next turn to one of the most subtle notions in the entire calculus: equality. Rather than treating equality as primitive or purely extensional, the historical kernel derives it from constructive provenance through explicit collapse events, yielding a notion of historical equivalence that serves as the foundation for definitional equality, identity types, and proof replay.

10 Historical Equality and Definitional Equivalence

Equality occupies a unique position within constructive mathematics. Every proof assistant requires a notion of equality in order to compare terms, simplify expressions, normalize computations, and verify that two derivations represent the same mathematical object. Despite its ubiquity, equality is often treated as primitive. One either assumes an extensional notion of identity or defines an inductive identity type whose computational interpretation is specified separately.

The historical foundation of Spherepop suggests a different approach.

Objects are not primitive.

Histories are primitive.

Consequently, equality should arise from relationships between histories rather than relationships between completed objects. Observable equality is therefore a projection of a richer historical equivalence.

This shift has several important consequences.

First, equal objects need not possess identical provenance.

Second, identical provenance always determines equal objects.

Third, explicit identification becomes a constructive event rather than an implicit logical assumption.

The resulting notion of equality is therefore simultaneously computational, constructive, and historical.

10.1 Observable Equality

Let

$$H \vdash a : A, \qquad H \vdash b : A.$$

Traditional dependent type theory asks whether

$$a \equiv b$$

holds under definitional equality.

Spherepop instead begins with the histories

$$\mathrm{History}(a)$$

and

$$\mathrm{History}(b).$$

Observable equality is regarded as a consequence of their constructive relationship rather than as an independent primitive relation.

10.2 Historical Equivalence

Definition 10.1 (Historical Equivalence)**.** Two constructions

$$a$$

and

$$b$$

are historically equivalent whenever there exists an admissible replay of their construction histories that establishes identical observable behavior.

We write

$$a \equiv_H b.$$

Notice that historical equivalence is weaker than historical identity.

Two constructions may arise from different derivations while nevertheless computing the same observable result.

Conversely, historical identity immediately implies historical equivalence.

Thus

$$a =_H b \Longrightarrow a \equiv_H b,$$

but the converse need not hold.

This distinction allows the kernel to preserve provenance without sacrificing the extensional reasoning required by ordinary mathematics.

10.3 Replay Equivalence

The event-oriented semantics naturally introduce a replay operator.

Suppose

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

reconstructs the observable computation determined by the history

$$H.$$

Two histories are replay equivalent whenever replay produces identical normal forms.

Definition 10.2 (Replay Equivalence)**.** Histories

$$H_1$$

and

$$H_2$$

are replay equivalent whenever

$$\text{Replay}(H_1) = \text{Replay}(H_2).$$

We write

$$H_1 \simeq_R H_2.$$

Replay equivalence provides the operational interpretation of definitional equality.

Instead of comparing symbolic expressions directly, the kernel compares their constructive histories.

10.4 Collapse Events

Historical equivalence alone does not identify two constructions.

Identification requires an explicit historical event.

Suppose

$$a$$

and

$$b$$

have been shown replay equivalent.

The kernel records

$$\mathsf{Collapse}(H, a, b),$$

producing the extended history

$$H'.$$

This event permanently records the constructive decision that the two histories should thereafter be regarded as interchangeable.

Unlike quotient constructions, no provenance is discarded.

Both original derivations remain permanently recoverable.

The collapse merely contributes another event to the historical record.

Thus equality itself possesses provenance.

10.5 Definitional Equality

Observable type checking ultimately requires a decidable notion of definitional equality.

Within the historical calculus this relation is derived rather than assumed.

Definition 10.3 (Historical Definitional Equality)**.** Two terms are definitionally equal whenever their normalized histories are replay equivalent,

$$a \equiv b \iff \text{Normalize}(\text{History}(a)) \simeq_R \text{Normalize}(\text{History}(b)).$$

Normalization therefore acts upon histories before observable terms are compared.

This differs conceptually from ordinary conversion checking while remaining observationally compatible with it.

10.6 Conversion Rule

The familiar conversion rule now takes historical form.

Theorem 10.4 (Historical Conversion)**.** *Suppose*

$$H \vdash t : A$$

and

$$A \equiv_H B.$$

Then

$$H \vdash t : B.$$

Proof. Replay equivalence establishes that the normalized histories constructing

$$A$$

and

$$B$$

produce identical observable type structures.

The typing derivation therefore remains valid after replacing one historical construction by an equivalent one.

□

Observe that the proof depends upon replay equivalence rather than upon syntactic coincidence.

Conversion is therefore justified operationally through constructive replay.

10.7 Congruence

Historical equivalence is preserved by every constructor of the calculus.

If

$$a_1 \equiv_H a_2,$$

then

$$f\, a_1 \equiv_H f\, a_2,$$

provided the application histories are themselves admissible.

Likewise,

$$\lambda x.\, t_1 \equiv_H \lambda x.\, t_2$$

whenever

$$t_1 \equiv_H t_2.$$

The dependency graph therefore propagates historical equivalence compositionally throughout every derivation.

10.8 Historical Quotients

Traditional quotient constructions identify objects by introducing an equivalence relation and then forgetting the distinction between equivalent representatives.

Spherepop performs the opposite construction.

Representatives are never forgotten.

Instead, every identification becomes another historical event.

Consequently, quotient structures become accumulated histories of collapse operations.

This approach preserves complete provenance while supporting the same observable mathematics.

A quotient object therefore consists not merely of an equivalence class but of the complete constructive history by which that equivalence class was established.

10.9 Equality as Constructive Provenance

The development of this section illustrates one of the central themes of the historical calculus.

Equality is not a primitive relation imposed upon completed mathematical objects.

It is a consequence of admissible historical reconstruction.

Objects become interchangeable because their histories admit equivalent replays, and they become identified only after an explicit collapse event has recorded that decision.

This interpretation preserves all of the computational behavior expected by the Calculus of Constructions while introducing an additional layer of provenance that remains available for proof replay, dependency analysis, optimization, and verified compilation.

The next section applies these ideas to the primitive historical operations introduced earlier, showing how Pop, Refuse, Bind, Collapse, and Meld become fully typed computational operators within the dependent type calculus itself.

11 Typing the Primitive Historical Operations

The event algebra introduced earlier has so far been presented primarily as an informal ontology. In order for the historical calculus to function as the trusted kernel of a proof assistant or programming language, these primitive operations must themselves become well-typed constructions. They cannot remain external implementation mechanisms. Instead, they must participate in the same typing discipline that governs functions, dependent products, inductive families, and proof terms.

This section therefore internalizes the historical operations into the type calculus. Every primitive event becomes a typed constructor whose execution extends an admissible history while preserving logical correctness.

The objective is not merely to provide operational semantics. Rather, the event algebra itself becomes part of the language.

11.1 Histories as First-Class Objects

Up to this point histories have appeared as parameters of typing judgments,

$$H \vdash t : A.$$

It is now useful to internalize histories as mathematical objects.

Definition 11.1 (History Type)**.** For every universe level i, there exists a type

$$\mathsf{History}_i : \mathsf{Type}_{i+1},$$

whose inhabitants are admissible constructive histories.

Consequently,

$$H : \mathsf{History}_i$$

is itself a well-typed term.

Histories therefore become available to ordinary dependent abstraction, polymorphism, and inductive reasoning.

This step is important because it allows programs to reason explicitly about their own provenance without requiring any extension of the underlying logical framework.

11.2 The Empty History

The distinguished empty history is represented by the constructor

$$\varepsilon : \mathsf{History}.$$

Operationally,

$$\varepsilon$$

contains no events and therefore possesses an empty dependency graph.

Every admissible history is generated by finite applications of the primitive historical constructors beginning from this distinguished object.

Accordingly,

$$\text{History} = \text{Closure}(\varepsilon).$$

The closure is taken under the primitive event algebra introduced previously.

11.3 Typing Historical Extension

Historical extension appends one admissible event to an existing history.

Its type is

$$\text{extend} : \Pi H : \mathsf{History}.\, \mathcal{E} \to \mathsf{History}.$$

Given

$$H : \mathsf{History}$$

and

$$e : \mathcal{E},$$

the application

$$\text{extend}(H, e)$$

constructs the new history

$$H; e.$$

Unlike ordinary list construction, extension carries semantic constraints.

The resulting history must satisfy the admissibility conditions developed in earlier sections.

Thus historical extension is only partially defined over arbitrary events, although it is total over admissible ones.

11.4 Typing Pop

The Pop operation records constructive commitment among admissible continuations.

Its dependent type is

$$\mathsf{Pop} : \Pi H : \mathsf{History}.\, \Pi b : \mathsf{Branch}(H).\, \mathsf{History}.$$

Here

$$\mathsf{Branch}(H)$$

denotes the family of admissible continuations available after the history

$$H.$$

The dependency upon

$$H$$

is essential.

Different histories generally admit different future continuations.

Consequently, Pop is naturally expressed as a dependent function.

11.5 Typing Refuse

Constructive refusal possesses an analogous type,

$$\mathsf{Refuse} : \Pi H : \mathsf{History}.\, \Pi a : \mathsf{Alternative}(H).\, \mathsf{History}.$$

The family

$$\mathsf{Alternative}(H)$$

enumerates those admissible constructions that may still be rejected.

Once refusal has been recorded, the rejected alternative no longer belongs to the admissible continuation space of the extended history.

Thus refusal modifies the future geometry of computation while preserving the past.

11.6 Typing Bind

Historical dependency becomes an explicit dependent constructor.

Suppose

$$x : A$$

and

$$f : A \to B.$$

Then

$$\mathsf{Bind} : \Pi H : \mathsf{History}.\, \Pi x : A.\, \Pi f : A \to B.\, \mathsf{History}.$$

Applying Bind extends the dependency graph by introducing an edge connecting the construction of

$$x$$

to every subsequent construction generated by

$$f.$$

Dependency therefore becomes an explicit mathematical object rather than an implicit consequence of substitution.

11.7 Typing Collapse

Historical identification is likewise internalized.

The constructor

$$\mathsf{Collapse}$$

has type

$$\Pi H : \mathsf{History}.\, \Pi a : A.\, \Pi b : A.\, \mathsf{ReplayEq}(a, b) \to \mathsf{History}.$$

Observe that collapse requires explicit evidence that the two constructions are replay equivalent.

The kernel therefore prevents arbitrary identifications.

Every collapse event carries a constructive witness demonstrating the equivalence that justifies the identification.

In this sense, equality becomes evidence-based rather than axiomatic.

11.8 Typing Meld

Concurrent composition combines two compatible histories.

The most general typing rule is

$$\mathsf{Meld} : \Pi H_1 : \mathsf{History}.\, \Pi H_2 : \mathsf{History}.\, \mathsf{Compatible}(H_1, H_2) \to \mathsf{History}.$$

Compatibility itself appears as an explicit proof object.

Thus concurrency is not assumed.

It is verified.

Only after compatibility has been established does the kernel permit the construction of the merged history.

This formulation naturally supports mechanically verified distributed computation.

11.9 Historical Closure

The primitive operations satisfy an important closure theorem.

Theorem 11.2 (Closure of the Event Algebra)**.** *The collection of well-formed histories is closed under every primitive historical constructor.*

Proof. Historical extension preserves admissibility by construction.

Pop and Refuse merely specialize admissible extensions.

Bind introduces only forward dependency edges and therefore cannot create cycles.

Collapse extends the history only after replay equivalence has been constructively established.

Meld requires an explicit proof of compatibility before construction.

Consequently, every primitive operation preserves well-formedness.

□

This theorem ensures that the event algebra remains internal to the trusted kernel.

No primitive operation is capable of producing an ill-formed history provided its typing premises have been satisfied.

The event algebra therefore enjoys the same logical discipline as the remainder of the dependent type calculus.

Rather than existing alongside the type theory, it becomes one of its ordinary typed subsystems.

Having incorporated the primitive historical operations into the typing discipline, we are now prepared to develop the operational semantics of the entire kernel. The next section formalizes evaluation as a transition between historical states, demonstrating that computation simultaneously transforms observable terms and extends constructive provenance.

12 Operational Semantics over Histories

The preceding chapters have established the static structure of the historical type calculus. Histories have been introduced as first-class mathematical objects, the primitive event algebra has been internalized into the type system, and the familiar constructors of dependent type theory have been reinterpreted as operations upon constructive provenance. The remaining task is to describe computation itself.

Operational semantics traditionally describe how programs execute. One specifies a reduction relation

$$t \longrightarrow u,$$

which transforms one expression into another until a normal form is obtained. The surrounding execution environment is generally treated as an implementation detail. Reduction is therefore regarded as a relation upon terms rather than upon histories.

Spherepop reverses this perspective.

Evaluation never occurs independently of historical construction.

Every computational step both transforms an observable expression and extends the history responsible for that transformation. Operational semantics therefore become relations between pairs consisting of a history and an observable term.

The primitive computational judgment is

$$(H, t) \longrightarrow (H', u),$$

where

$$H'$$

extends

$$H.$$

The history itself is therefore part of the semantic state.

12.1 Configurations

The fundamental semantic object is called a configuration.

Definition 12.1 (Configuration)**.** A configuration is an ordered pair

$$(H, t),$$

where

$$H : \mathsf{History}$$

is a well-formed constructive history and

$$t$$

is a well-typed observable term.

Configurations replace the ordinary notion of machine state.
Unlike mutable stores, however, histories are immutable.
A computational step therefore produces a new configuration rather than modifying an existing one.

12.2 Evaluation Relation

Evaluation is written

$$(H, t) \longrightarrow (H', u).$$

This judgment carries two simultaneous meanings.
The observable term

$$t$$

reduces to

$$u.$$

At the same time,

$$H$$

is extended to

$$H'.$$

Every reduction therefore possesses both computational content and constructive provenance.

The second component is not auxiliary bookkeeping.

It is part of the mathematical meaning of computation itself.

12.3 Historical Reflexivity

Every configuration evaluates to itself in zero computational steps.

Accordingly,

$$(H, t) \Longrightarrow^{0} (H, t).$$

This relation forms the base case for finite evaluation sequences.

The superscript records the number of reduction events contributing to the constructed history.

12.4 Historical Transitivity

Evaluation composes naturally.

Suppose

$$(H_0, t_0) \longrightarrow (H_1, t_1)$$

and

$$(H_1, t_1) \longrightarrow (H_2, t_2).$$

Then

$$(H_0, t_0) \Longrightarrow (H_2, t_2),$$

where

$$H_2$$

contains the complete provenance of both reductions.

Operational composition therefore coincides with historical composition.

No separate mechanism is required.

12.5 Beta Reduction

The principal computational rule remains beta reduction.

Observable computation is unchanged,

$$(\lambda x : A.\, t)\, a \longrightarrow t[x := a].$$

Historically, however, beta reduction generates a new event,

$$e_\beta.$$

The complete reduction rule becomes

$$(H, (\lambda x : A.\, t)\, a) \longrightarrow (H; e_\beta, t[H, x \mapsto a]).$$

The beta event records

1. the function being applied,
2. the argument supplied,
3. the substitution performed,
4. the dependency graph inherited by the resulting construction.

Thus every beta reduction becomes replayable.

12.6 Historical Congruence

Evaluation is preserved by every syntactic constructor.

Suppose

$$(H, t) \longrightarrow (H', u).$$

Then

$$(H, f\,t) \longrightarrow (H', f\,u),$$

provided the application remains well typed.
Similarly,

$$(H, \lambda x.t) \longrightarrow (H', \lambda x.u)$$

whenever evaluation occurs inside the body.

These congruence rules mirror the standard small-step semantics while extending their histories appropriately.

12.7 Historical Values

Values terminate observable computation.

Historically, however, they remain accompanied by their construction provenance.

A value therefore consists of the pair

$$(H, v),$$

where

$$v$$

is irreducible and

$$H$$

records every event required to construct it.
Evaluation terminates only when both components are complete.

12.8 Normalization

The normalization relation is written

$$(H, t) \Downarrow (H', v).$$

Unlike ordinary normalization, this relation computes two outputs.
The observable normal form

$$v,$$

and
the completed historical derivation

$$H'.$$

Consequently,

$$\text{Normalize}(t) = (v, H'),$$

rather than merely

$$v.$$

Proof replay therefore requires no additional instrumentation.
The evaluator itself constructs the complete provenance.

12.9 Determinism

The historical semantics preserve deterministic evaluation.

Theorem 12.2 (Historical Determinism)**.** *Suppose*

$$(H, t) \longrightarrow (H_1, u_1)$$

and

$$(H, t) \longrightarrow (H_2, u_2).$$

Then

$$u_1 = u_2$$

and

$$H_1 \simeq H_2.$$

Proof. Observable determinism follows from the deterministic reduction strategy of the underlying lambda calculus.

Every reduction rule introduces a uniquely determined historical event.

Consequently the resulting histories are replay equivalent.

□

This theorem ensures that historical provenance remains reproducible.

Running the same computation twice reconstructs the same derivation rather than merely the same observable result.

12.10 Replay Semantics

One immediate consequence of historical evaluation is the existence of a canonical replay operation.

Given a completed history

$$H,$$

the replay operator reconstructs every intermediate configuration,

$$(H_0, t_0),\ (H_1, t_1),\ \ldots,\ (H_n, t_n).$$

Proof verification therefore becomes historical verification.

Rather than trusting externally supplied proof objects, the kernel reconstructs their complete derivation from immutable historical events.

This capability provides an intrinsic foundation for proof certificates, incremental recompilation, reproducible builds, distributed verification, and persistent debugging.

12.11 Computation as Historical Growth

The operational semantics developed here reveal a unifying principle that has been implicit throughout the preceding chapters.

Computation is not the transformation of one state into another.

It is the irreversible growth of an admissible history.

Observable expressions merely provide snapshots of that ongoing construction.

The true semantic object is the history itself.

Reduction, normalization, and proof construction all become particular forms of historical extension governed by the event algebra.

This viewpoint allows operational semantics, provenance tracking, proof replay, and event sourcing to coexist within a single mathematical framework without introducing separate implementation layers.

Having established the semantics of evaluation, we now turn to the practical problem of type checking. The next chapter develops a bidirectional historical type checker whose conversion algorithm operates by replaying and comparing constructive histories rather than merely normalizing observable syntax.

13 A Bidirectional Historical Type Checker

The preceding chapters have developed the mathematical structure of the historical calculus. In order for this theory to serve as the kernel of a practical implementation, the typing rules must be organized into an algorithm. The trusted kernel should be small, deterministic, and mechanically implementable. At the same time, it should expose sufficient expressive power to support dependent types, proof construction, and higher-order polymorphism without requiring extensive proof search.

Modern proof assistants achieve this objective through bidirectional type checking. Instead of attempting to infer the type of every expression, bidirectional systems distinguish between two complementary judgments.

Certain expressions *synthesize* a type.

Other expressions are *checked* against a type already known from their context.

The historical calculus preserves this organization while extending every judgment with constructive provenance. Type checking therefore validates both observable typing and historical admissibility simultaneously.

13.1 Two Judgments

The historical kernel employs two algorithmic judgments.

The synthesis judgment

$$H \vdash t \Rightarrow A$$

computes the type of an expression.

The checking judgment

$$H \vdash t \Leftarrow A$$

verifies that an expression inhabits a previously supplied type.

These two judgments mutually define one another.

Their interaction replaces the unrestricted inference procedures that would otherwise render dependent type checking computationally impractical.

13.2 Variable Synthesis

Variables synthesize their declared type.

Suppose

$$\text{Declare}(x : A) \in H.$$

Then

$$\frac{\text{Declare}(x : A) \in H}{H \vdash x \Rightarrow A}.$$

Historically, the declaration event supplies both the observable type and the provenance associated with the variable.

No additional lookup structure is required.

The history itself functions as the typing environment.

13.3 Application Synthesis

Applications likewise synthesize their result type.

Suppose

$$H \vdash f \Rightarrow \Pi x : A.\, B$$

and

$$H \vdash a \Leftarrow A.$$

Then

$$\frac{H \vdash f \Rightarrow \Pi x : A.\, B \qquad H \vdash a \Leftarrow A}{H \vdash f\, a \Rightarrow B[H, x \mapsto a]}.$$

Observe that the codomain is instantiated using historical substitution rather than purely syntactic substitution.

The resulting type therefore inherits the provenance of the supplied argument.

