Spherepop

Histories, Continuations, and the Algebra

of Reachable Computation

Toward a Complete Theory: History-Primary Computation,

Type Systems, Error Correction, and Garbage Collection

Flyxion

Independent Researcher

Abstract

Most programming languages begin with values, variables, and state transitions. Spherepop begins elsewhere: with the claim that histories are ontologically prior to states, and that computation is the progressive restriction of possibility rather than the manipulation of values.

This monograph presents the complete theory of the Spherepop programming language and its philosophical foundations. We proceed from two converging arguments. The first is a theory of notes: the observation that notes are not passive records of the past but anti-collapse artifacts that preserve access to continuations. The second is the Spherepop inversion: the observation that state is a collapsed history and that taking this seriously demands a new computational ontology.

These arguments converge because they are the same argument. A note is an externalized continuation. A Spherepop history is an internalized note. Civilization builds note infrastructure; Spherepop builds note infrastructure for computation.

The monograph develops this convergence through nine parts. Part I establishes the dual failure of the passive theory of notes and the state-centric theory of computation. Part II derives the four Spherepop operators—Pop, Refuse, Bind, and Collapse—from first principles as the minimal complete set for managing possibility space. Part III develops the full operational semantics, the History category, and collapse as quotient. Part IV presents the complete admissibility type system with Progress, Preservation, and Collapse Soundness theorems. Part V derives the seven note classes as physical realizations of the four operators, establishing that notes are Spherepop programs and Spherepop programs are notes. Part VI covers compilation, the Event IR, history equivalence as compiler correctness, garbage collection as admissibility-guided collapse, and error correction as certified refusal. Part VII develops the deep mathematical structure: sheaf semantics, CLIO projections, the Santoro–Waghmare–Panaretos unification, and active geodesic inference. Part VIII situates the frame-

work within civilization-scale note infrastructure, MEM|8, and Repair Theory. Part IX collects open problems.

Four principal theorems anchor the work: the Conservation of Possibility, the Ephemerality Inversion, the Civilizational Collapse Theorem, and the History–Observation Adjunction. Four implementation discoveries retroactively sharpened the theory: refusal must carry reasons; collapse must specify quotient rules; type checking depends on the assumed boundary of the world; and compiler correctness requires history equivalence, not mere output equivalence.

> *This manuscript is not presented as a finished formalization. It is a staged research program. Some results are proved fully, some are proof sketches, and some are conjectural bridges to be developed through implementation and subsequent work. The goal is to make the program precise enough to be falsifiable, not to claim it complete.*

Contents

Part I

The Problem

Chapter 1

The Passive Theory of Notes and Its Failure

There is a theory of notes so pervasive that it rarely receives explicit statement. On this theory, a note is a device for compensating the limitations of biological memory. Something happens—a meeting is scheduled, a formula is derived, a perception is formed—and because the mind cannot reliably retain it, the note is made. The note is therefore a prosthetic memory: it externalizes information onto a more durable substrate, where it may be retrieved when biological recall fails.

This is the passive theory of notes. It places the note in a retrospective relation to time. The note exists because something happened in the past that is worth preserving. Its value is proportional to the fidelity with which it reproduces a prior state: a good note accurately records what occurred; a poor note distorts it; an ephemeral note is low-value because it preserves little and does not endure.

The passive theory is not entirely wrong. Many notes are made for the reason it describes. But as a general account of what notes are and why they matter, it fails on its own terms and in a direction that reveals something important about cognition, language, and civilization.

The failure becomes apparent the moment one examines not celebrated notes but common ones. A grocery list does not record anything that happened. The items on the list do not exist yet in the form the list anticipates: they have not been purchased, assembled, or carried home. The list records a future act. A blueprint describes a building not yet constructed. Source code records computations not yet run. A theorem records consequences not yet derived. A calendar records commitments not yet honored. A playlist records a future act of listening. A bookmark records a future act of reading.