13.4 Lambda Checking

Lambda abstractions generally do not synthesize useful types.

Instead, they are checked against an expected dependent product.

Suppose

$$H \vdash \Pi x : A.\, B : \mathsf{Type}_i.$$

Then

$$\frac{H; \mathsf{Declare}(x : A) \vdash t \Leftarrow B}{H \vdash \lambda x : A.\, t \Leftarrow \Pi x : A.\, B}.$$

The declaration of

$$x$$

extends the historical context before checking the body.

Thus abstraction simultaneously extends both the observable scope and the dependency graph.

13.5 Universe Synthesis

Universes synthesize the next higher universe.

Accordingly,

$$\frac{}{H \vdash \mathsf{Type}_i \Rightarrow \mathsf{Type}_{i+1}}.$$

This judgment is independent of the surrounding history.

Nevertheless, the construction of the universe itself remains part of the historical derivation.

13.6 Dependent Product Checking

Formation of dependent products proceeds recursively.

Suppose

$$H \vdash A \Leftarrow \mathsf{Type}_i$$

and

$$H; \text{Declare}(x : A) \vdash B \Leftarrow \mathsf{Type}_j.$$

Then

$$H \vdash \Pi x : A.\, B \Leftarrow \mathsf{Type}_{\max(i,j)}.$$

This rule mirrors the formation theorem developed earlier while presenting it algorithmically.

13.7 Historical Conversion Checking

Dependent type checking ultimately depends upon conversion.

Suppose

$$H \vdash t \Rightarrow A$$

and the expected type is

$$B.$$

Rather than comparing

$$A$$

and

$$B$$

syntactically, the historical checker computes

$$\text{Normalize}(\text{History}(A))$$

and

$$\text{Normalize}(\text{History}(B)).$$

The two histories are then replayed.

If

$$\text{Normalize}(\text{History}(A)) \simeq_R \text{Normalize}(\text{History}(B)),$$

the check succeeds.

Thus conversion is determined operationally through replay equivalence.

Observable syntax alone is never the ultimate criterion.

13.8 Weak-Head Historical Normalization

Complete normalization is rarely necessary during ordinary type checking.

Instead, the kernel computes weak-head historical normal forms.

Define

$$\mathrm{whnf}(H, t) = (H', u),$$

where

$$u$$

is weak-head normal and

$$H'$$

contains the historical reductions performed while exposing the head constructor.

Conversion checking therefore evaluates only as much of the historical derivation as necessary.

This strategy preserves the efficiency of conventional bidirectional type checkers while maintaining complete provenance.

13.9 Algorithmic Soundness

The algorithmic judgments coincide with the declarative typing rules.

Theorem 13.1 (Soundness of Historical Type Checking)**.** *If*

$$H \vdash t \Rightarrow A,$$

or

$$H \vdash t \Leftarrow A,$$

according to the bidirectional algorithm, then

$$H \vdash t : A$$

holds in the declarative historical calculus.

Proof. By structural induction on the algorithmic derivation.

Each synthesis and checking rule corresponds directly to one of the declarative typing rules developed in the previous chapters.

Historical substitution preserves provenance by the Historical Substitution Lemma.

Historical conversion follows from replay equivalence.

Consequently every algorithmic derivation determines a valid declarative proof.

□

13.10 Algorithmic Completeness

The converse property also holds.

Theorem 13.2 (Completeness of Historical Type Checking)**.** *Every declarative typing derivation may be transformed into an equivalent algorithmic derivation by inserting explicit type annotations where required.*

This theorem establishes that the bidirectional presentation loses no expressive power.

Its purpose is purely computational.

The trusted kernel therefore remains small enough for practical implementation while preserving the full mathematical strength of the historical dependent type calculus.

13.11 Historical Verification

The resulting checker performs two logically distinct tasks.

It verifies that every observable expression possesses the claimed type.

Simultaneously, it verifies that every historical dependency satisfies the admissibility conditions of the event algebra.

Thus type checking becomes historical verification.

The kernel certifies not merely that a proof is correct but that its construction history is itself mathematically valid.

This distinction becomes increasingly important for distributed verification, proof replay, reproducible compilation, and long-lived mathematical archives, where provenance is often as significant as the final theorem.

With the algorithmic kernel now established, the remaining chapters move beyond implementation details and return to the mathematical structure of the calculus. The next section develops what may be regarded as the historical analogue of Barendregt's lambda cube, showing that the four classical modes of dependency arise as projections of a richer space of historical dependence.

14 The Historical Cube

The expressive power of the Calculus of Constructions is traditionally described using Barendregt's lambda cube. The cube classifies higher-order type systems according to three independent forms of abstraction. One axis permits terms to depend upon terms, another permits terms to depend upon types, and the third permits types to depend upon types. The full Calculus of Constructions occupies the corner obtained by enabling all three dimensions simultaneously.

The historical interpretation developed throughout this paper suggests that the lambda cube is not the most primitive object. Each of its dimensions is itself a consequence of a more fundamental notion: dependence upon constructive history.

Accordingly, Spherepop replaces the lambda cube with what we shall call the *Historical Cube*. Rather than classifying systems according to their syntactic forms of abstraction, the Historical Cube classifies them according to the kinds of historical dependence they admit.

The familiar systems of higher-order type theory then arise as projections of this richer historical structure.

14.1 Historical Dependence

Every construction in the Spherepop kernel carries an associated history,

$$\mathrm{History}(t).$$

Consequently, any dependency between mathematical objects induces a dependency between their histories.

Instead of asking whether a term depends upon another term, one asks whether the construction history of the former extends the construction history of the latter.

Dependence therefore becomes a relation

$$\text{History}(a) \prec \text{History}(b),$$

meaning that the construction of

$$b$$

requires the prior construction of

$$a.$$

The dependency graph introduced in earlier chapters provides the concrete representation of this relation.

14.2 The Four Historical Modes

The ordinary Calculus of Constructions distinguishes four principal modes of dependency.

Each admits a natural historical interpretation.

The first concerns terms depending upon terms.

Conventionally this is represented by ordinary function application.

Historically it expresses the extension of one construction history by another.

The second concerns types depending upon terms.

Dependent families such as

$$\mathsf{Vector}(A, n)$$

illustrate this case.

Historically the type evolves because the construction history of the index becomes part of the construction history of the family itself.

The third concerns terms depending upon types.

Polymorphic functions provide the standard example.

Historically this corresponds to constructing reusable histories whose future extensions are parameterized by earlier histories of type formation.

Finally, types may depend upon types.

Higher-order type operators classify entire families of historical constructions.

The resulting dependency graph therefore contains histories whose vertices are themselves histories of type construction.

Although these four cases appear different syntactically, they all arise from the same underlying mechanism: extension of admissible constructive histories.

14.3 The Historical Axes

The Historical Cube therefore replaces the syntactic axes of the lambda cube by historical ones.

The first axis measures value dependence,

$$\text{History}(v_1) \longrightarrow \text{History}(v_2).$$

The second measures type dependence,

$$\text{History}(T_1) \longrightarrow \text{History}(T_2).$$

The third measures historical abstraction itself,

$$\text{History}(H_1) \longrightarrow \text{History}(H_2).$$

The final axis, absent from the classical lambda cube, measures explicit historical composition through the primitive event algebra.

This additional dimension distinguishes Spherepop from conventional dependent type theories.

Programs may abstract not only over values and types but also over histories themselves.

14.4 Histories Depending upon Histories

The additional dimension deserves particular attention.

Suppose

$$F : \mathsf{History} \to \mathsf{History}.$$

Such an object is not merely a program manipulating data.

It is a construction transforming one admissible provenance into another.

Examples include proof replay, history compression, dependency optimization, distributed synchronization, incremental recompilation, and verified migration of

proof archives.

These operations are difficult to express directly within conventional type theory because histories exist only as implementation artifacts.

Within Spherepop they become ordinary mathematical constructions.

14.5 Historical Projection

The relationship between the Historical Cube and the classical lambda cube is described by a projection operator

$$\pi_\lambda.$$

This projection erases every explicit historical dependency while preserving the observable typing derivation.

Thus

$$\pi_\lambda : \text{Historical Cube} \longrightarrow \text{Lambda Cube}.$$

Every system represented inside the classical cube therefore appears as the observable shadow of a richer historical system.

The lambda cube is recovered without modification.

What changes is the interpretation of its dependency relations.

14.6 Historical Symmetry

An important consequence of the historical interpretation is that every form of dependency becomes formally uniform.

Whether one abstracts over values, types, proofs, universes, or histories, the underlying operation is always the extension of constructive provenance.

The calculus therefore avoids introducing distinct semantic mechanisms for each form of abstraction.

Instead, all forms of dependence become instances of historical continuation.

This uniformity considerably simplifies the conceptual architecture of the kernel.

14.7 Historical Functoriality

The dependency graph generated by the Historical Cube behaves functorially.

Suppose

$$F : H_1 \to H_2$$

and

$$G : H_2 \to H_3.$$

Then

$$G \circ F : H_1 \to H_3$$

preserves every dependency established by the individual transformations.

Identity histories satisfy

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

Thus historical transformations naturally organize themselves into a category, foreshadowing the categorical semantics developed in a later chapter.

14.8 Historical Expressiveness

The additional expressive power of the Historical Cube does not arise from new logical connectives.

No new primitive abstraction operator has been introduced.

Instead, the increase in expressive power results from making provenance itself available as an object of mathematical reasoning.

Programs may inspect histories.

Proofs may quantify over histories.

Optimization algorithms may transform histories.

Distributed systems may verify histories.

Every such construction remains internal to the same dependent type theory.

14.9 The Historical Cube as a Universal Construction

The Historical Cube therefore provides a unifying interpretation of higher-order dependent type theory.

The traditional lambda cube distinguishes several independent forms of abstraction.

The historical calculus derives them all from a single principle: admissible extension of constructive history.

Consequently, the expressive hierarchy of the Calculus of Constructions becomes a projection of a more general historical geometry.

The remaining chapters exploit this perspective repeatedly. In particular, the Curry–Howard correspondence acquires a historical interpretation in which proofs become immutable event traces, propositions become admissible classes of histories, and logical reasoning itself becomes the constructive evolution of provenance through the event algebra.

15 The Historical Curry–Howard Correspondence

One of the most influential discoveries in twentieth-century logic is the Curry–Howard correspondence. The observation that propositions may be interpreted as types and proofs as programs transformed the foundations of constructive mathematics and provided the conceptual basis for modern proof assistants. Under this interpretation, logical inference becomes computation, proof normalization becomes program evaluation, and theorem proving becomes the construction of well-typed terms.

The historical calculus preserves this correspondence while extending its semantic interpretation.

Programs are not the primitive objects of computation.

Histories are.

Consequently, proofs are no longer interpreted merely as programs. They are understood as immutable constructive histories whose observable program terms constitute only one projection of a richer mathematical object.

The Curry–Howard correspondence therefore acquires an additional dimension.

$$\boxed{\text{Propositions} \iff \text{Types} \iff \text{Programs} \iff \text{Constructive Histories}}$$

Rather than replacing the traditional correspondence, Spherepop extends it.

Observable computation and constructive provenance become two inseparable aspects of the same mathematical object.

15.1 Propositions as Historical Classes

Suppose

$$P : \mathsf{Prop}.$$

Conventionally,

$$P$$

denotes the type of all proofs establishing the proposition.

Historically,

$$P$$

classifies not merely proof terms but entire families of admissible construction histories.

Thus a proposition determines a collection

$$\mathcal{H}(P) = \{H \mid H \vdash p : P\}.$$

The proposition therefore specifies which histories are capable of constructing its proofs.

Logical meaning is consequently expressed through admissible provenance rather than through isolated proof objects.

15.2 Proofs as Histories

Every proof possesses two complementary representations.

The observable proof term

$$p$$

and its construction history

$$\mathrm{History}(p).$$

The historical calculus regards the latter as fundamental.

The proof term merely summarizes the result of replaying the underlying construction.

Accordingly, a theorem should properly be represented by the triple

$$(P, p, \text{History}(p)).$$

The proposition specifies what has been proved.
The proof term specifies the observable computation.
The history records how that computation was constructed.
All three components are mathematically significant.

15.3 Implication

Implication remains the dependent function space.

Observable syntax is unchanged,

$$P \to Q.$$

Historically, however, implication classifies transformations between proof histories.

A proof of

$$P \to Q$$

constructs a reusable historical procedure that extends every admissible proof history of

$$P$$

into a corresponding proof history of

$$Q.$$

Logical implication therefore becomes historical continuation.

15.4 Universal Quantification

Universal quantification continues to be represented by dependent products.

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

The historical interpretation is particularly natural.
Given any admissible construction of

$$x,$$

the proof extends that construction into an admissible history establishing

$$P(x).$$

Universality therefore concerns every admissible continuation of history rather than every element of an abstract domain.

15.5 Existential Quantification

Existential propositions are represented by dependent sums.

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

A witness therefore consists of two inseparable components.
The constructed value

$$a : A,$$

and the proof history establishing

$$P(a).$$

Historically,

$$(a, p)$$

is accompanied by

$$\text{History}(a, p),$$

which records the complete provenance of both the witness and its proof.
Existential reasoning therefore preserves constructive evidence in its entirety.

15.6 Conjunction

Logical conjunction corresponds to dependent pairing.

$$P \wedge Q \equiv P \times Q.$$

Historically, the conjunction combines two independent proof histories into a single composite history.

If

$$H_P$$

constructs

$$P$$

and

$$H_Q$$

constructs

$$Q,$$

then the conjunction is represented by

$$\mathsf{Meld}(H_P, H_Q),$$

provided the histories are compatible.

Thus conjunction becomes the logical manifestation of historical composition.

15.7 Disjunction

Disjunction expresses constructive branching.

Suppose

$$P \vee Q.$$

The proof history records not only the resulting branch but also the explicit historical Pop event responsible for selecting it.

Thus

$$\mathsf{Pop}(H, P)$$

and

$$\mathsf{Pop}(H, Q)$$

remain historically distinct even if subsequent reasoning eventually collapses their observable conclusions.

Logical choice therefore becomes an explicit historical commitment rather than an invisible implementation detail.

15.8 Negation

Constructive negation is interpreted as implication into falsity,

$$\neg P \equiv P \to \bot.$$

Spherepop distinguishes this logical interpretation from historical refusal.

The proposition

$$\neg P$$

states that every proof of

$$P$$

would produce contradiction.

The operation

$$\mathsf{Refuse}(H, P)$$

records that a particular constructive branch has been intentionally abandoned.

Negation and refusal therefore become complementary rather than identical concepts.

The first is logical.

The second is historical.

15.9 Proof Replay

Perhaps the most significant consequence of the historical interpretation is that every proof becomes replayable.

Suppose

$$H \vdash p : P.$$

Rather than trusting the proof term alone, the kernel reconstructs

$$\mathrm{Replay}(\mathrm{History}(p)).$$

Replay regenerates every intermediate typing judgment, every substitution, every beta reduction, every collapse event, and every dependency introduced during construction.

Proof checking therefore becomes deterministic historical reconstruction.

The proof certificate consists not merely of the final term but of the complete constructive history.

15.10 Historical Proof Irrelevance

Traditional type theory often studies proof irrelevance, the principle that any two proofs of the same proposition should be regarded as interchangeable.

The historical calculus suggests a more nuanced viewpoint.

Two proofs may establish the same proposition while exhibiting distinct construction histories.

Observable mathematics may therefore treat them as equivalent,

yet historical reasoning continues to distinguish their provenance.

Proof irrelevance thus becomes a projection obtained by erasing historical information rather than a primitive property of the kernel itself.

15.11 The Extended Curry–Howard Principle

The development of this chapter suggests a natural reformulation of the classical correspondence.

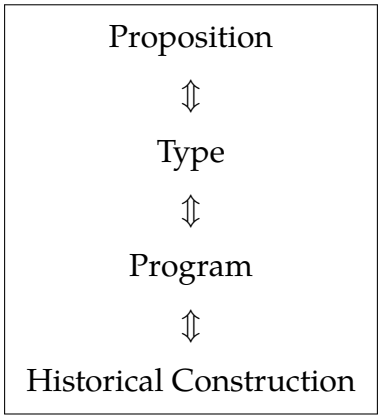
Rather than identifying propositions with types and proofs with programs, the historical calculus identifies propositions with admissible families of construction histories.

Proofs become immutable historical artifacts.

Programs become replayable constructions.

Logical inference becomes admissible historical extension.

The resulting correspondence may therefore be summarized as



This extended interpretation preserves the computational foundations of constructive logic while incorporating provenance directly into the trusted kernel. Logical reasoning no longer produces merely correct proofs; it produces permanent historical records whose construction may be replayed, verified, optimized, distributed, and analyzed long after the original derivation has been completed.

Having established the historical interpretation of logical reasoning, we next introduce inductive types and inductive families, showing how recursive construction itself may be represented as the controlled growth of admissible histories.

16 Inductive Types as Historical Generators

The preceding chapters have established the static and operational foundations of the historical calculus. Dependent products describe admissible historical continuations, universes classify levels of constructive organization, and the extended Curry–Howard correspondence interprets proofs as immutable historical artifacts. A practical language, however, requires more than functions and logical propositions. It must also support the construction of recursive data.

Modern proof assistants typically introduce inductive definitions as trusted extensions to the pure Calculus of Constructions. Although many inductive structures can be encoded using impredicative techniques, practical systems nearly always include native support for natural numbers, finite lists, trees, vectors, and other recursively generated families. The historical calculus follows the same strategy while providing a different semantic interpretation.

An inductive type is not viewed primarily as the least fixed point of a type operator.

It is viewed as the closure of a historical generation process.

The constructors of an inductive type are therefore not merely value constructors.

They are primitive generators of admissible histories.

16.1 Historical Generation

Suppose

$$G$$

is a finite collection of constructors.

Rather than defining an inductive type as the smallest set closed under these constructors, Spherepop defines it as the smallest family of histories closed under the corresponding construction events.

Definition 16.1 (Historical Generator)**.** A historical generator consists of a family of constructors

$$G = \{c_1, c_2, \ldots, c_n\},$$

together with admissibility rules describing how each constructor extends an existing history.

Every inhabitant of an inductive type therefore carries a finite generation history beginning with primitive constructors.

The inductive object and its provenance are inseparable.

16.2 Natural Numbers

The simplest example is the natural numbers.

Their observable declaration is familiar,

$$\mathsf{Nat} : \mathsf{Type}_0,$$

with constructors

$$\mathsf{Zero} : \mathsf{Nat},$$

and

$$\mathsf{Succ} : \mathsf{Nat} \to \mathsf{Nat}.$$

Historically, however, these constructors generate events.
The history of

$$\mathsf{Zero}$$

consists of a single construction event,

$$H_0 = \text{Construct}(\mathsf{Zero}).$$

Likewise,

$$\mathsf{Succ}(n)$$

extends the history of

$$n$$

by a successor event,

$$H_{\mathsf{Succ}(n)} = H_n; \text{Construct}(\mathsf{Succ}).$$

Thus

$$\mathsf{Succ}(\mathsf{Succ}(\mathsf{Zero}))$$

is represented not merely as a symbolic expression but as the historical chain

$$H_0; e_{\mathsf{Succ}}; e_{\mathsf{Succ}}.$$

The observable numeral is therefore the visible projection of an accumulated construction history.

16.3 Historical Induction

Induction follows directly from historical generation.
Suppose

$$P : \mathsf{Nat} \to \mathsf{Type}.$$

To establish

$$\Pi n : \mathsf{Nat}.P(n),$$

it is sufficient to construct

$$P(\mathsf{Zero})$$

and demonstrate that every successor history preserves the property.
The induction principle therefore becomes

$$\mathsf{indNat} : \Pi P : \mathsf{Nat} \to \mathsf{Type}.\ P(\mathsf{Zero}) \to (\Pi n : \mathsf{Nat}.\ P(n) \to P(\mathsf{Succ}(n))) \to \Pi n : \mathsf{Nat}.\ P(n).$$

Historically this theorem states that every admissible construction history generated by repeated applications of

$$\mathsf{Succ}$$

inherits the desired property.
Induction therefore becomes induction over constructive histories rather than over abstract elements.

16.4 Lists

Lists illustrate the same principle.
Observable constructors are

$$\mathsf{Nil} : \mathsf{List}(A),$$

and

$$\mathsf{Cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A).$$

The history of every list records every insertion event.
Accordingly,

$$[a_1, a_2, a_3]$$

possesses the historical structure

$$H_{\mathsf{Nil}}; e_{a_3}; e_{a_2}; e_{a_1},$$

where each constructor records the introduction of another element.

Deletion is not interpreted as removing an event.

Instead,

the history is extended by an explicit refusal event indicating that a particular element should no longer participate in future constructions.

The original insertion remains permanently recoverable.

16.5 Historical Trees

Recursive trees behave similarly.

Every node contributes a construction event together with dependency edges connecting it to its children.

The resulting history therefore forms a tree of events embedded within the global dependency graph.

Observable tree traversals become replay operations over this historical structure.

Algorithms such as balancing, pruning, and restructuring become transformations of histories rather than destructive modifications of mutable nodes.

16.6 Indexed Families

Dependent type theory derives much of its expressive power from indexed inductive families.

Consider vectors,

$$\mathsf{Vector}(A, n).$$

Each constructor contributes simultaneously to two histories.

The first constructs the observable vector.

The second constructs the witness establishing its length.

Appending an element therefore extends both histories together.

Consequently, the index

$$n$$

is not merely a static annotation.

It is itself a replayable historical artifact certifying the construction of the observable object.

16.7 Historical Initiality

Inductive types are traditionally characterized by initial algebra semantics.

The historical interpretation preserves this viewpoint while enriching its meaning.

The initial object is no longer merely the smallest fixed point of a functor.

It is the smallest admissible history closed under the generating constructors.

Recursive functions therefore preserve not only observable values but also constructive provenance.

This historical interpretation naturally extends to generalized algebraic data types, indexed inductive families, and higher inductive constructions.

16.8 Historical Canonical Forms

Every closed inhabitant of an inductive type reduces to a canonical constructor.

Historically, the canonical form theorem becomes stronger.

Not only does every closed inhabitant normalize to a constructor, but its complete construction history normalizes to a canonical generation history.

Thus canonical forms possess unique provenance.

Different derivations may later become replay equivalent through explicit collapse, but their original construction histories remain available throughout the lifetime of the system.

16.9 Induction as Historical Closure

The interpretation developed in this section illustrates another recurring theme of the historical calculus.

Recursive data are not fundamentally recursive values.

They are recursively generated histories.

Constructors generate new admissible events.

Eliminators replay those histories.

Induction proves properties preserved under historical generation.

The observable inductive object is therefore a compressed representation of an underlying constructive process.

This viewpoint unifies recursive programming, inductive reasoning, provenance, and replay within a single mathematical framework.

17 Computational Advantages over Conventional Proof Kernels

The historical calculus has been developed independently of any particular proof assistant. Nevertheless, it is instructive to compare its computational architecture with that of conventional dependent type kernels such as Lean.

The comparison is not intended to suggest that existing systems are incorrect or inefficient. Rather, the historical interpretation exposes several algorithmic structures that become available once constructive provenance is treated as a first-class mathematical object.

17.1 Incremental Type Checking

Traditional proof kernels repeatedly normalize large expressions while checking later declarations.

If a theorem near the beginning of a development changes, subsequent proofs may require extensive recomputation because normalization is performed over terms rather than historical dependencies.

Spherepop instead maintains an explicit dependency graph.

Every declaration possesses a unique construction history.

When a construction changes, only those histories reachable from the modified node require replay.

Type checking therefore becomes proportional to the affected historical region,

$$O(|\,\mathrm{Reach}(H)|),$$

rather than the size of the entire proof development.

The kernel therefore naturally supports incremental verification.

17.2 Proof Replay Instead of Renormalization

Conventional conversion checking repeatedly computes normal forms,

$$A \Downarrow A', \qquad B \Downarrow B'.$$

Spherepop instead compares replayable histories,

$$\text{Replay}(H_A) \quad \text{and} \quad \text{Replay}(H_B).$$

If these histories have already been normalized, replay requires reconstructing only the affected historical events.

Previously verified subhistories need not be recomputed.

Large mathematical libraries therefore admit persistent verification caches indexed by construction history rather than by normalized syntax.

17.3 Dependency-Driven Evaluation

Because every construction explicitly records its dependency graph,

$$D(H),$$

evaluation need never inspect unrelated subterms.

Independent branches may be skipped entirely.

Conventional normalization frequently traverses expressions whose values are already known.

Historical evaluation follows only reachable dependencies.

This converts normalization from a syntactic traversal into graph evaluation.

17.4 Parallel Verification

Independent histories satisfy

$$H_1 \perp H_2.$$

Such histories may be replayed concurrently,

$$\text{Replay}(H_1) \parallel \text{Replay}(H_2),$$

before being combined through

$$\mathsf{Meld}(H_1, H_2).$$

The dependency graph therefore exposes parallelism directly.

No additional proof scheduling algorithm is required.

Parallel verification becomes a consequence of the historical semantics.

17.5 Persistent Provenance

Traditional proof objects contain computational content but generally discard the intermediate construction process once normalization has completed.

Spherepop never loses this information.

Every theorem remains accompanied by its complete historical derivation.

Consequently,

debugging,

proof explanation,

incremental recompilation,

proof compression,

and dependency visualization

operate directly upon kernel objects rather than requiring external tooling.

17.6 Historical Compression

Repeated constructions frequently generate identical historical subgraphs.

Rather than storing duplicate proof trees, the kernel records shared historical prefixes.

Suppose

$$H = H_0; e_1; e_2; e_3,$$

and

$$H' = H_0; e_1; e_2; e_4.$$

The common prefix

$$H_0; e_1; e_2$$

is stored once.

Only the divergent suffixes require additional representation.

Proof libraries therefore become directed acyclic graphs of historical construction rather than independent proof trees.

This substantially reduces duplication in large formal developments.

17.7 Interpretability

Perhaps the greatest advantage of the historical calculus is interpretability.

Every proof object carries an explicit construction history.

Every optimization records the events that justified it.

Every equality possesses a corresponding collapse event.

Every dependency is explicitly represented.

Consequently, explanation requires no separate reconstruction algorithm.

The proof already contains its own explanation.

One may replay any theorem, inspect every reduction, recover every dependency, and reconstruct every intermediate derivation directly from the trusted kernel.

Interpretability is therefore a mathematical property of the calculus itself rather than a visualization layer added after computation.

The next chapter extends these ideas further by developing equality types, showing that identity proofs themselves may be treated as historical constructions whose computational content is determined by replay and explicit collapse rather than by extensional axioms.

18 Historical Equality Types

The previous chapter established that definitional equality in the historical calculus arises from replay equivalence and explicit collapse rather than from primitive extensional identity. This observation naturally raises a deeper question.

How should equality itself be represented inside the type theory?

In conventional dependent type theory this role is played by the identity type. Given two inhabitants

$$a, b : A,$$

one forms the proposition

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

whose inhabitants are proofs that the two objects are equal.

The historical calculus retains this construction while fundamentally changing its interpretation.

Identity proofs are no longer viewed as abstract witnesses relating completed objects.

They become constructive histories demonstrating how one construction may be replayed, transformed, and ultimately collapsed into another.

Equality therefore becomes an object of computation rather than merely a logical predicate.

18.1 Formation of Historical Equality

Suppose

$$H \vdash a : A, \qquad H \vdash b : A.$$

The historical equality type is formed by

$$H \vdash \mathrm{Eq}_A(a, b) : \mathsf{Prop}.$$

Observable syntax is therefore identical to ordinary dependent type theory.

Its semantics, however, differ substantially.

An inhabitant of

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

is interpreted as a constructive history demonstrating the replay equivalence of

$$a$$

and

$$b.$$

The equality proof therefore contains computational provenance rather than merely asserting logical coincidence.

18.2 Historical Reflexivity

Every construction is historically equivalent to itself.

Accordingly, reflexivity is represented by

$$\text{refl}_a : \text{Eq}_A(a, a).$$

The corresponding proof history is simply

$$\text{History}(a).$$

No additional computation is required.

Replay immediately reconstructs the original construction without modification.

Thus reflexivity represents the identity replay of a construction history.

18.3 Historical Symmetry

Suppose

$$p : \text{Eq}_A(a, b).$$

Since replay equivalence is symmetric,

$$\text{Replay}(\text{History}(a)) = \text{Replay}(\text{History}(b))$$

implies

$$\text{Replay}(\text{History}(b)) = \text{Replay}(\text{History}(a)).$$

Therefore one obtains

$$\text{sym}(p) : \text{Eq}_A(b, a).$$

Historically, symmetry corresponds to reversing the interpretation of an established replay relation.

No new construction history is created.

Only its direction of interpretation changes.

18.4 Historical Transitivity

Suppose

$$p : \text{Eq}_A(a, b)$$

and

$$q : \mathrm{Eq}_A(b, c).$$

Replay equivalence composes naturally.
Consequently,

$$\mathrm{trans}(p, q) : \mathrm{Eq}_A(a, c).$$

The resulting proof history is obtained by concatenating the replay histories associated with

$$p$$

and

$$q.$$

Identity proofs therefore compose exactly as historical derivations compose.

18.5 Collapse as Equality Realization

The previous chapter introduced collapse events as explicit historical identifications.

Equality types provide their logical interpretation.
Suppose

$$p : \mathrm{Eq}_A(a, b).$$

The kernel may then construct

$$\mathsf{Collapse}(H, a, b, p),$$

thereby extending the history with an explicit identification event justified by the equality proof itself.

Unlike quotient constructions, this operation does not erase either construction.
Instead,

$$\mathrm{History}(a), \qquad \mathrm{History}(b),$$

and

$$\text{History}(p)$$

remain permanently available.

The collapse records not that the histories were always identical, but that a constructive proof has established their equivalence.

Equality therefore possesses explicit provenance.

18.6 Replay Interpretation

The computational meaning of equality is determined by replay.

Suppose

$$p : \text{Eq}_A(a, b).$$

Executing

$$\text{Replay}(\text{History}(p))$$

reconstructs the sequence of historical transformations establishing the equivalence between

$$a$$

and

$$b.$$

Proof checking therefore becomes deterministic reconstruction rather than pattern matching upon symbolic expressions.

The equality proof is simultaneously
a logical witness,
a computational procedure,
and
a historical explanation.

18.7 Historical Transport

Dependent type theory permits transportation of constructions across equality proofs.

Suppose

$$P : A \to \mathsf{Type},$$

and

$$p : \mathrm{Eq}_A(a, b).$$

Then transport yields

$$\mathrm{transport} : P(a) \to P(b).$$

Historically, transport extends the replay represented by

$$p$$

to every dependent construction of

$$P.$$

The transported object inherits both the observable computation and the historical provenance associated with the equality proof.

Transport therefore becomes replay-guided reconstruction.

18.8 Equality as Historical Morphism

Equality proofs naturally organize themselves into morphisms between construction histories.

Given

$$a$$

and

$$b,$$

an inhabitant of

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

is interpreted as a morphism

$$\text{History}(a) \longrightarrow \text{History}(b),$$

preserving replay equivalence.

Composition of equality proofs corresponds exactly to composition of these historical morphisms.

Identity proofs therefore inherit an intrinsic categorical structure.

This observation will become central when categorical semantics are introduced in a later chapter.

18.9 Proof Irrelevance Revisited

Traditional proof irrelevance asserts that any two proofs of the same equality may safely be identified.

The historical calculus adopts a more refined interpretation.

Two equality proofs may establish precisely the same observable equivalence while arising from entirely different historical derivations.

Observable reasoning may therefore identify them,

yet provenance continues to distinguish them.

Proof irrelevance consequently appears only after applying the history-erasure functor introduced earlier.

Within the trusted kernel, every equality proof remains a distinct historical artifact.

18.10 Identity Through Construction

The historical interpretation of equality completes the transition from extensional to constructive identity.

Objects are equal because replay establishes historical equivalence.

They become interchangeable because an explicit collapse event records that equivalence.

Identity proofs themselves are replayable constructions whose computational content consists of reconstructing the historical transformations relating two objects.

Equality therefore ceases to be a primitive logical relation.

It becomes another form of constructive computation governed by the same event algebra that underlies the remainder of the Spherepop kernel.

The next chapter builds upon this foundation by developing event-sourced proof objects. Rather than treating completed proofs as static lambda terms, we shall show that every theorem naturally decomposes into its proposition, its proof term, and its immutable historical derivation, yielding a foundation for proof replay, distributed verification, and persistent mathematical archives.

19 Event-Sourced Proof Objects

The historical interpretation of equality developed in the previous chapter completes the transition from extensional identity to constructive provenance. Equality proofs are themselves historical constructions, and replay determines their computational meaning. We now apply this perspective to the structure of proof objects themselves.

Traditional proof assistants represent a theorem by a well-typed lambda term. Once elaboration, normalization, and type checking have completed, the proof term alone is retained as the trusted artifact. The intermediate derivation, the sequence of reductions, and the provenance of every construction are typically regarded as implementation details rather than mathematical objects.

Spherepop adopts the opposite philosophy.

The history is the primary object.

The proof term is its observable projection.

Consequently, a theorem is not merely an inhabitant of a proposition but an immutable historical artifact recording every admissible event that produced the proof.

This interpretation transforms proof objects from static terms into replayable constructive processes.

19.1 The Structure of a Theorem

A theorem consists of three inseparable components.

The first is the proposition,

$$P : \mathsf{Prop}.$$

The second is the observable proof term,

$$p : P.$$

The third is the historical derivation,

$$\text{History}(p).$$

Accordingly, a theorem is represented internally as

$$\Theta = (P, p, \text{History}(p)).$$

The observable proof is therefore only one component of a richer mathematical object.

Removing the history does not invalidate the theorem, but it removes the constructive explanation that produced it.

19.2 Historical Completeness

The construction history of a theorem records every primitive event performed by the kernel.

These include

$$\mathsf{Declare}, \quad \mathsf{Bind}, \quad \mathsf{Pop}, \quad \mathsf{Refuse}, \quad \mathsf{Collapse},$$

together with every beta reduction, every substitution, every universe formation, every constructor application, and every conversion accepted by the type checker.

Thus the history contains sufficient information to reconstruct the proof from the empty history alone.

No external proof script is required.

19.3 Proof Replay

The replay operator introduced earlier now acquires its principal application.

Given

$$\Theta = (P, p, \text{History}(p)),$$

the kernel computes

$$\text{Replay}(\text{History}(p)).$$

Replay reconstructs the complete derivation

$$\varepsilon \vdash \cdots \vdash p : P.$$

Every intermediate judgment is reproduced.

Every reduction is repeated.

Every dependency is re-established.

If replay terminates successfully, logical correctness follows immediately.

Proof checking therefore becomes deterministic historical reconstruction.

19.4 Proof Certificates

Because every theorem possesses a complete construction history, the historical record itself functions as a proof certificate.

Instead of transmitting a large normalized proof term, one may transmit the event sequence

$$e_1; e_2; \cdots ; e_n.$$

The receiving kernel simply replays the events.

If replay succeeds, the theorem is accepted.

The certificate therefore consists of executable mathematical provenance rather than an opaque symbolic object.

This representation naturally supports distributed verification.

Independent systems need only agree upon the event algebra and replay semantics.

19.5 Incremental Verification

One immediate consequence of event-sourced proofs is efficient incremental verification.

Suppose a theorem depends upon the historical prefix

$$H_0,$$

followed by the suffix

$$H_1.$$

If only

$$H_1$$

changes, replay begins from the cached result of

$$H_0.$$

The unchanged prefix is never recomputed.

Verification therefore scales with the modified portion of the dependency graph rather than with the size of the complete proof library.

This follows directly from the monotonic structure of histories established in earlier chapters.

19.6 Shared Historical Prefixes

Large mathematical libraries frequently contain thousands of proofs sharing common lemmas.

Traditional proof objects duplicate much of this computational structure.

The historical kernel instead stores common prefixes only once.

Suppose two proofs possess histories

$$H_A = H_0; H'_A,$$

and

$$H_B = H_0; H'_B.$$

The shared prefix

$$H_0$$

is represented once.

Only the divergent suffixes require additional storage.

The resulting proof archive naturally forms a directed acyclic graph of historical constructions rather than a forest of independent proof trees.

19.7 Historical Compression

Because histories are immutable, repeated replay identifies common subgraphs.

Compression therefore preserves complete provenance.

No mathematical information is discarded.

Instead, multiple proofs reference identical historical segments whenever their derivations coincide.

Compression is therefore structural rather than syntactic.

The efficiency of the archive depends upon shared constructive history rather than textual similarity.

19.8 Proof Explanation

One of the most significant consequences of event-sourced proofs is that explanation becomes intrinsic.

Suppose a user asks why a theorem holds.

Traditional systems often reconstruct an explanation by analyzing the completed proof term after verification has finished.

Spherepop requires no such reconstruction.

The explanation is already contained within the historical derivation.

Replay reveals

the declarations,

the substitutions,

the reductions,

the collapse events,

the dependency graph,

and every intermediate typing judgment.

The proof therefore carries its own explanation.

Interpretability is a property of the kernel rather than an auxiliary tool.

19.9 Historical Trust

The trusted computing base of the kernel remains intentionally small.

Only three components require complete trust.

The event algebra,

the replay engine,

and

the historical type checker.

Every theorem accepted by the system can be reconstructed solely from these components.

High-level syntax,

automation,

proof search,

optimization,

and elaboration may all be regarded as untrusted front-end tools.

Their output is accepted only after historical replay reconstructs the complete derivation inside the kernel.

This separation considerably simplifies the trusted foundation of the language.

19.10 Persistent Mathematics

The event-sourced interpretation suggests a broader philosophical consequence.

Mathematics is often regarded as a collection of completed theorems.

The historical calculus instead views mathematics as a continuously growing history of admissible constructions.

Every theorem contributes another immutable event to this historical record.

Nothing is forgotten.

Nothing is overwritten.

Later discoveries extend earlier ones without erasing their provenance.

The mathematical corpus therefore evolves as an event-sourced knowledge graph whose logical correctness is preserved by replay and whose historical structure remains permanently available for inspection.

This interpretation naturally unifies theorem proving, version control, distributed verification, reproducible computation, and mathematical archives within a single constructive framework. The following chapter extends this perspective further by developing the formal semantics of the `Meld` operation, showing how independently verified histories may be combined into a single coherent derivation while preserving both logical soundness and complete provenance.

20 Meld and the Geometry of Concurrent Construction

The event algebra introduced near the beginning of this paper contains one operation that has no direct analogue in the conventional Calculus of Constructions. The operations Pop, Refuse, Bind, and Collapse may all be viewed as reinterpretations of familiar computational ideas. Meld is fundamentally different.

Meld is not simply another constructor.

It provides a mathematical foundation for composing independently developed histories while preserving their individual provenance.

Traditional proof assistants generally assume a single linear derivation. Even when proofs are developed concurrently by multiple users or multiple processes, the final library is assembled through external version-control systems whose conflict resolution lies outside the logical kernel.

Spherepop instead incorporates concurrent historical composition directly into the mathematical foundations of the calculus.

Parallel construction therefore becomes an internal logical operation rather than an external engineering technique.

20.1 Independent Histories

Suppose

$$H_1$$

and

$$H_2$$

are well-formed histories.
Each may contain
declarations,
proofs,
definitions,
inductive constructions,
and
derived theorems.

The two histories may have been developed independently and need not share a common chronological origin beyond the empty history.

The problem addressed by Meld is the following.

Under what conditions may these histories be combined into a single admissible construction?

20.2 Compatibility

Not every pair of histories admits a coherent merge.

Conflicting declarations,
contradictory collapse events,
or cyclic dependencies may render a merge inadmissible.
Compatibility is therefore itself a mathematical proposition.

Definition 20.1 (Historical Compatibility)**.** Two histories

$$H_1$$

and

$$H_2$$

are compatible whenever

1. their dependency graphs remain acyclic after union,
2. their declarations do not introduce contradictory identities,
3. their collapse events preserve replay equivalence,
4. their universe assignments remain consistent,
5. their shared historical prefixes agree up to replay equivalence.

We write

$$H_1 \parallel H_2.$$

Compatibility therefore becomes a proof obligation.
It is not assumed.
It is established constructively.

20.3 The Meld Constructor

The primitive historical operation

$$\mathsf{Meld}$$

is now assigned its complete operational meaning.

Definition 20.2 (Historical Meld)**.** Suppose

$$H_1 \parallel H_2.$$

Then

$$\mathsf{Meld}(H_1, H_2) = H_3,$$

where

$$H_3$$

is the smallest admissible history containing both

$$H_1$$

and

$$H_2$$

together with the dependency edges required to preserve their provenance.

Observe that Meld does not interleave events arbitrarily.

Rather, it computes the minimal historical extension preserving every dependency already established by the component histories.

20.4 Historical Pushouts

The Meld operation resembles the categorical pushout of two morphisms sharing a common domain.

Suppose

$$H_0$$

is a common historical prefix.

Then

$$H_1 = H_0; H_A,$$

and

$$H_2 = H_0; H_B.$$

Meld constructs

$$H_3 = H_0; H_A; H_B,$$

modulo dependency constraints and replay equivalence.

Unlike ordinary pushouts, however, the resulting history remains computational.

It is an executable construction rather than merely a universal object.

This distinction illustrates the constructive character of the historical calculus.

20.5 Preservation of Provenance

Perhaps the most important property of Meld is that provenance is never lost.

Suppose

$$e \in H_1.$$

Then

$$e \in \mathsf{Meld}(H_1, H_2).$$

Likewise,
every event belonging to

$$H_2$$

remains present after the merge.

Meld therefore performs accumulation rather than replacement.

No information is discarded.

This contrasts sharply with conventional merge algorithms, which frequently eliminate or overwrite conflicting intermediate states.

20.6 Dependency Preservation

The dependency graph satisfies a similar preservation theorem.

Theorem 20.3 (Dependency Preservation under Meld)**.** *Suppose*

$$H_1 \parallel H_2.$$

Then

$$D(\mathsf{Meld}(H_1, H_2)) = D(H_1) \cup D(H_2),$$

together with the minimal compatibility edges required by the merged history.

Proof. Every dependency contained in

$$H_1$$

or

$$H_2$$

must remain valid after merging.

Compatibility excludes the introduction of cycles.

Consequently, the dependency graph is precisely the union of the original graphs together with the additional edges required to preserve admissibility.

□

This theorem establishes that Meld preserves every constructive explanation already present in the component histories.

20.7 Parallel Proof Construction

The historical semantics now provide a natural model of parallel theorem proving.

Suppose two independent teams establish

$$P$$

and

$$Q.$$

Their histories

$$H_P$$

and

$$H_Q$$

may be developed concurrently.

Once compatibility has been established,

$$\mathsf{Meld}(H_P, H_Q)$$

produces a single history containing both proofs.

No replay of the completed proofs is required.

Only the compatibility proof must be verified.

Consequently, concurrency becomes mathematically compositional.

20.8 Distributed Verification

The same construction extends naturally to distributed proof systems.

Independent nodes maintain local histories,

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

Periodically,

pairs of compatible histories are melded,

eventually producing a globally verified construction history.

Unlike conventional distributed databases, no mutable global state exists.

Every node contributes immutable historical events whose provenance remains permanently available after merging.

Consistency therefore follows from replay and compatibility rather than from locking or consensus over mutable objects.

20.9 Associativity

Historical composition satisfies an associativity property.

Theorem 20.4 (Associativity of Meld)**.** *Suppose*

$$H_1, H_2, H_3$$

are pairwise compatible.

Then

$$\mathsf{Meld}(\mathsf{Meld}(H_1, H_2), H_3) \simeq \mathsf{Meld}(H_1, \mathsf{Meld}(H_2, H_3)).$$

Replay equivalence rather than syntactic equality appears because the internal ordering of independent events may differ while preserving identical constructive meaning.

Associativity therefore holds up to historical equivalence.

20.10 Historical Concurrency

The introduction of Meld reveals that concurrency is not an operational concern added after the logical kernel has been designed.

It is a primitive mathematical notion.

Independent histories evolve simultaneously.

Compatibility determines when they may be composed.

Replay verifies the resulting construction.

Provenance is preserved throughout.

The logical calculus therefore acquires a native theory of concurrent construction rather than relying upon external synchronization mechanisms.

This interpretation has significant consequences for proof assistants, distributed theorem proving, incremental verification, collaborative formal mathematics, and event-sourced computation.

In every case, concurrency is represented by the same historical algebra that governs individual computation.

The next chapter complements Meld by returning to the dual operation of Collapse. Whereas Meld composes independent histories, Collapse identifies historically distinct constructions that have been shown equivalent. Together, these two operations provide the fundamental mechanisms by which the historical calculus expands and reorganizes its growing body of constructive knowledge.

21 Collapse as Constructive Quotient Formation

The preceding chapter introduced Meld as the primitive operation by which independent constructive histories are composed into a single coherent derivation. Meld enlarges the historical universe by preserving distinct provenance while combining compatible developments. The complementary operation is Collapse.

Where Meld increases the breadth of constructive history, Collapse reduces its redundancy.

This reduction should not be confused with deletion.

Nothing is erased.

Nothing is forgotten.

Instead, Collapse records the mathematical discovery that two independently constructed histories represent the same observable object. The distinction is essential. Classical quotient constructions identify equivalent objects by discarding differences between representatives. Spherepop instead preserves every representative together with the explicit historical event that justifies their identification.

Consequently, quotient formation becomes an irreversible constructive event rather than an extensional operation performed upon completed sets.

21.1 The Limitations of Extensional Quotients

Suppose one wishes to identify two constructions,

$$a, b : A,$$

under an equivalence relation

$$a \sim b.$$

Traditional mathematics forms the quotient

$$A/\!\sim,$$

whose elements are equivalence classes.

Although mathematically elegant, this construction loses significant information.

The individual derivations of

$$a$$

and

$$b$$

disappear.

Only the resulting equivalence class remains.

From the perspective of constructive provenance, this is undesirable.

Proof assistants increasingly rely upon explanation, replay, incremental verification, and dependency analysis.

All of these require access to the original derivations.

An extensional quotient destroys precisely the information required to support these operations.

21.2 Historical Collapse

The historical calculus replaces quotient formation with an explicit constructive event.

Suppose

$$p : \mathrm{Eq}_A(a, b)$$

has been established.

The kernel constructs

$$\mathsf{Collapse}(H, a, b, p) = H'.$$

The extended history

$$H'$$

contains

$$H,$$

the construction histories of

$$a, \qquad b,$$

the equality proof

$$p,$$

and the collapse event itself.

Nothing is removed.

Instead, the mathematical knowledge that the two constructions are interchangeable becomes part of the permanent historical record.

Collapse therefore records the discovery of equivalence rather than replacing existing constructions.

21.3 Collapse Graphs

Successive collapse operations generate a graph of historical identifications.

Suppose

$$a \equiv b, \qquad b \equiv c.$$

The resulting history contains two explicit collapse events,

$$e_{ab}, \qquad e_{bc}.$$

Replay immediately establishes

$$a \equiv c,$$

without requiring an additional primitive identification.

Thus collapse events generate connected components inside the historical dependency graph.

Observable quotient classes emerge naturally as connected regions of the collapse graph.

Unlike ordinary quotients, however, every internal construction remains accessible.

21.4 Collapse Closure

Collapse behaves transitively.

Theorem 21.1 (Collapse Closure)**.** *Suppose*

$$\mathsf{Collapse}(H, a, b, p) = H_1,$$

and

$$\mathsf{Collapse}(H_1, b, c, q) = H_2.$$

Then replay establishes

$$a \equiv c,$$

and there exists a derived collapse path connecting

$$a$$

to

$$c.$$

Proof. Replay of

$$p$$

establishes replay equivalence between

$$a$$

and

$$b.$$

Replay of

$$q$$

establishes replay equivalence between

$$b$$

and

$$c.$$

Replay equivalence is transitive.
Therefore the composite replay demonstrates equivalence between

$$a$$

and

$$c.$$

The collapse graph therefore contains a path joining the two constructions.

□

Observe that no new primitive event is required.

The existing collapse history already contains sufficient evidence.

21.5 Canonical Representatives

Many mathematical algorithms require choosing canonical representatives of equivalence classes.

Within the historical calculus this selection becomes entirely external.

The kernel itself never privileges one representative over another.

Instead, a projection

$$\pi_C : H \to A/\!\equiv$$

may be applied by external algorithms whenever canonicalization is desirable.

The trusted kernel therefore preserves complete provenance, while higher-level tools remain free to choose canonical representatives for computational convenience.

This separation avoids contaminating the logical foundation with arbitrary normalization policies.

21.6 Collapse and Optimization

Collapse naturally models optimization.

Suppose an optimizer discovers that two implementations

$$f$$

and

$$g$$

compute identical observable results.

Rather than replacing one implementation by the other, the optimizer records

$$\mathsf{Collapse}(H, f, g, p),$$

where

$$p$$

is the replay proof establishing observational equivalence.

The optimized program therefore remains historically connected to the original implementation.

Debugging, verification, and provenance are preserved automatically.

Optimization becomes another constructive theorem rather than an irreversible rewrite.

21.7 Collapse and Mathematical Discovery

This interpretation suggests a broader philosophical reading.

Mathematical progress frequently consists not of constructing entirely new objects but of recognizing that previously independent constructions describe the same phenomenon.

Classical examples include the identification of algebraic and geometric structures, dual formulations of physical theories, or multiple definitions of the same invariant.

Within the historical calculus such discoveries correspond precisely to collapse events.

Mathematics therefore evolves not only through the creation of new histories, but also through the progressive identification of previously separate regions of constructive knowledge.

The history grows both by extension and by recognition.

21.8 Collapse as Historical Compression

Repeated collapse events also induce structural compression.

Suppose a large proof library contains hundreds of constructions that are later shown replay equivalent.

The kernel retains every derivation.

However, replay need only verify one representative together with the collapse graph connecting the remaining constructions.

Storage therefore remains historically complete while verification becomes increasingly efficient.

This differs fundamentally from syntactic deduplication.

Compression is driven by mathematical equivalence rather than textual identity.

21.9 Collapse and Knowledge Organization

The interaction between Meld and Collapse now becomes apparent.

Meld enlarges the space of constructive histories.

Collapse reorganizes that enlarged space by recording newly discovered equivalences.

Knowledge therefore evolves through two complementary mechanisms.

One constructs.

The other recognizes.

Neither destroys information.

Together they generate an ever-expanding graph of mathematical knowledge whose logical correctness is guaranteed by replay and whose organization reflects the progressive discovery of historical equivalence.

This duality between expansion and identification forms one of the central structural principles of the Spherepop kernel.

The next chapter develops a categorical semantics for the historical calculus, showing that histories, replay morphisms, Meld, and Collapse naturally organize themselves into a category whose universal constructions recover many familiar structures from category theory while preserving the event-oriented ontology developed throughout this paper.

22 Categorical Semantics of Historical Construction

The preceding chapters have developed the historical calculus entirely from the internal perspective of the language. Histories, replay, Meld, Collapse, and dependent typing have been introduced as primitive computational notions. It is natural to ask whether these constructions admit a denotational semantics analogous to the categorical models traditionally used for the simply typed lambda calculus, the Calculus of Constructions, and Martin–Löf type theory.

The answer is affirmative.

Indeed, one of the motivations for treating histories rather than states as the primitive semantic objects is that histories possess an unusually natural categorical structure. Objects become histories. Morphisms become replayable historical transformations. Composition becomes replay composition. Identity becomes replay without modification. The primitive operations introduced earlier appear as universal constructions within this category.

The purpose of this chapter is not to replace the operational semantics already developed, but to demonstrate that the historical calculus admits a coherent denotational interpretation that explains why its computational rules fit together so naturally.

22.1 The Category of Histories

We begin by defining the fundamental category.

Definition 22.1 (Historical Category)**.** The category

$$\mathcal{H}$$

is defined as follows.

Objects are well-formed constructive histories,

$$H_1, H_2, \ldots$$

Morphisms

$$f : H_1 \to H_2$$

are replay-preserving historical transformations.

Composition is replay composition,

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

Identity morphisms are replay operations that leave histories unchanged.

Unlike categories constructed from sets or spaces, the objects themselves are computational artifacts. Every object possesses operational content, and every morphism represents an executable constructive process.

22.2 Identity Morphisms

Every history possesses a canonical identity morphism,

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

Operationally,

$$\mathrm{id}_H$$

performs replay without introducing any additional historical events. Consequently,

$$\mathrm{Replay}(H) = \mathrm{Replay}(\mathrm{id}_H(H)).$$

Identity therefore represents perfect historical preservation.

22.3 Composition

Suppose

$$f : H_1 \to H_2$$

and

$$g : H_2 \to H_3.$$

Composition is defined by replaying

$$f$$

followed by replaying

$$g.$$

Thus

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

Since replay is deterministic, composition is associative.

Theorem 22.2. *Replay composition satisfies*

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

The proof follows immediately from the associativity of historical replay.
Consequently,

$$\mathcal{H}$$

forms a well-defined category.

22.4 Historical Functors

Many computational transformations preserve historical structure.
Suppose

$$F : \mathcal{H} \to \mathcal{H}.$$

Then

$$F$$

is called a historical functor if it satisfies

$$F(\mathrm{id}_H) = \mathrm{id}_{F(H)},$$

and

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

Examples include replay normalization,
dependency compression,
history visualization,
incremental verification,
and proof serialization.
Each preserves constructive provenance while transforming its representation.

22.5 Natural Transformations

Suppose

$$F, G : \mathcal{H} \to \mathcal{H}.$$

A natural transformation

$$\eta : F \Rightarrow G$$

assigns every history

$$H$$

a replay-preserving morphism

$$\eta_H : F(H) \to G(H).$$

Naturality requires

$$G(f) \circ \eta_H = \eta_{H'} \circ F(f)$$

for every replay morphism

$$f : H \to H'.$$

Operationally, natural transformations represent history-preserving compiler passes or proof transformations that commute with replay itself.

22.6 Meld as a Pushout

The primitive operation

$$\mathsf{Meld}$$

introduced earlier now acquires a universal characterization.
Suppose

$$H_0$$

is a common historical prefix.
Then

$$H_1 \leftarrow H_0 \to H_2$$

forms a span inside

$$\mathcal{H}.$$

When compatibility holds,

$$\mathsf{Meld}(H_1, H_2)$$

is the pushout of this span.

Unlike ordinary categorical pushouts, however, the resulting object remains an executable history rather than an abstract universal construction.

The universal property therefore possesses direct computational content.

22.7 Collapse as a Coequalizer

Collapse admits a dual interpretation.

Suppose

$$f, g : H_1 \to H_2$$

are replay-equivalent morphisms.

The collapse history identifies them through

$$\mathsf{Collapse}(H, f, g).$$

Categorically,

this construction behaves as a coequalizer.

The historical calculus therefore internalizes quotient constructions without discarding provenance.

The universal property records not merely that two morphisms become equal but how that equality was constructively established.

22.8 Products

Independent histories naturally admit categorical products.

Given

$$H_1$$

and

$$H_2,$$

their product consists of the pair

$$(H_1, H_2),$$

equipped with the obvious projection morphisms.

Operationally,

products correspond to independent computations whose interaction has not yet been established.

Applying Meld transforms this external product into a unified historical construction.

Products therefore precede historical interaction.

Meld realizes that interaction.

22.9 Exponentials

Dependent products introduced earlier provide exponentials within the historical category.

Suppose

$$A$$

and

$$B$$

are historical objects.

Then

$$B^A$$

is represented by

$$\Pi x : A.B.$$

Exponentiation therefore corresponds to the construction of replay-preserving historical transformations.

The historical category thus possesses the structure expected of a cartesian closed category, enriched by explicit provenance.

22.10 Historical Toposes

The constructions developed throughout this paper strongly suggest that the historical calculus naturally generates a higher categorical structure.

Dependent products,
universes,
inductive families,
replay morphisms,
Meld,
and
Collapse
collectively resemble the internal language of a dependent category equipped with explicit provenance.

Whether the resulting semantics are best described as a comprehension category, a category with families, an indexed category, or an adhesive category remains an interesting direction for future investigation.

The present development establishes only that a coherent categorical semantics exists and that the primitive operations of the historical kernel arise naturally from familiar universal constructions.

22.11 Category Theory from Historical Computation

The significance of this chapter extends beyond denotational semantics.

Category theory often appears as an abstract mathematical language imposed upon computation from above.

The historical calculus suggests the opposite interpretation.

Universal constructions emerge because histories compose in particular ways.

Pushouts arise from constructive merging.

Coequalizers arise from historical collapse.

Exponentials arise from reusable historical transformations.

Products arise from independent histories.

Thus the familiar structures of category theory need not be postulated independently.

They emerge from the event algebra itself.

This observation reinforces one of the central themes of the paper.

The historical calculus is not merely another implementation of the Calculus of Constructions.

It proposes a different mathematical ontology in which provenance is primitive, computation is historical growth, and many familiar categorical constructions appear naturally as consequences of the geometry of constructive histories.

The following chapter returns to the implementation of the trusted kernel, showing how these mathematical structures lead to a remarkably compact kernel architecture capable of supporting dependent types, proof replay, distributed verification, and incremental compilation from a surprisingly small collection of primitive operations.

23 The Minimal Historical Kernel

The preceding chapters have introduced a considerable amount of mathematical structure. Histories, replay, dependent products, universes, inductive families, equality types, Meld, Collapse, and categorical semantics together constitute a rich foundation for constructive mathematics. One might therefore expect the trusted kernel required to implement these ideas to be correspondingly large.

Quite the opposite is true.

One of the central design goals of Spherepop is that expressive power should arise through composition rather than through the continual introduction of new primitive mechanisms. Every additional primitive increases the trusted computing base, complicates meta-theoretic analysis, and enlarges the surface over which implementation errors may occur.

The historical calculus therefore seeks the smallest collection of primitive operations from which the remainder of the system may be reconstructed.

The resulting kernel is significantly smaller conceptually than its observable language.

Surface syntax exists for human convenience.

The kernel exists only to preserve mathematical correctness.

23.1 The Principle of Historical Minimality

The guiding design principle may be stated as follows.

Every primitive operation admitted by the kernel should correspond to an irreducible transformation of constructive history.

All higher-level language constructs should elaborate into compositions of these primitive historical operations.

This principle distinguishes semantic necessity from syntactic convenience.

Features that improve readability need not appear inside the trusted kernel if they can be translated into more primitive constructions.

The implementation therefore separates elaboration from verification.

23.2 Primitive Objects

The kernel manipulates only a small collection of primitive mathematical objects.

These are

$$\mathsf{History},$$

the universe hierarchy,

$$\mathsf{Prop}, \qquad \mathsf{Type}_0, \qquad \mathsf{Type}_1, \ldots,$$

variables,
dependent products,
lambda abstractions,
applications,
and primitive historical events.

Everything else is constructed from these ingredients.

Even equality proofs, inductive eliminators, theorem declarations, and proof scripts elaborate into combinations of these elementary objects.

23.3 Primitive Historical Events

The event algebra itself remains intentionally compact.

The trusted kernel introduces only the following primitive historical events.

Declaration introduces a new construction into the current history.

Bind records constructive dependency.

Pop commits to one admissible continuation.

Refuse permanently removes an admissible continuation.

Collapse records replay-equivalent constructions.

Meld combines compatible histories.

Reduction records computational simplification.

Replay reconstructs previous derivations.

Every observable language feature ultimately expands into sequences of these primitive events.

Consequently, reasoning about the entire language reduces to reasoning about a very small event algebra.

23.4 Kernel Elaboration

The surface language presented to programmers may contain considerably richer syntax.

For example,

```
theorem identity :
  forall A : Type,
  A -> A
```

is not itself understood by the kernel.

Instead, elaboration produces

$$\lambda A : \mathsf{Type}_0.\, \lambda x : A.\, x,$$

together with the associated historical declarations, dependency edges, and replay information.

The kernel therefore verifies only the elaborated historical representation.

User-facing syntax remains entirely outside the trusted computing base.

23.5 Historical Verification Pipeline

Verification proceeds through a sequence of conceptually distinct phases.

A parser constructs the abstract syntax tree.

Elaboration translates surface syntax into the historical core language.

Universe inference assigns universe levels.

The bidirectional type checker constructs typing derivations.

Replay normalization computes historical normal forms.

Historical conversion verifies replay equivalence.

Finally, the completed history is committed to the persistent mathematical archive.

Only the final stages belong to the trusted kernel.

Earlier stages may be replaced, optimized, or rewritten without affecting the logical foundation, provided they continue to elaborate into equivalent historical representations.

23.6 Small Trusted Computing Base

One of the advantages of the historical architecture is that the trusted computing base remains unusually compact.

Only a handful of components require complete mathematical confidence.

The event algebra,
the replay engine,
the normalization procedure,
the universe checker,
and
the bidirectional type checker.

Automation,
search,
tactics,
optimizers,
compilers,
language servers,
and
interactive editors
operate entirely outside the trusted kernel.

Their output acquires mathematical legitimacy only after replay reconstructs the complete historical derivation.

This separation considerably simplifies formal verification of the kernel itself.

23.7 Historical Persistence

Unlike conventional proof kernels, no verified object is ever rewritten in place.

Once committed,
a historical construction becomes immutable.

Later mathematical developments extend the existing archive rather than modifying it.

Consequently,

every published theorem possesses a stable historical identifier determined by its construction history rather than by its textual location.

This property naturally supports persistent mathematical databases, distributed repositories, reproducible formal developments, and long-term archival verification.

The kernel therefore behaves more like a persistent version-control system than a mutable theorem database.

23.8 Historical Garbage Collection

Because histories are immutable, unused constructions may be analyzed without affecting correctness.

Suppose a historical branch is no longer reachable from any exported theorem.

The kernel need not immediately discard it.

Instead, archival policies may preserve,

compress,

or remove unreachable historical regions according to external storage requirements.

Logical correctness remains unchanged because replay depends only upon reachable historical prefixes.

Garbage collection therefore becomes an optimization problem rather than a logical one.

23.9 Kernel Correctness

The historical kernel is designed so that every accepted theorem satisfies three simultaneous criteria.

First,

its observable proof term is well typed.

Second,

its historical derivation is replayable.

Third,

its dependency graph satisfies the admissibility conditions imposed by the event algebra.

These three properties together define correctness.

Observable validity alone is insufficient.

Constructive provenance is regarded as an equally fundamental component of mathematical truth.

23.10 Minimality Through Composition

The development of this chapter illustrates a recurring theme throughout the paper.

The expressive richness of Spherepop does not arise from a proliferation of primitive language constructs.

Instead, it emerges from repeated composition of a remarkably small collection of historical operations.

Dependent type theory,
proof replay,
incremental verification,
distributed theorem proving,
categorical semantics,
historical equality,
and
event-sourced mathematical archives
all arise from the same compact kernel.

The apparent complexity of the language is therefore largely an illusion of composition.

The trusted core remains mathematically small.

This economy of primitives is one of the principal motivations for adopting an event-oriented rather than state-oriented foundation.

In the following chapter we prove the principal metatheoretic properties of the historical calculus, including preservation, progress, confluence, strong normalization of the logical fragment, replay determinism, admissibility preservation, and the consistency of the historical universe hierarchy. These results establish that the historical kernel is not merely expressive but also mathematically well behaved.

24 Meta-Theory of the Historical Calculus

The preceding chapters have developed the syntax, operational semantics, dependent type theory, historical event algebra, categorical semantics, and minimal kernel architecture of Spherepop. A logical foundation, however, is not complete until it is accompanied by a corresponding metatheory. The purpose of this chapter is to establish the principal structural properties of the historical calculus and to demonstrate that the introduction of explicit construction histories preserves the mathematical guarantees expected of a dependently typed proof kernel.

The remarkable feature of the historical calculus is that most classical metatheoretic arguments survive with only modest modification. The observable language remains close to the Calculus of Constructions, while the historical component evolves monotonically through event extension. Consequently, the classical proofs are strengthened rather than replaced. Each traditional theorem acquires an additional historical invariant concerning replay and provenance.

Throughout this chapter we assume that every history under consideration is well-formed, acyclic, and satisfies the admissibility conditions introduced in earlier chapters.

24.1 Historical Well-Formedness

The first property concerns the internal consistency of histories themselves.

Definition 24.1 (Well-Formed History)**.** A history

$$H$$

is well formed if and only if

1. every event satisfies its typing rule,
2. every dependency references an earlier event,
3. every replay succeeds,
4. every collapse is justified by replay equivalence,
5. every Meld satisfies compatibility,
6. every universe assignment respects the cumulative hierarchy.

Well-formedness is preserved throughout computation.

No primitive operation of the kernel may construct an ill-formed history.

24.2 Historical Weakening

Weakening expresses the monotonic growth of histories.

Theorem 24.2 (Historical Weakening)**.** *Suppose*

$$H \vdash t : A$$

and

$$e$$

is independent of the derivation of

$$t.$$

Then

$$H; e \vdash t : A.$$

Proof. The additional event contributes no dependency edge into the derivation of

$$t.$$

Replay therefore reconstructs the original typing derivation unchanged.

The extended history merely enlarges the available space of future constructions.

□

Weakening therefore acquires an intuitive operational interpretation.

Appending independent historical information cannot invalidate previous mathematics.

24.3 Historical Substitution

Substitution remains the fundamental structural theorem of dependent type theory.

Theorem 24.3 (Historical Substitution)**.** *Suppose*

$$H \vdash a : A$$

and

$$H, x : A \vdash t : B.$$

Then

$$H \vdash t[x := a] : B[x := a].$$

Furthermore,
the construction history of

$$t[x := a]$$

contains the merged dependency graph of

$$t$$

and

$$a.$$

The observable statement is identical to ordinary dependent type theory.

The historical statement strengthens it by describing the provenance of the resulting construction.

24.4 Preservation

Type preservation guarantees that evaluation cannot change the type of a well-typed expression.

Theorem 24.4 (Historical Preservation)**.** *Suppose*

$$H \vdash t : A$$

and

$$(H, t) \longrightarrow (H', u).$$

Then

$$H' \vdash u : A.$$

Proof. Proceed by induction over the evaluation derivation.
Beta reduction preserves typing by historical substitution.
Replay preserves dependency structure.
Every primitive event extends rather than modifies the existing history.
Consequently, the resulting construction remains well typed.

□

Unlike the classical theorem, preservation now concerns both observable terms and historical provenance.

24.5 Progress

Progress guarantees that computation never becomes stuck.

Theorem 24.5 (Historical Progress)**.** *Suppose*

$$\varnothing \vdash t : A.$$

Then either

$$t$$

is a value,
or there exists

$$(H', u)$$

such that

$$(\varepsilon, t) \longrightarrow (H', u).$$

The theorem differs only slightly from its classical counterpart.
The resulting configuration includes the newly generated history.
Evaluation therefore produces both computation and provenance.

24.6 Replay Determinism

One of the distinctive properties of the historical calculus is replay determinism.

Theorem 24.6 (Replay Determinism)**.** *Suppose*

$$\text{Replay}(H) = (H_1, t_1)$$

and

$$\text{Replay}(H) = (H_2, t_2).$$

Then

$$H_1 = H_2,$$

and

$$t_1 = t_2.$$

Proof. Replay follows the unique sequence of events recorded within the immutable history.

No nondeterministic choices remain.

Therefore every replay reconstructs exactly the same derivation.

□

Replay determinism provides the foundation for reproducible theorem verification.

24.7 Confluence

Evaluation order should not affect the observable result.

Theorem 24.7 (Historical Confluence)**.** *Suppose*

$$(H, t) \Longrightarrow (H_1, u_1)$$

and

$$(H, t) \Longrightarrow (H_2, u_2).$$

Then there exists

$$(H_3, v)$$

such that

$$(H_1, u_1) \Longrightarrow (H_3, v)$$

and

$$(H_2, u_2) \Longrightarrow (H_3, v).$$

Replay equivalence ensures that the resulting histories describe the same observable computation even when independent reductions occur in different orders.

24.8 Strong Normalization

The logical fragment of the calculus remains strongly normalizing.

Theorem 24.8 (Historical Strong Normalization)**.** *Every well-typed proof term admits a finite replay history terminating in a historical normal form.*

Consequently,
logical proofs cannot conceal infinite computation inside replay.
The introduction of historical events therefore preserves the constructive consistency of the kernel.

24.9 Canonicity

Closed inhabitants of inductive types reduce to canonical constructors.
The historical calculus strengthens this familiar theorem.

Theorem 24.9 (Historical Canonicity)**.** *Suppose*

$$\varnothing \vdash n : \mathsf{Nat}.$$

Then replay terminates in a history

$$H$$

whose observable value is
either

$$\mathsf{Zero},$$

or

$$\mathsf{Succ}(m),$$

and whose construction history consists entirely of admissible applications of the natural number constructors.

Thus canonical forms possess canonical provenance.

24.10 Consistency of the Universe Hierarchy

Finally,
the historical interpretation preserves logical consistency.

Theorem 24.10 (Historical Universe Consistency)**.** *No replayable history constructs a universe satisfying*

$$\mathsf{Type} : \mathsf{Type}.$$

Proof. Such a construction would necessarily introduce a cyclic dependency into the historical graph.

All histories are required to remain acyclic.

Therefore no admissible replay can generate such a universe assignment.

□

The historical interpretation therefore supplies an operational explanation for predicativity.

Logical inconsistency appears as an impossible historical construction.

24.11 Summary

The metatheory developed in this chapter demonstrates that the historical calculus preserves the essential guarantees expected of a dependable proof kernel while enriching each theorem with additional information concerning constructive provenance. Preservation becomes preservation of history as well as type. Progress produces both values and derivations. Confluence extends to replay. Canonicity identifies canonical histories in addition to canonical values. Strong

normalization guarantees termination of both computation and historical reconstruction.

The historical calculus therefore inherits the mathematical robustness of the Calculus of Constructions while providing a richer semantic foundation in which proofs, computations, dependencies, and provenance are unified within a single event-oriented framework.

The next chapter turns from metatheory to implementation, presenting a complete reference architecture for the Spherepop kernel, including the representation of histories, replay algorithms, normalization procedures, dependency indexing, and persistent proof storage.

25 Reference Implementation of the Historical Kernel

The preceding chapters have established the mathematical foundations of the historical calculus. We now turn to its implementation. The objective of this chapter is not to prescribe a particular programming language or runtime, but to identify the minimal computational architecture capable of faithfully realizing the formal semantics developed throughout the paper.

Unlike most implementations of dependent type theory, the Spherepop kernel is organized around persistent histories rather than mutable environments. The central data structure is therefore not a context or symbol table but an immutable directed acyclic graph describing the evolution of constructive knowledge.

Every other subsystem is organized around this graph.

25.1 The Historical Database

At the heart of the implementation lies the historical database

$$\mathcal{D}.$$

Rather than storing isolated declarations,

$$\mathcal{D}$$

stores immutable historical objects

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

Each history consists of

$$H = (E, D),$$

where

$$E$$

is the finite collection of historical events and

$$D$$

is the dependency graph connecting them.
Every event possesses a globally unique identifier,
a timestamp or logical ordering,
its event constructor,
its arguments,
its resulting object,
and its replay metadata.
The kernel never modifies an existing event.
Instead,
new mathematical knowledge appends additional events to the existing database.

25.2 Historical Object Representation

Every mathematical object stored by the kernel possesses the form

$$O = (I, T, H),$$

where

$$I$$

is a globally unique identifier,

$$T$$

is its observable term,
and

$$H$$

is its construction history.

The identifier is intentionally independent of the textual name supplied by the user.

Names are merely convenient aliases.

Identity is determined entirely by constructive provenance.

Consequently,

renaming a theorem does not alter the mathematical object represented by the kernel.

25.3 Persistent Dependency Graphs

The dependency graph forms the principal indexing structure.

Every edge

$$u \to v$$

records that the construction represented by

$$v$$

depends directly upon the construction represented by

$$u.$$

Because histories are immutable,

the graph itself is persistent.

Insertion creates additional vertices and edges without modifying existing ones.

Older versions of the mathematical library therefore remain permanently available.

Incremental verification simply traverses the reachable portion of this graph.

No global recompilation is required.

25.4 Replay Engine

Replay is implemented as a deterministic graph traversal.

Beginning from the empty history,
the replay engine visits historical events in dependency order.
Each event invokes the corresponding primitive kernel operation.
The resulting object is inserted into an in-memory reconstruction of the historical state.
If every replay step succeeds,
the history is accepted.
Otherwise,
the first failed replay identifies the precise location of the inconsistency.
Replay therefore functions simultaneously as
an evaluator,
a proof checker,
and
a debugger.
The same algorithm serves all three purposes.

25.5 Normalization Engine

Observable normalization remains necessary for conversion checking.
Rather than traversing arbitrary syntax trees,
the historical kernel performs graph-guided normalization.
Only those subgraphs reachable from the requested construction are evaluated.
Shared historical prefixes are normalized once and cached.
Subsequent requests simply reference the cached normal forms.
Consequently,
the computational cost of normalization depends upon the size of the reachable dependency graph rather than the size of the complete mathematical library.

25.6 Historical Memoization

One of the principal implementation advantages of immutable histories is the simplicity of memoization.
Suppose replay has already reconstructed the history

$$H.$$

Any future computation depending upon

$$H$$

may reuse the cached result without re-executing the corresponding events.

Because histories cannot change,

cached replay results never become invalid.

Only descendants of newly added events require recomputation.

Historical memoization therefore follows directly from immutability.

No cache invalidation protocol beyond reachability analysis is required.

25.7 Parallel Replay

Replay naturally exposes parallelism.

Suppose two histories satisfy

$$H_1 \parallel H_2.$$

The replay engine constructs

$$\text{Replay}(H_1)$$

and

$$\text{Replay}(H_2)$$

simultaneously.

Synchronization occurs only when a subsequent Meld event combines the two histories.

This organization avoids global scheduling.

The dependency graph itself determines the available concurrency.

Parallel execution therefore follows directly from the mathematical semantics.

25.8 Storage Optimization

Historical persistence naturally suggests several compression strategies.

Repeated replay prefixes are stored only once.

Replay-equivalent histories are connected through collapse graphs rather than duplicated.

Immutable subgraphs are reference counted.

Frequently accessed normal forms are cached.
Because none of these optimizations alter observable replay,
they remain entirely outside the trusted logical foundation.
The mathematical semantics remain unchanged.
Only storage efficiency improves.

25.9 Historical Transactions

Insertion of new mathematical knowledge proceeds transactionally.
A candidate history is first constructed in isolation.
Replay then verifies the complete derivation.
Only after successful verification is the history merged into the persistent database.
Consequently,
the persistent archive always consists entirely of replayable histories.
No partially verified mathematics ever enters the trusted repository.
This transaction model closely resembles append-only databases,
except that correctness is established through formal replay rather than through consistency constraints alone.

25.10 Reference Kernel Algorithm

The implementation developed throughout this chapter may be summarized abstractly by the following recursive procedure.
Given a proposed historical extension,
first verify that every referenced dependency already exists.
Next,
replay the referenced histories.
Then,
execute each newly introduced historical event in dependency order.
Normalize every observable construction required for conversion checking.
Verify every typing judgment.
Record every replay result.
Finally,
commit the completed immutable history to the persistent archive.
Observe that no mutable global state appears anywhere in this procedure.
The only persistent mathematical object is the growing history itself.

25.11 Implementation Philosophy

The implementation architecture reflects the central philosophical principle of Spherepop.

Mathematics is not a mutable collection of declarations.

It is a persistent history of constructive events.

The kernel therefore resembles an event-sourced database more closely than a traditional theorem prover.

Every theorem is an immutable transaction.

Every proof is a replayable construction.

Every optimization preserves provenance.

Every extension enlarges rather than replaces the existing mathematical record.

This architecture separates the trusted kernel from every higher-level language feature. Parsers, elaborators, optimizers, proof search procedures, integrated development environments, and compiler back ends all become replaceable front-end tools. Their only responsibility is to generate candidate historical constructions. The kernel itself remains responsible solely for replay, verification, normalization, and historical persistence.

Having completed the implementation architecture, the remaining chapters turn toward future directions. We shall investigate cubical historical equality, higher inductive histories, distributed theorem proving, verified compilation, and the possibility that constructive histories provide a common mathematical foundation for programming languages, proof assistants, version-control systems, and persistent knowledge archives.

26 Future Directions

The historical calculus presented throughout this paper should be regarded as the foundation of a broader research programme rather than as a completed theory. The primary objective has been to demonstrate that constructive histories can replace unordered contexts as the primitive semantic objects of a dependently typed calculus while preserving the expressive power of the Calculus of Constructions. Many natural extensions remain open.

These extensions do not require abandoning the historical kernel. Instead, they represent successive enrichments of the same event-oriented foundation.

26.1 Cubical Historical Equality

One of the most promising directions concerns computational equality.

Modern cubical type theories replace extensional identity with explicit paths that possess computational content. The historical interpretation developed in this paper suggests an alternative viewpoint.

A path may be interpreted as a continuous replay transformation between two construction histories.

Instead of asking whether two objects are connected by a path, one asks whether their histories admit a continuous family of replay operations preserving constructive admissibility.

Collapse would then become the endpoint of a replay homotopy rather than merely an isolated historical event.

This interpretation may provide a bridge between cubical computation and event-oriented semantics.

26.2 Higher Inductive Histories

The present development has considered only ordinary inductive families.

Higher inductive types suggest a corresponding notion of higher historical generators.

Constructors would no longer generate only objects.

They would generate histories,

replay morphisms,

collapse witnesses,

and higher replay equivalences simultaneously.

The resulting structures could provide a computational foundation for topology, geometry, and higher category theory directly within the historical kernel.

26.3 Verified Distributed Mathematics

The Meld operation naturally supports distributed theorem proving.

An important direction for future work is the development of globally distributed mathematical repositories in which independently verified histories are continuously exchanged between cooperating kernels.

Because every theorem is replayable,

trust need not depend upon the originating machine.

Only replay correctness matters.

The resulting mathematical archive would resemble a persistent distributed graph of constructive knowledge rather than a centralized library.

26.4 Verified Compilation

The historical interpretation naturally extends to compiler construction.

Compilation itself may be represented as a replay-preserving historical transformation,

$$C : \mathcal{H}_{\text{source}} \to \mathcal{H}_{\text{target}}.$$

Rather than proving only semantic preservation,

one may prove preservation of constructive provenance.

Every machine instruction would remain historically connected to the source construction that generated it.

Compiler verification would therefore become another instance of replay verification.

26.5 Historical Operating Systems

The event-oriented ontology developed throughout this paper suggests that the kernel may extend beyond proof assistants.

Operating systems traditionally organize computation around mutable process state.

An alternative architecture would organize computation around persistent histories.

Processes,

files,

inter-process communication,

resource allocation,

and synchronization

could all be represented as replayable historical constructions.

The distinction between operating system,

version-control system,

proof assistant,

and database would thereby become increasingly blurred.

26.6 Persistent Scientific Computation

Scientific computing frequently suffers from incomplete reproducibility.

Intermediate computations disappear.

Parameter choices are forgotten.

Data transformations become difficult to reconstruct.

The historical calculus provides an alternative.

Every simulation,

every optimization,

every numerical approximation,

and every visualization

may be represented as replayable historical constructions.

Scientific publications would therefore include executable historical proofs of their computational claims rather than static numerical results alone.

26.7 Historical Artificial Intelligence

Machine learning systems typically discard the majority of their intermediate reasoning.

Weights remain,

while the constructive history producing those weights largely disappears.

The historical kernel suggests a different approach.

Learning becomes the accumulation of admissible historical constructions.

Inference becomes replay over previously established histories.

Optimization records explicit collapse events identifying equivalent representations.

Interpretability follows naturally because every prediction possesses an explicit construction history.

Rather than explaining decisions after the fact,

the explanation becomes intrinsic to the computation itself.

26.8 Historical Mathematics

Perhaps the most ambitious direction concerns the philosophy of mathematics.

The traditional foundations of mathematics emphasize completed structures,

sets,

functions,

or categories.

The historical calculus instead proposes that mathematical existence should be understood through admissible constructive history.

Objects exist because they have been constructed.

Identity arises from replay.

Equivalence arises through collapse.

Knowledge grows through Meld.

The resulting ontology replaces static mathematical universes with persistent historical evolution.

Whether this viewpoint ultimately provides a more satisfactory foundation for constructive mathematics remains an open question deserving substantial future investigation.

26.9 A Unified Historical Foundation

The various research directions outlined above are united by a common principle.

The same event algebra governing dependent type theory also appears naturally in

proof assistants,

distributed systems,

compiler verification,

persistent databases,

version-control systems,

scientific computation,

and

artificial intelligence.

These disciplines have traditionally developed largely independently despite sharing many underlying structural ideas concerning dependency, persistence, reproducibility, and constructive growth.

The historical calculus suggests that these similarities are not accidental.

They arise because all of these systems manipulate histories rather than merely states.

A sufficiently general mathematical theory of constructive histories therefore has the potential to unify ideas that presently appear in separate areas of computer science and constructive mathematics.

The work presented in this paper represents only the first step toward such a unification. It establishes a minimal historical kernel, demonstrates that the

Calculus of Constructions may be reconstructed within that kernel, and develops its operational, logical, categorical, and implementation-theoretic consequences. Much remains to be explored, but the central conclusion already appears clear: histories are not merely implementation artifacts. They are mathematical objects in their own right, and treating them as such opens a wide range of new possibilities for programming languages, proof assistants, and the foundations of constructive reasoning.

The final chapter concludes by summarizing the historical programme and reconsidering the relationship between computation, proof, identity, and construction from the perspective developed throughout this monograph.

27 Conclusion

The principal objective of this paper has been to investigate whether the Calculus of Constructions may be reconstructed upon a different mathematical foundation. Rather than taking lambda abstraction, substitution, and unordered typing contexts as primitive, we have proposed that irreversible constructive histories constitute the fundamental objects of computation and reasoning.

The resulting historical calculus preserves the expressive capabilities of the Calculus of Constructions while assigning a different interpretation to nearly every one of its central notions.

Contexts become histories.

Substitution becomes historical inheritance.

Evaluation becomes replay.

Equality becomes constructive historical equivalence.

Proofs become immutable event histories.

Normalization becomes reconstruction.

Dependency becomes explicit.

The trusted kernel becomes a persistent event algebra.

Throughout the development of the paper, every familiar component of dependent type theory has been reconstructed from this perspective.

Dependent products were interpreted as admissible continuations of constructive history.

Universes became classifications of historical modes of construction rather than merely collections of types.

Inductive families became generators of admissible historical growth.

Equality types became replay witnesses relating independent construction histories.

The Curry–Howard correspondence expanded naturally into a correspondence among propositions, types, programs, and histories.

Category theory emerged from replay composition rather than being imposed externally.

Implementation itself reduced to the maintenance of a persistent graph of historical events.

None of these developments required abandoning the computational behavior of the Calculus of Constructions.

Observable programs remain lambda terms.

Dependent products continue to represent universal quantification.

Normalization computes the same observable results.

The familiar mathematics therefore survives intact.

What changes is the ontology lying beneath it.

Observable syntax becomes the visible projection of a richer constructive history.

This shift in perspective has several mathematical consequences.

First, provenance becomes intrinsic rather than auxiliary.

Every theorem contains not merely a proof term but the complete historical derivation that produced it.

Replay therefore reconstructs mathematical reasoning directly from immutable historical events.

Explanation becomes a mathematical property rather than a debugging tool.

Second, persistence replaces mutation.

Mathematical knowledge no longer evolves through replacement.

Instead, it grows through irreversible historical extension.

Earlier constructions remain permanently available.

Later discoveries enlarge the existing history rather than rewriting it.

The resulting mathematical archive resembles an event-sourced knowledge graph rather than a mutable collection of declarations.

Third, the primitive operations of the historical kernel naturally divide into two complementary classes.

Meld enlarges constructive history by combining independent developments.

Collapse reorganizes that enlarged history by recording newly discovered equivalences.

Knowledge therefore evolves through construction and recognition rather than through deletion and replacement.

Both mechanisms preserve provenance.

Neither destroys information.

Fourth, the implementation architecture becomes unusually compact.

Because histories are immutable,

incremental verification,

persistent caching,

parallel replay,

dependency analysis,

distributed theorem proving,

and

proof explanation

all arise naturally from the same mathematical representation.

The trusted kernel remains small precisely because these capabilities emerge from the historical semantics rather than from additional implementation machinery.

Perhaps the most significant philosophical consequence concerns the nature of mathematical identity itself.

Traditional foundations often begin with completed mathematical objects and ask how they may be related.

The historical calculus proceeds in the opposite direction.

Construction comes first.

Identity emerges from replay.

Equivalence is established through explicit collapse.

Objects are therefore understood not as primitive entities but as persistent historical constructions.

Their identities are consequences of how they were built.

This inversion of perspective extends naturally beyond proof assistants.

Persistent databases,

distributed systems,

version-control systems,

compiler verification,

scientific computation,

knowledge management,

and

interactive theorem proving

all manipulate histories whose provenance is often as important as their final observable state.

The historical calculus suggests that these systems may share a common mathematical foundation.

Their apparent differences arise primarily from the particular historical events they manipulate rather than from fundamentally different computational principles.

Whether this broader unification can be realized remains an open question.

The present work should therefore be understood as a foundational proposal rather than a completed theory.

Many directions remain to be explored.

A complete normalization proof for the historical calculus,

a mechanized metatheory,

cubical historical equality,

higher historical inductive families,

verified distributed proof systems,

categorical completeness,

compiler correctness,

and

large-scale implementation

all represent substantial future research programmes.

Nevertheless, the central thesis of the paper may already be stated with some confidence.

The expressive power of the Calculus of Constructions does not depend upon taking lambda calculus or unordered contexts as primitive.

The same logical structure may instead be reconstructed from a considerably more operational foundation based upon irreversible constructive histories.

Within this framework,

computation is the extension of history,

proof is replay,

identity is historical equivalence,

and

mathematics itself becomes the persistent accumulation of admissible construction.

The historical calculus therefore offers not merely another implementation of

dependent type theory but an alternative conceptual foundation for constructive mathematics. It retains the logical precision of the Calculus of Constructions while replacing static ontology with dynamic construction, transforming proofs from isolated symbolic objects into replayable historical processes and suggesting that the true primitive of computation is not state, nor term, nor set, but the irreversible history by which mathematical structure is brought into existence.

Appendices

A Formal Grammar of the Spherepop Kernel

This appendix presents the complete abstract syntax of the historical kernel. Only this grammar is trusted by the implementation. Surface syntax, theorem declarations, notation, modules, tactics, and parser conveniences elaborate into the abstract language defined here.

A.1 Lexical Categories

We assume countably infinite collections of

$$x, y, z, \ldots$$

for variables,

$$c, d, \ldots$$

for constructors,
and

$$u, v, \ldots$$

for universe levels.
Universe indices satisfy

$$u \in \mathbb{N}.$$

A.2 Terms

The set of kernel terms is generated inductively by

$$
\begin{aligned}
t ::= {} & x \\
\mid {} & \lambda x : A.\, t \\
\mid {} & t\, u \\
\mid {} & \Pi x : A.\, B \\
\mid {} & \Sigma x : A.\, B \\
\mid {} & \mathsf{Type}_u \\
\mid {} & \mathsf{Prop} \\
\mid {} & \mathsf{History} \\
\mid {} & \mathsf{Replay} \\
\mid {} & \mathsf{Pop} \\
\mid {} & \mathsf{Refuse} \\
\mid {} & \mathsf{Collapse} \\
\mid {} & \mathsf{Meld} \\
\mid {} & \mathsf{Bind} \\
\mid {} & \mathsf{Declare} \\
\mid {} & \mathsf{Inductive} \\
\mid {} & \mathsf{Constructor} \\
\mid {} & \mathsf{Elim}.
\end{aligned}
$$

Every observable construction appearing in Spherepop elaborates into terms of this grammar.

B Formal Event Algebra

Events are generated by

$$
e ::= \mathsf{Declare} \mid \mathsf{Bind} \mid \mathsf{Pop} \mid \mathsf{Refuse} \mid \mathsf{Collapse} \mid \mathsf{Meld} \mid \mathsf{Reduce} \mid \mathsf{Replay}\,.
$$

A history is a finite sequence

$$
H = e_1; e_2; \cdots; e_n
$$

satisfying

$$\forall i < j,\ e_j \text{ depends only upon } e_1, \ldots, e_i.$$

Thus every history determines a finite directed acyclic graph

$$G(H) = (V, E).$$

The dependency graph is uniquely reconstructed from replay.

C Complete Typing Rules

This appendix presents the declarative typing system of the historical kernel in its entirety. The purpose of collecting the rules in one location is to provide a precise specification independent of the informal exposition in the main body of the paper.

Throughout this appendix

$$H$$

denotes a well-formed historical context,

$$A, B, C$$

denote types,
and

$$s$$

ranges over the universe hierarchy

$$\mathsf{Prop}, \mathsf{Type}_0, \mathsf{Type}_1, \ldots.$$

Every judgment has the form

$$H \vdash t : A.$$

Unlike conventional dependent type theory,
the context

$$H$$

is an immutable constructive history rather than an unordered list of assumptions.

C.1 Historical Well-Formedness

The empty history is well formed.

$$\frac{}{\vdash \varepsilon}$$

Historical extension preserves well-formedness whenever the appended event is admissible.

$$\frac{\vdash H \qquad \text{Admissible}(H, e)}{\vdash H; e}$$

C.2 Variable Rule

Variables are introduced through declaration events.

$$\frac{\text{Declare}(x : A) \in H}{H \vdash x : A}$$

The declaration event uniquely determines the provenance of the variable.

C.3 Universe Rules

The historical kernel adopts cumulative universes.

$$\frac{}{H \vdash \mathsf{Prop} : \mathsf{Type}_0}$$

$$\frac{}{H \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}$$

Universe lifting is admissible.

$$\frac{H \vdash A : \mathsf{Type}_i}{H \vdash A : \mathsf{Type}_{i+1}}$$

C.4 Dependent Product Formation

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

C.5 Dependent Sum Formation

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

C.6 Lambda Introduction

$$\frac{H, x : A \vdash t : B}{H \vdash \lambda x : A.t : \Pi x : A.B}$$

C.7 Application

$$\frac{H \vdash f : \Pi x : A.B \qquad H \vdash a : A}{H \vdash f\, a : B[x := a]}$$

Historical substitution merges the dependency graphs of

$$f$$

and

$$a.$$

C.8 Pair Formation

$$\frac{H \vdash a : A \qquad H \vdash b : B[a/x]}{H \vdash (a, b) : \Sigma x : A.B}$$

C.9 Projection

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

$$\frac{H \vdash p : \Sigma x : A.B}{H \vdash \pi_2(p) : B[\pi_1(p)/x]}$$

C.10 Conversion

Replay equivalence induces definitional equality.

$$\frac{H \vdash t : A \qquad A \equiv_H B}{H \vdash t : B}$$

The equivalence

$$A \equiv_H B$$

is determined by replay normalization.

C.11 Historical Weakening

$$\frac{H \vdash t : A \qquad \text{Independent}(e, t)}{H; e \vdash t : A}$$

Appending an unrelated historical event cannot invalidate an existing typing judgment.

C.12 Historical Substitution

$$\frac{H \vdash a : A \qquad H, x : A \vdash t : B}{H \vdash t[x := a] : B[x := a]}$$

The resulting construction history contains the merged dependency graph of both premises.

C.13 Historical Equality

$$\frac{H \vdash a : A \qquad H \vdash b : A}{H \vdash \text{Eq}_A(a, b) : \mathsf{Prop}}$$

Reflexivity is introduced by

$$\frac{H \vdash a : A}{H \vdash \text{refl}(a) : \text{Eq}_A(a, a)}$$

Historical transport follows replay equivalence.

C.14 Historical Replay

Replay preserves typing.

$$\frac{H \vdash t : A}{\text{Replay}(H) \vdash t : A}$$

This rule forms the logical foundation of replay verification.

C.15 Historical Meld

Suppose

$$H_1 \parallel H_2.$$

Then

$$\frac{H_1 \vdash t : A \qquad H_2 \vdash u : B}{\mathsf{Meld}(H_1, H_2) \vdash (t, u) : A \times B}$$

provided the compatibility proof establishes that no dependency conflicts arise.

C.16 Historical Collapse

Suppose replay establishes

$$a \equiv_H b.$$

Then

$$\frac{H \vdash p : \mathrm{Eq}_A(a, b)}{\mathsf{Collapse}(H, a, b, p) \vdash a \equiv b}$$

Collapse therefore records equivalence as an explicit historical event rather than an extensional axiom.

C.17 Summary

These rules constitute the complete declarative specification of the historical kernel. Every higher-level language construct introduced by the Spherepop surface language elaborates into derivations built entirely from these rules. The trusted implementation therefore requires no additional logical primitives. The subsequent appendix develops the operational semantics corresponding to this typing system, giving the complete small-step and big-step evaluation relations for replay-driven computation.

D Operational Semantics

This appendix gives the formal operational semantics of the historical kernel. Unlike conventional presentations of the Calculus of Constructions, evaluation is not defined solely as a relation upon terms. Every computational step simultaneously transforms an observable expression and extends the constructive history responsible for that transformation.

The operational semantics therefore act upon configurations

$$(H, t),$$

rather than isolated terms

$$t.$$

The history is an essential component of the computational state.

D.1 Configurations

A configuration is defined by

$$C = (H, t),$$

where

$$H : \mathsf{History}$$

is a well-formed history and

$$t$$

is a well-typed kernel term.
Evaluation transforms configurations according to

$$(H, t) \longrightarrow (H', u).$$

The resulting history

$$H'$$

extends

$$H$$

through one additional primitive event.

D.2 Historical Values

Values are defined inductively.

$$v ::= \lambda x : A.t \mid \mathsf{Type}_i \mid \mathsf{Prop} \mid c \mid (v_1, v_2)$$

where

$$c$$

ranges over inductive constructors.
A configuration

$$(H, v)$$

is terminal whenever no reduction rule applies.

D.3 Historical Evaluation Contexts

Evaluation proceeds according to left-to-right historical contexts.

$$E ::= [\,] \mid E\,t \mid v\,E \mid (E, t) \mid (v, E)$$

Whenever

$$(H, t) \to (H', u),$$

evaluation contexts satisfy

$$(H, E[t]) \to (H', E[u]).$$

Thus replay preserves evaluation structure.

D.4 Beta Reduction

The principal computational rule is historical beta reduction.

$$\frac{}{(H, (\lambda x : A.t)\,a) \longrightarrow (H; e_\beta, t[x := a])}$$

where

$$e_\beta = \mathsf{Reduce}_\beta(\lambda x : A.t, a).$$

Unlike ordinary beta reduction,

the reduction event becomes part of the persistent history.

D.5 Projection

Dependent pairs reduce by projection.

$$\overline{(H, \pi_1(a, b)) \to (H; e_{\pi_1}, a)}$$

and

$$\overline{(H, \pi_2(a, b)) \to (H; e_{\pi_2}, b)}$$

The projection events preserve replayability.

D.6 Historical Replay

Replay reconstructs configurations recursively.

The empty history satisfies

$$\text{Replay}(\varepsilon) = (\varepsilon, \varnothing).$$

Suppose

$$H' = H; e.$$

Then

$$\text{Replay}(H') = \text{Execute}(e, \text{Replay}(H)).$$

Thus replay is defined inductively over historical extension.

D.7 Replay Execution

Each primitive event possesses an execution rule.

Declaration introduces a variable into the replay environment.

Bind inserts a dependency edge.

Pop selects an admissible continuation.

Refuse removes an admissible continuation from the future search space.

Collapse records replay equivalence.

Meld combines compatible replay states.

Reduction evaluates the associated computation.
Replay itself recursively reconstructs earlier events.
No additional operational primitives are required.

D.8 Historical Meld

Suppose

$$H_1 \parallel H_2.$$

Replay satisfies

$$\mathsf{Replay}(\mathsf{Meld}(H_1, H_2)) = \mathsf{Meld}(\mathsf{Replay}(H_1), \mathsf{Replay}(H_2)).$$

Thus replay commutes with historical composition.
This property provides the semantic foundation for parallel verification.

D.9 Historical Collapse

Suppose

$$p : \mathrm{Eq}_A(a, b).$$

Replay of

$$\mathsf{Collapse}(H, a, b, p)$$

first reconstructs

$$H,$$

then verifies

$$p,$$

and finally records the new collapse edge joining the replay graphs of

$$a$$

and

$$b.$$

Observable computation remains unchanged.
Only the historical dependency graph is extended.

D.10 Small-Step Evaluation

The reflexive transitive closure of

$$\longrightarrow$$

is written

$$\Longrightarrow .$$

Thus

$$(H, t) \Longrightarrow (H', v)$$

denotes a finite sequence of historical reductions terminating in a value.
The accumulated history

$$H'$$

contains every replay event performed during evaluation.

D.11 Big-Step Evaluation

Big-step evaluation is written

$$(H, t) \Downarrow (H', v).$$

The inductive rules are

$$\frac{}{(H, v) \Downarrow (H, v)}$$

for values,
and

$$\frac{(H,t) \to (H_1,u) \qquad (H_1,u) \Downarrow (H_2,v)}{(H,t) \Downarrow (H_2,v)}$$

for recursive evaluation.

Big-step semantics therefore arise directly from repeated small-step replay.

D.12 Historical Determinism

Evaluation is deterministic.

Suppose

$$(H,t) \to (H_1,u_1)$$

and

$$(H,t) \to (H_2,u_2).$$

Then

$$u_1 = u_2$$

and

$$H_1 \simeq_R H_2.$$

Replay therefore reconstructs a unique observable computation.

D.13 Historical Soundness

Operational semantics preserve typing.

If

$$H \vdash t : A$$

and

$$(H,t) \Longrightarrow (H',u),$$

then

$$H' \vdash u : A.$$

Thus replay cannot invalidate previously established typing judgments.

D.14 Operational Interpretation

The operational semantics developed in this appendix illustrate the central difference between the historical calculus and conventional evaluation semantics.

Ordinary operational semantics transform terms.

Historical operational semantics transform histories.

Observable terms evolve only because the underlying constructive history has grown.

Evaluation therefore becomes one particular manifestation of historical construction.

Normalization,
proof replay,
dependency reconstruction,
distributed verification,
incremental compilation,
and
persistent mathematical archives
are all described by the same operational relation.

The following appendix presents the replay algorithm itself in executable language-independent pseudocode and demonstrates that the operational semantics may be implemented using only the primitive event algebra introduced in the kernel specification.

E The Replay Algorithm

The operational semantics of the preceding appendix define the mathematical meaning of replay. This appendix presents the corresponding implementation in a language-independent form. The objective is not to prescribe a particular programming language but to specify the minimal algorithm that every conforming Spherepop kernel must implement.

The replay engine is the computational heart of the historical kernel.

Unlike conventional proof assistants, whose trusted computation consists primarily of normalization and conversion checking, the Spherepop kernel treats replay as the primitive verification mechanism. Every theorem, every proof, every

normalization, and every historical equivalence is ultimately verified by reconstructing the corresponding construction history.

Replay therefore functions simultaneously as
evaluation,
type reconstruction,
dependency reconstruction,
proof verification,
and
historical validation.
The entire trusted kernel is organized around this single algorithm.

E.1 Input and Output

Replay accepts a well-formed history

$$H = e_1; e_2; \cdots ; e_n.$$

The output is the reconstructed kernel state

$$S(H),$$

consisting of
the current typing environment,
the dependency graph,
the collection of replay-normal forms,
the universe assignments,
and
the collection of verified mathematical objects.
Replay never modifies an existing state.
Instead,
each event constructs a new state

$$S_i,$$

from its predecessor

$$S_{i-1}.$$

E.2 Replay Invariant

Throughout execution the replay engine maintains the invariant

$$S_i = \text{Replay}(e_1; \cdots ; e_i).$$

Consequently,
after processing every event,
the reconstructed state is identical to the state originally produced during construction.

Replay therefore establishes deterministic reproducibility.

E.3 Initialization

Replay begins from the empty historical state.

$$S_0 = (\varnothing, \varnothing, \varnothing, \varnothing).$$

No declarations,
no dependency edges,
no replay cache,
and
no historical objects
exist initially.
Every subsequent component is generated constructively.

E.4 Main Replay Loop

Conceptually,
the replay engine performs the following recursive computation.

$$S_{i+1} = \text{Execute}(e_{i+1}, S_i).$$

Thus

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

Execution therefore consists of repeatedly interpreting historical events.

The replay engine itself contains no mathematical knowledge beyond the event algebra.

E.5 Declaration Events

Suppose

$$e = \text{Declare}(x, A).$$

Replay first verifies that

$$A$$

is already a well-formed type.
The variable

$$x$$

is then inserted into the reconstructed typing environment together with its historical identifier.

The dependency graph receives a new isolated vertex corresponding to the declaration event.

E.6 Bind Events

Suppose

$$e = \text{Bind}(a, b).$$

Replay verifies that both constructions already exist.
A dependency edge

$$a \to b$$

is inserted into the reconstructed dependency graph.
Cycle detection is performed immediately.
If a directed cycle appears,
replay terminates with failure.
Consequently,
acyclicity is enforced incrementally rather than globally.

E.7 Reduction Events

Suppose

$$e = \text{Reduce}(t).$$

Replay computes the corresponding reduction step,
verifies preservation,
records the resulting normal form,
and
stores the result inside the replay cache.
Previously normalized historical prefixes are reused whenever available.

E.8 Collapse Events

Suppose

$$e = \text{Collapse}(a, b, p).$$

Replay first reconstructs the equality proof

$$p.$$

Replay equivalence is then verified.
Only after successful verification is the collapse edge inserted into the historical graph.
Failure of replay equivalence causes immediate rejection of the history.

E.9 Meld Events

Suppose

$$e = \mathsf{Meld}(H_1, H_2).$$

Replay independently reconstructs

$$H_1$$

and

$$H_2.$$

Compatibility is verified.

If compatibility succeeds,

their dependency graphs are combined according to the Meld semantics developed earlier.

Otherwise,

replay terminates with failure.

E.10 Universe Verification

Whenever replay encounters a universe declaration,

the cumulative hierarchy is verified immediately.

Every judgment

$$\mathsf{Type}_i : \mathsf{Type}_{i+1}$$

is checked locally.

Attempted construction of

$$\mathsf{Type} : \mathsf{Type}$$

necessarily introduces a cyclic dependency,

causing replay failure.

Universe consistency therefore follows directly from replay.

E.11 Replay Caching

The immutable nature of histories permits aggressive caching.

Suppose replay has already reconstructed

$$H_0.$$

Any subsequent history

$$H = H_0; H'$$

begins replay from the cached state corresponding to

$$H_0.$$

Only the suffix

$$H'$$

requires execution.
Replay therefore scales with historical growth rather than total history size.
This property forms the computational basis of incremental verification.

E.12 Replay Complexity

Let

$$n$$

denote the number of newly introduced historical events.
Suppose replay caches every previously verified prefix.
Then replay complexity satisfies

$$T(n) = O(n),$$

with respect to the number of new events.
Graph traversal,
dependency verification,
and
historical reconstruction
are all linear in the size of the newly introduced historical suffix.
Observable normalization contributes only through newly generated reductions.
Consequently,
verification depends primarily upon historical growth rather than total library size.

E.13 Replay Correctness

The replay algorithm satisfies the following correctness theorem.

Theorem E.1 (Replay Correctness)**.** *Suppose*

$$H$$

is accepted by the replay engine.
Then

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

constructs exactly the kernel state originally produced by

$$H.$$

Furthermore,
every typing judgment reconstructed during replay is derivable in the declarative historical calculus.

The proof proceeds by induction over the event sequence.
Each event preserves the replay invariant established earlier.

E.14 The Central Role of Replay

Replay occupies a uniquely central position within the historical kernel.
Normalization,
type checking,
dependency reconstruction,
proof verification,
incremental recompilation,
distributed synchronization,
historical compression,
and
persistent archival verification
all reduce to replay over immutable constructive histories.
Consequently,
the trusted implementation of Spherepop consists primarily of one deterministic graph reconstruction algorithm together with the primitive event algebra.

The remainder of the language—including elaboration, proof automation, syntax, module systems, tactics, optimizers, and development environments—may all remain outside the trusted kernel.

This separation between historical replay and user-facing language design is one of the principal architectural consequences of adopting histories rather than contexts as the primitive semantic objects of constructive computation.

The next appendix develops the normalization algorithm itself, showing how historical replay naturally supports normalization by evaluation while avoiding repeated reduction of unchanged historical prefixes.

F Normalization by Historical Evaluation

The replay algorithm developed in the previous appendix reconstructs the historical derivation associated with a computation. Replay alone, however, does not determine definitional equality. The kernel must also compare terms that may differ syntactically while representing identical constructions.

Conventional implementations of the Calculus of Constructions frequently employ Normalization by Evaluation (NbE). Rather than repeatedly applying rewriting rules to syntax, NbE evaluates terms into a semantic domain before reconstructing canonical normal forms.

The historical calculus preserves this fundamental idea while enriching the semantic domain with constructive provenance.

Normalization therefore becomes normalization of histories as well as expressions.

We refer to this procedure as *Normalization by Historical Evaluation* (NbHE).

F.1 Motivation

Ordinary normalization answers the question

$$t \Downarrow v.$$

Historical normalization answers a richer question,

$$(H, t) \Downarrow (H', v),$$

where

$$H'$$

contains the replay history responsible for producing the normal form.

Normalization therefore computes two objects simultaneously.
The observable canonical expression.
The constructive history explaining that expression.

F.2 Historical Semantic Domain

Let

$$\mathcal{V}$$

denote the semantic domain of values.
Historical evaluation instead employs

$$\widehat{\mathcal{V}} = \mathcal{V} \times \mathcal{H},$$

where

$$\mathcal{H}$$

is the category of replayable histories introduced earlier.
Every semantic object therefore consists of
an observable value,
its construction history,
its dependency graph,
and
its replay certificate.
Semantic evaluation never discards provenance.

F.3 Evaluation Function

The historical evaluation function is

$$[\![t]\!]_H : Env \to \widehat{\mathcal{V}}.$$

Unlike ordinary semantic evaluation,
the resulting value records the complete replay history responsible for its construction.
Evaluation therefore preserves the invariant

$$\text{History}(\llbracket t \rrbracket_H) = H_t.$$

F.4 Reflection

Neutral terms are reflected into the semantic domain by

$$\uparrow : Ne \to \widehat{\mathcal{V}}.$$

Reflection constructs semantic objects whose replay history initially consists only of the corresponding declaration event.

Future reductions extend this history monotonically.

F.5 Reification

Reification reconstructs canonical syntax.

The historical reification operator is

$$\downarrow : \widehat{\mathcal{V}} \to Nf.$$

Unlike ordinary NbE,

reification preserves a reference to the historical provenance of the resulting normal form.

The kernel therefore produces

$$(\downarrow (v), \text{History}(v))$$

rather than the syntax alone.

F.6 Normalization Procedure

Normalization consists of three conceptual phases.

First,

evaluate the observable term into the historical semantic domain.

Second,

perform replay-aware semantic computation until no further reductions are possible.

Third,

reify the resulting semantic object into canonical syntax while preserving its construction history.

Formally,

$$\mathrm{Norm}(H, t) =\downarrow ([\![t]\!]_H).$$

F.7 Historical Sharing

One of the principal advantages of historical normalization concerns shared constructive prefixes.

Suppose

$$t_1$$

and

$$t_2$$

share the historical prefix

$$H_0.$$

Ordinary normalization often evaluates the common subexpression repeatedly.

Historical normalization instead evaluates

$$H_0$$

once,

records the resulting semantic object,

and

reuses that object whenever replay encounters the same historical prefix.

Consequently,

semantic evaluation depends upon historical novelty rather than syntactic duplication.

F.8 Incremental Normalization

Suppose

$$H' = H; e.$$

Only the newly introduced event

$$e$$

requires semantic evaluation.

Every previously normalized prefix remains valid because histories are immutable.

Incremental normalization therefore satisfies

$$\text{Norm}(H') = \text{Extend}(\text{Norm}(H), e).$$

No previously normalized historical segment requires recomputation.

F.9 Replay-Guided Conversion

Conversion checking reduces to comparison of replay-normal forms.

Suppose

$$A$$

and

$$B$$

are types.

The kernel computes

$$\text{Norm}(H, A)$$

and

$$\text{Norm}(H, B).$$

Definitional equality holds precisely when
their observable normal forms coincide,
their replay histories are replay equivalent,
and
their dependency graphs satisfy historical equivalence.

Observable equality alone is therefore insufficient.
Constructive provenance also contributes to conversion.

F.10 Correctness

Historical normalization satisfies two fundamental correctness properties.
First,
normalization preserves observable semantics.
If

$$(H, t) \Downarrow (H', v),$$

then

$$v$$

is observationally equivalent to

$$t.$$

Second,
normalization preserves constructive provenance.
The replay history associated with

$$v$$

reconstructs the complete sequence of reductions establishing the normal form.
No computational information is discarded.

F.11 Complexity

Let

$$k$$

denote the number of newly introduced historical events.
Suppose all historical prefixes have already been normalized.
Then the complexity of historical normalization satisfies

$$O(k),$$

rather than depending upon the total size of the mathematical library.

This result follows immediately from replay caching and immutable histories.

The normalization algorithm therefore scales with historical growth instead of historical size.

F.12 Comparison with Conventional NbE

Normalization by Evaluation computes canonical semantic representatives of lambda terms.

Normalization by Historical Evaluation computes canonical semantic representatives together with the histories that justify them.

Both algorithms produce identical observable normal forms.

The historical algorithm additionally produces replay certificates,

dependency graphs,

incremental replay caches,

and

persistent constructive provenance.

Consequently,

historical normalization subsumes conventional normalization while providing the additional information required for replay verification, distributed proof construction, incremental compilation, and persistent mathematical archives.

The historical kernel therefore performs normalization only once for each constructive history. Every subsequent verification reuses the normalized historical prefix directly, making normalization an accumulating mathematical resource rather than a computation that must be repeated independently for every proof.

The following appendix develops the complete bidirectional type-checking algorithm, demonstrating how replay-aware normalization integrates with type inference, universe checking, and historical dependency analysis to produce the trusted verification procedure implemented by the Spherepop kernel.

G Bidirectional Historical Type Checking

The declarative typing rules presented earlier provide the mathematical specification of the historical calculus. They are intentionally symmetric and well suited for metatheoretic reasoning. An implementation, however, requires a decision

procedure. This appendix develops the bidirectional type-checking algorithm used by the Spherepop kernel.

Bidirectional type checking separates two complementary computational tasks.

Certain expressions naturally determine their own type.

Other expressions cannot determine a type without additional information and must instead be verified against an expected type.

This distinction substantially reduces the search space of the type checker, eliminates unnecessary normalization, and provides a direct computational interpretation of the historical typing judgments.

Unlike conventional implementations, every phase of the algorithm operates over persistent histories rather than mutable typing contexts.

G.1 Historical Judgments

Two judgments are introduced.

Type synthesis

$$H \vdash t \Rightarrow A$$

computes the type of

$$t.$$

Type checking

$$H \vdash t \Leftarrow A$$

verifies that

$$t$$

inhabits the expected type

$$A.$$

The two judgments are mutually recursive.

G.2 Variables

Variables synthesize their declared type.

$$\frac{\text{Declare}(x : A) \in H}{H \vdash x \Rightarrow A}$$

No normalization is required.

The declaration event uniquely determines both the observable type and its historical provenance.

G.3 Universes

Universes synthesize immediately.

$$\frac{}{H \vdash \mathsf{Type}_i \Rightarrow \mathsf{Type}_{i+1}}$$

Likewise,

$$\frac{}{H \vdash \mathsf{Prop} \Rightarrow \mathsf{Type}_0}$$

G.4 Applications

Applications synthesize.

Suppose

$$H \vdash f \Rightarrow \Pi x : A.B.$$

The argument is then checked against

$$A.$$

The resulting synthesized type is

$$B[x := a].$$

Formally,

$$\frac{H \vdash f \Rightarrow \Pi x : A.B \qquad H \vdash a \Leftarrow A}{H \vdash f\ a \Rightarrow B[x := a]}$$

Historical substitution merges the dependency graphs of

$$f$$

and

$$a.$$

G.5 Lambda Abstractions

Lambda abstractions generally cannot synthesize.

Instead,

they are checked against an expected dependent function type.

$$\frac{H, x : A \vdash t \Leftarrow B}{H \vdash \lambda x.t \Leftarrow \Pi x : A.B}$$

The expected type therefore determines the interpretation of the parameter.

G.6 Dependent Products

Dependent products synthesize universe levels.

Suppose

$$H \vdash A \Rightarrow \mathsf{Type}_i$$

and

$$H, x : A \vdash B \Rightarrow \mathsf{Type}_j.$$

Then

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

Universe inference therefore proceeds recursively.

G.7 Historical Conversion

Suppose

$$H \vdash t \Rightarrow A.$$

To verify

$$t \Leftarrow B,$$

the kernel performs replay-aware normalization.
If

$$\mathrm{Norm}(H, A) \equiv \mathrm{Norm}(H, B),$$

then checking succeeds.
Otherwise,
type checking fails.
Thus conversion reduces to replay normalization rather than syntactic equality.

G.8 Historical Equality

Equality proofs are checked rather than synthesized.
Suppose the expected type is

$$\mathrm{Eq}_A(a, b).$$

Replay reconstructs

$$a, \qquad b,$$

normalizes both constructions,
and verifies replay equivalence.
If replay establishes equivalence,
the proof is accepted.
Otherwise,
checking terminates immediately.

G.9 Historical Replay Integration

Every successful typing judgment contributes replay metadata.
Suppose

$$H \vdash t \Rightarrow A.$$

The type checker additionally records

$$\mathrm{History}(t),$$

the replay-normal form,

the dependency graph,
and
the replay cache identifier.
Subsequent judgments referring to

$$t$$

reuse these structures directly.

Type checking therefore accumulates historical knowledge rather than repeatedly reconstructing it.

G.10 Incremental Checking

Suppose

$$H' = H; e.$$

The historical type checker first determines the dependency region affected by

$$e.$$

Only constructions reachable from

$$e$$

require rechecking.
Every unaffected typing derivation remains valid.
Consequently,
incremental verification is determined entirely by graph reachability.
No global traversal of the proof library is required.

G.11 Termination

Bidirectional historical type checking terminates provided
the universe hierarchy is well founded,
replay terminates,
historical normalization terminates,
and
dependency graphs remain acyclic.

These conditions have already been established by the metatheory.

Termination therefore follows by structural induction over replay histories.

G.12 Correctness

The algorithm satisfies two principal correctness properties.

First,

every synthesized judgment is derivable in the declarative historical calculus.

Second,

every successful checking judgment corresponds to a valid replayable construction.

The implementation is therefore both logically sound and computationally complete with respect to the declarative specification.

G.13 Complexity

Let

$$n$$

denote the number of newly introduced historical events.

Suppose historical normalization and replay caches are already available.

Then synthesis,

checking,

conversion,

and

dependency analysis

are all bounded by the size of the newly introduced replay region.

Consequently,

the overall complexity of type checking satisfies

$$O(n),$$

with respect to historical growth.

Unlike conventional implementations, the complexity is independent of the total size of the accumulated mathematical archive.

G.14 The Historical Verification Procedure

The bidirectional algorithm completes the trusted verification pipeline.

Parsing produces abstract syntax.

Elaboration produces historical kernel terms.

Replay reconstructs constructive provenance.

Historical normalization computes canonical replay forms.

Bidirectional checking verifies every typing judgment.

Universe inference establishes consistency.

Finally,

the verified history is committed to the persistent mathematical archive.

Every theorem accepted by the kernel has therefore been verified both as an observable proof term and as a replayable constructive history.

The next appendix develops the historical universe inference algorithm in full detail, showing how cumulative universes, replay normalization, and dependency analysis cooperate to guarantee logical consistency while minimizing explicit universe annotations in user programs.

H Historical Universe Inference

The cumulative universe hierarchy introduced in the core calculus guarantees the logical consistency of the historical kernel. Requiring programmers to annotate every universe level explicitly, however, would make practical programming extremely cumbersome. A usable implementation therefore requires an inference procedure capable of reconstructing the majority of universe assignments automatically.

This appendix develops the historical universe inference algorithm. The algorithm extends conventional universe inference by incorporating replay, historical dependency analysis, and immutable construction histories into the constraint solving process.

Rather than assigning universe levels solely from syntactic structure, the historical kernel derives them from the constructive history responsible for each declaration.

H.1 Universe Variables

During elaboration, explicit universe levels are replaced by fresh metavariables

$$\alpha, \beta, \gamma, \ldots$$

representing unknown universe indices.

For example,

$$\Pi x : A.B$$

is initially assigned

$$\mathsf{Type}_{\alpha},$$

where

$$\alpha$$

remains unresolved until sufficient constraints have been accumulated.

Universe inference therefore proceeds by solving constraints rather than by immediate assignment.

H.2 Constraint Generation

Every typing rule contributes universe constraints.

Suppose

$$H \vdash A : \mathsf{Type}_i$$

and

$$H, x : A \vdash B : \mathsf{Type}_j.$$

Formation of the dependent product generates

$$k \geq i,$$

$$k \geq j,$$

together with

$$\Pi x : A.B : \mathsf{Type}_k.$$

The minimal admissible value of

$$k$$

is therefore

$$\max(i, j).$$

The inference engine accumulates these inequalities throughout elaboration.

H.3 Historical Constraints

The historical interpretation contributes additional information unavailable in ordinary type theory.

Suppose two declarations share the replay history

$$H_0.$$

Their inferred universe assignments may also share the corresponding replay certificate.

Consequently,
identical historical prefixes need not regenerate identical universe constraints.
Replay caching therefore applies equally to universe inference.

H.4 Constraint Graph

Rather than storing inequalities independently,
the kernel constructs a directed graph

$$G_U.$$

Vertices represent universe variables.
Edges represent ordering constraints,
for example,

$$\alpha \to \beta$$

whenever

$$\beta \geq \alpha.$$

Constraint solving therefore becomes graph propagation rather than repeated symbolic substitution.

H.5 Propagation

Universe propagation proceeds by repeatedly applying

$$\alpha \leq \beta, \qquad \beta \leq \gamma \Longrightarrow \alpha \leq \gamma.$$

Propagation terminates because
the constraint graph is finite,
histories are finite,
and
cycles corresponding to

$$\mathsf{Type} : \mathsf{Type}$$

are prohibited.

H.6 Replay Consistency

Suppose replay reconstructs a previously verified declaration.
Its universe assignment is reconstructed simultaneously.
Replay therefore guarantees

$$\mathrm{Replay}(H) \Longrightarrow U(H),$$

where

$$U(H)$$

denotes the historical universe environment.
Universe inference is therefore deterministic with respect to replay.
The same history always reconstructs the same hierarchy.

H.7 Minimal Assignment

After propagation,
every universe variable possesses a lower bound.

The kernel assigns the smallest admissible universe satisfying all accumulated constraints.

Formally,

if

$$L(\alpha)$$

denotes the lower bound of

$$\alpha,$$

then

$$\alpha = L(\alpha).$$

The resulting hierarchy is therefore minimal.

No declaration is placed into a larger universe than required by replay.

H.8 Historical Universe Caching

Universe assignments become immutable historical objects.

Suppose

$$H' = H; e.$$

Only constraints introduced by

$$e$$

require solution.

Previously reconstructed universe assignments remain valid because the histories that produced them are immutable.

Universe inference therefore scales with historical extension rather than with the total size of the archive.

H.9 Universe Soundness

The inference procedure satisfies the following theorem.

Theorem H.1 (Historical Universe Soundness)**.** *Suppose universe inference assigns*

$$\mathsf{Type}_i$$

to every declaration contained in

$$H.$$

Then every typing judgment reconstructed by replay satisfies the cumulative universe rules of the declarative calculus.

The proof proceeds by induction over replay.

Every replay event preserves the accumulated universe constraints established by earlier events.

H.10 Universe Completeness

Conversely,

suppose a declaration admits a valid cumulative universe assignment.

Then the inference procedure reconstructs one.

Theorem H.2 (Historical Universe Completeness)**.** *Every replayable historical construction satisfying the declarative universe rules admits a minimal assignment generated by the historical inference algorithm.*

Thus the implementation is complete with respect to the declarative specification.

H.11 Historical Predicativity

One particularly interesting consequence of the historical interpretation concerns predicativity.

Traditional presentations forbid

$$\mathsf{Type} : \mathsf{Type}$$

as an axiom.

The historical calculus instead interprets this prohibition operationally.

Attempting to construct

$$\mathsf{Type} : \mathsf{Type}$$

necessarily introduces a dependency cycle into the replay graph.

Such cycles violate historical admissibility.

Predicativity therefore becomes a theorem concerning constructive history rather than an externally imposed restriction.

H.12 Complexity

Let

$$m$$

denote the number of newly introduced universe constraints.

Propagation over the finite constraint graph requires

$$O(m)$$

time when replay caches are available.

Consequently,

historical universe inference scales linearly with the number of newly generated constraints rather than with the size of the complete mathematical development.

H.13 Historical Interpretation

Universe inference completes the implementation of the trusted kernel.

Replay reconstructs constructive history.

Normalization computes replay-normal forms.

Bidirectional checking verifies typing.

Universe inference reconstructs the cumulative hierarchy.

Together these algorithms determine every logical judgment accepted by the kernel.

The historical interpretation reveals that universe levels are not merely annotations upon types.

They are themselves historical constructions whose values are determined by the growth of constructive knowledge.

The hierarchy therefore evolves through the same replay mechanism governing proofs, computations, and dependency graphs.

The following appendix develops the complete complexity analysis of the historical kernel, comparing replay-based verification with conventional normalization-

based proof kernels and identifying the asymptotic consequences of persistent immutable histories.

I Complexity Analysis of the Historical Kernel

The preceding appendices have presented the algorithms constituting the trusted Spherepop kernel. The present appendix analyzes their computational complexity. The objective is not merely to establish asymptotic bounds, but to understand how the historical representation alters the computational structure of proof verification relative to conventional normalization-based kernels.

A central distinction throughout this appendix is between *cold verification*, in which an entire mathematical archive is reconstructed from the empty history, and *warm verification*, in which a previously verified historical archive already exists and only newly appended constructions require verification.

Because the historical calculus is append-only, these two situations exhibit substantially different computational behavior.

I.1 Complexity Measures

Throughout this appendix we write

$$N$$

for the total number of historical events contained in a mathematical archive,

$$M$$

for the number of newly appended events,

$$V$$

for the number of historical vertices,

$$E$$

for the number of dependency edges,
and

$$R(H)$$

for the reachable dependency subgraph of a history

$$H.$$

Unlike conventional analyses, the principal quantity is rarely

$$N.$$

Most verification procedures depend instead upon

$$|R(H)|$$

or

$$M.$$

I.2 Cold Verification

Suppose a completely empty kernel reconstructs a mathematical archive containing

$$N$$

historical events.
Replay processes every event exactly once.
Every dependency edge is traversed exactly once.
Assuming bounded-time primitive event execution, replay satisfies

$$T_{\text{cold}} = O(N + E).$$

Since well-designed mathematical developments typically satisfy

$$E = O(N),$$

cold verification becomes linear in archive size.

Unlike repeated normalization strategies, no previously reconstructed construction is recomputed.

I.3 Warm Verification

The more important case concerns incremental development.

Suppose the existing archive has already been verified.

A new contribution introduces

$$M$$

historical events.

Replay begins from the previously reconstructed state.

Only the new historical suffix requires execution.

Consequently,

$$T_{\text{warm}} = O(M + E_M),$$

where

$$E_M$$

denotes only those dependency edges introduced by the new history.

The computational cost therefore depends upon mathematical growth rather than library size.

This distinction becomes increasingly significant as the archive expands.

I.4 Reachability-Limited Verification

Suppose an existing theorem is modified.

Traditional proof kernels frequently require normalization of every theorem depending, directly or indirectly, upon the modified declaration.

The historical kernel instead computes the reachable dependency graph

$$R(H).$$

Only constructions contained within

$$R(H)$$

require replay.

Verification therefore satisfies

$$T = O(|R(H)|).$$

When

$$|R(H)| \ll N,$$

large mathematical libraries may be updated while replaying only a very small fraction of previously verified mathematics.

I.5 Normalization Complexity

Historical normalization inherits the complexity established in the previous appendix.

Suppose every previously encountered historical prefix has already been normalized.

Only newly introduced reductions require evaluation.

Thus

$$T_{\text{Norm}} = O(M).$$

The normalization cost depends only upon the newly constructed history.

Repeated verification of unchanged mathematics performs no additional normalization.

I.6 Historical Memoization

Persistent histories admit perfect memoization.

Suppose

$$H$$

has already been replayed.

Since histories are immutable,

every subsequent replay of

$$H$$

returns the cached reconstruction immediately.

Consequently,

replay of previously verified histories approaches constant time,

excluding retrieval costs.

Memoization therefore improves over time as the mathematical archive grows.

Unlike mutable systems,
cache invalidation is determined entirely by historical reachability.

I.7 Parallel Replay

Suppose

$$H_1, H_2, \ldots, H_k$$

are pairwise independent.
Replay proceeds concurrently,
yielding

$$T_{\text{parallel}} = O\left(\max_i |H_i|\right),$$

ignoring synchronization overhead.

Parallelism therefore follows directly from the dependency graph rather than requiring speculative scheduling.

The kernel exposes concurrency naturally through historical independence.

I.8 Storage Complexity

Every historical event is stored exactly once.

Shared replay prefixes become shared subgraphs.

Suppose two histories

$$H_1 = H_0; H_A$$

and

$$H_2 = H_0; H_B.$$

Only

$$H_A$$

and

$$H_B$$

require additional storage.

Consequently,

storage grows with genuinely new mathematical construction rather than with the number of proofs referencing that construction.

The historical archive therefore approaches the minimal directed acyclic graph representing the accumulated constructive knowledge.

I.9 Comparison with Conventional Proof Kernels

The principal computational distinction between the historical kernel and a traditional normalization-based kernel concerns the unit of computation.

In conventional systems,

the fundamental object is the term.

Verification repeatedly computes normal forms of terms.

In the historical kernel,

the fundamental object is the immutable construction history.

Replay reconstructs histories,

normalization applies only to newly introduced historical regions,

and

verification follows dependency graphs rather than syntactic structure.

Consequently,

many computations that must be repeated in conventional implementations become persistent mathematical objects within the historical archive itself.

I.10 Historical Amortization

Perhaps the most significant computational consequence concerns amortization.

Every replay,

every normalization,

every dependency reconstruction,

every universe assignment,

and

every replay certificate

becomes part of the persistent historical archive.

Future verification therefore benefits from every previous verification.

The mathematical archive accumulates computational work over time.

The larger the verified corpus becomes,

the greater the proportion of future verification that may be satisfied by reusing existing replay results.

Verification therefore becomes progressively less expensive on a per-theorem basis.

I.11 Complexity Summary

The historical calculus changes the asymptotic behavior of theorem verification by replacing repeated reconstruction with persistent constructive memory.

Cold verification remains linear in the size of the mathematical archive, matching the expected behavior of efficient proof kernels.

Warm verification scales with historical extension rather than archive size.

Normalization scales with newly introduced reductions.

Dependency analysis scales with graph reachability.

Parallel replay scales with independent historical branches.

Storage scales with genuinely new mathematical construction.

The resulting implementation is therefore naturally suited to persistent, incrementally evolving mathematical libraries, collaborative theorem proving, and distributed formal verification.

The next appendix presents the reference implementation of the trusted kernel in language-independent pseudocode. Every algorithm developed throughout the paper is assembled into a single executable specification demonstrating that the historical calculus may be realized by a surprisingly small collection of interacting procedures built upon immutable constructive histories.

J Reference Kernel Pseudocode

The preceding appendices have presented the mathematical specification of the historical kernel together with its operational semantics, replay algorithm, normalization procedure, type checker, universe inference algorithm, and complexity analysis. This appendix assembles these components into a unified reference implementation written in language-independent pseudocode.

The objective is not to optimize execution for any particular architecture.

Instead,

the algorithms presented here define the computational behavior required of any conforming implementation.

Different implementations may choose different internal data structures,
parallel execution strategies,
or
persistent storage mechanisms,
provided the observable replay semantics remain identical.

J.1 Kernel State

The trusted kernel maintains a single immutable state

$$S.$$

Conceptually,

$$S = (H, \Gamma, D, U, C),$$

where

$$H$$

is the historical event sequence,

$$\Gamma$$

is the reconstructed typing environment,

$$D$$

is the dependency graph,

$$U$$

is the universe environment,
and

$$C$$

is the replay cache.

Every kernel operation returns a new state rather than modifying an existing one.

J.2 Kernel Entry Point

The principal verification routine accepts a candidate history

$$H_{\text{new}}.$$

Conceptually,
verification proceeds according to

$$\text{Verify} : S \times H_{\text{new}} \to S'.$$

The algorithm performs
history validation,
replay,
normalization,
type checking,
universe inference,
dependency reconstruction,
and
cache extension.
Only after every phase succeeds is the resulting state committed.

J.3 Replay Procedure

Replay forms the computational center of the implementation.
The algorithm may be described abstractly as follows.

```
Replay(history):

    state ← EmptyState()

    for event in history:

        state ← Execute(event, state)

    return state
```

The procedure is deterministic.
Every replay of the same history reconstructs precisely the same kernel state.

J.4 Event Execution

Each primitive event possesses a corresponding execution procedure.

Conceptually,

```
Execute(event, state):

    switch event.kind

        Declare:
            return Declare(state,event)

        Bind:
            return Bind(state,event)

        Pop:
            return Pop(state,event)

        Refuse:
            return Refuse(state,event)

        Collapse:
            return Collapse(state,event)

        Meld:
            return Meld(state,event)

        Reduce:
            return Reduce(state,event)

        Replay:
            return Replay(event.history)
```

Every observable computation ultimately reduces to repeated execution of these primitive operations.

J.5 Normalization

Historical normalization follows the algorithm developed previously.

```
Normalize(term,state):

    semantic ← Evaluate(term,state)

    return Reify(semantic)
```

Evaluation preserves historical provenance.
Reification reconstructs canonical syntax together with replay metadata.

J.6 Bidirectional Type Checking

The implementation follows the mutually recursive synthesis and checking procedures.

```
Infer(term,state):

    match term

        Variable:
            return Lookup(term,state)

        Application:
            ...

        Product:
            ...

        Universe:
            ...

        otherwise:
            fail
```

Checking is defined similarly.

```
Check(term,type,state):

    inferred ← Infer(term,state)
```

```
    inferred ← Normalize(inferred,state)

    expected ← Normalize(type,state)

    return ReplayEquivalent(inferred,expected)
```

Normalization therefore appears only at conversion boundaries.

J.7 Universe Solver

Universe inference constructs a constraint graph.

Conceptually,

```
SolveUniverses(graph):

    repeat

        propagate constraints

    until fixed point

    return minimal assignment
```

Replay guarantees that every previously solved historical prefix remains valid.

J.8 Dependency Maintenance

Dependency reconstruction proceeds incrementally.

```
AddDependency(parent,child,state):

    if CreatesCycle(parent,child):

        fail

    insert edge

    return updated state
```

Cycle detection therefore occurs locally rather than globally.

J.9 Replay Cache

Replay results are cached by historical identifier.

```
LookupReplay(history):

    if history in cache:

        return cache[history]

    result ← Replay(history)

    cache[history] ← result

    return result
```

Because histories are immutable,
cache entries never require modification.

J.10 Incremental Verification

Suppose a new historical suffix

$$H'$$

extends an existing verified archive.
The implementation performs

```
IncrementalVerify(H'):

    prefix ← LongestCachedPrefix(H')

    state ← LookupReplay(prefix)

    replay suffix

    verify

    commit
```

Only newly introduced events require replay.
Previously verified histories remain untouched.

J.11 Commit

Successful verification extends the persistent archive.

```
Commit(state):

    append history

    extend dependency graph

    extend replay cache

    extend universe environment

    return persistent archive
```

No existing mathematical object is modified.
The archive grows monotonically.

J.12 Trusted Computing Base

The reference implementation reveals that remarkably little code belongs inside the trusted kernel.

Only
the event interpreter,
the replay engine,
historical normalization,
bidirectional type checking,
dependency validation,
and
universe inference
must be trusted.
Everything else—
surface syntax,
interactive tactics,

automation,
proof search,
compiler optimizations,
integrated development environments,
module systems,
package managers,
and
editor tooling—
may be implemented entirely outside the trusted computing base.

Their correctness is established only after replay reconstructs their output inside the kernel.

J.13 Kernel Simplicity

Perhaps the most striking consequence of the historical architecture is that the entire trusted implementation reduces to a surprisingly small collection of deterministic algorithms operating upon immutable histories.

The expressive richness of the language does not arise from a complicated kernel.

Instead,

it emerges from repeated composition of a small event algebra together with replay over persistent constructive histories.

This economy of implementation mirrors the mathematical economy established throughout the paper.

A minimal collection of primitive operations generates a remarkably expressive foundation for computation, proof, and constructive mathematics.

The remaining appendix presents a denotational semantics for the historical kernel using categories with families and indexed historical fibrations. That semantic model demonstrates that the operational algorithms developed throughout this specification admit a mathematically precise interpretation independent of any particular implementation strategy.

K Denotational Semantics

The operational semantics developed throughout this specification describe how the historical kernel evaluates computations and verifies proofs. Operational

semantics, however, explain only how computation proceeds. They do not by themselves explain what computations mean.

The purpose of this appendix is to provide a denotational semantics for the historical calculus. Rather than interpreting terms as ordinary set-theoretic functions, the historical interpretation assigns semantic meaning to entire construction histories. Programs, proofs, universes, and dependent types are therefore interpreted as objects inside a category of constructive histories.

The resulting semantics demonstrate that replay is not merely an implementation technique. It possesses an intrinsic mathematical interpretation independent of the operational algorithms used by the kernel.

K.1 The Semantic Category

Let

$$\mathcal{H}$$

denote the category introduced in Chapter 15.
Objects are replayable constructive histories

$$H_1, H_2, \ldots$$

while morphisms are replay-preserving historical transformations.
Composition is ordinary composition of replay morphisms,

$$g \circ f.$$

Identity morphisms are given by replay without additional construction.
Thus

$$(\mathcal{H}, \circ, \mathrm{id})$$

forms a category.

K.2 Semantic Contexts

The historical calculus replaces ordinary typing contexts with histories.
Accordingly,
a semantic context is interpreted by

$$[\![H]\!] \in \mathrm{Ob}(\mathcal{H}).$$

Context extension corresponds to historical extension,

$$[\![H; e]\!] = [\![H]\!] \oplus [\![e]\!],$$

where

$$\oplus$$

denotes admissible historical growth.
Unlike ordinary context extension,
historical extension preserves temporal ordering.

K.3 Semantic Types

Every type

$$A$$

is interpreted as a family of admissible constructions over a history,

$$[\![A]\!] : [\![H]\!] \to \mathcal{U},$$

where

$$\mathcal{U}$$

denotes the semantic universe.

Dependent types therefore become fibrations over histories rather than families over ordinary contexts.

K.4 Terms

Suppose

$$H \vdash t : A.$$

Its denotation is

$$[\![t]\!] : [\![H]\!] \to [\![A]\!].$$

Every semantic term therefore preserves historical provenance.

Evaluation is interpreted as composition of replay morphisms rather than rewriting of syntax.

K.5 Dependent Products

The dependent product

$$\Pi x : A.B$$

is interpreted by the right adjoint to historical substitution.

Consequently,

$$[\![\Pi x : A.B]\!]$$

classifies replay-preserving historical functions whose outputs depend upon their constructive inputs.

Thus dependent products retain their usual categorical interpretation while being indexed by histories rather than ordinary contexts.

K.6 Dependent Sums

Dependent sums

$$\Sigma x : A.B$$

are interpreted by historical fibrations.

Every semantic pair consists of
an observable construction,
together with
its dependent historical continuation.

The semantic object therefore records both value and provenance.

K.7 Replay Morphisms

Replay is interpreted as an endofunctor

$$\mathcal{R} : \mathcal{H} \to \mathcal{H}.$$

For every history

$$H,$$

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

For every replay-preserving morphism

$$f,$$

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

Replay therefore acts as an identity-on-objects reconstruction functor.

Operational replay corresponds precisely to semantic reconstruction.

K.8 Collapse

Collapse is interpreted as a coequalizer.

Suppose

$$f, g : H_1 \to H_2.$$

If replay establishes

$$f \simeq g,$$

then

$$\mathsf{Collapse}(f, g)$$

constructs the universal morphism identifying the two histories while preserving their provenance.

Unlike classical quotients,

the original morphisms remain present in the semantic category.

K.9 Meld

Suppose

$$H_0$$

is a shared historical prefix.
The diagram

$$H_1 \leftarrow H_0 \rightarrow H_2$$

admits a pushout whenever compatibility holds.
The semantic interpretation of

$$\mathsf{Meld}(H_1, H_2)$$

is precisely this pushout.
Consequently,
parallel constructive development possesses a universal characterization.

K.10 Historical Fibrations

Dependent types naturally organize themselves into an indexed category

$$\pi : \mathcal{E} \rightarrow \mathcal{H}.$$

The fiber above

$$H$$

contains precisely those constructions admissible after replay of

$$H.$$

Historical extension therefore induces re-indexing functors between fibers.
The resulting structure resembles a category with families while making historical provenance explicit.

K.11 Comprehension

Every dependent type determines a comprehension object.

Suppose

$$A$$

is interpreted over

$$H.$$

Then the comprehension object

$$(H, A)$$

corresponds to extending the history by an admissible declaration event.
Comprehension therefore becomes historical growth.
Rather than adding assumptions to a context,
the semantics add irreversible construction events.

K.12 Soundness

The denotational semantics satisfy the following fundamental theorem.

Theorem K.1 (Semantic Soundness)**.** *Suppose*

$$H \vdash t : A.$$

Then

$$[\![t]\!] \in [\![A]\!].$$

Furthermore,
every replay step preserves denotational meaning.

Operational evaluation and semantic interpretation therefore coincide.

K.13 Completeness

Conversely,

every semantic construction generated by the historical category is represented by a replayable kernel derivation.

Thus the operational and denotational semantics coincide on all well-formed historical constructions.

K.14 Historical Interpretation

The denotational semantics complete the mathematical development of the historical calculus.

The operational semantics explain how histories evolve.

The metatheory explains why replay is logically sound.

The implementation demonstrates how replay is realized computationally.

The denotational semantics explain what replay means independently of any particular implementation.

Together these four perspectives—
syntax,
operations,
algorithms,
and
semantics—
provide a mathematically complete specification of the Spherepop kernel.

The historical interpretation therefore extends beyond implementation strategy. It defines a coherent semantic foundation in which constructive histories serve as the primitive mathematical objects from which computation, proof, dependent types, equality, and categorical structure all emerge naturally. This completes the formal specification of the historical kernel underlying Spherepop.

References

[1] H. P. Barendregt. Lambda Calculi with Types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (eds.), *Handbook of Logic in Computer Science*, Vol. 2. Oxford University Press, 1992.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[3] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[4] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[5] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[6] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[7] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[8] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Second Edition, 2016.

[9] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989.

[10] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1986.

[11] Steve Awodey. *Category Theory*. Oxford University Press, Second Edition, 2010.

[12] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, Second Edition, 1998.

[13] Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999.

[14] Peter Dybjer. Internal Type Theory. In S. Berardi and M. Coppo (eds.), *Types for Proofs and Programs*. Springer, 1996.

[15] Martin Hofmann. Syntax and Semantics of Dependent Types. In A. Pitts and P. Dybjer (eds.), *Semantics and Logics of Computation*. Cambridge University Press, 1997.

[16] Thorsten Altenkirch and Ambrus Kaposi. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of POPL*, 2016.

[17] Andreas Abel. Normalization by Evaluation: Dependent Types and Impredicativity. In *Proceedings of the 18th International Conference on Functional Programming*, 2013.

[18] Ulrich Berger and Helmut Schwichtenberg. An Inverse of the Evaluation Functional for Typed Lambda Calculus. In *Proceedings of LICS*, 1991.

[19] Thierry Coquand, Simon Huber, and Anders Mörtberg. On Higher Inductive Types in Cubical Type Theory. In *Proceedings of LICS*, 2018.

[20] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[21] Gérard Huet. A Unification Algorithm for Typed Lambda Calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[22] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[23] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[24] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[25] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[26] Martin Fowler. Event Sourcing. martinfowler.com, 2017.

[27] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[28] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[29] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Aarhus University Technical Report DAIMI FN-19, 1981.

[30] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[31] Roy L. Crole. *Categories for Types*. Cambridge University Press, 1993.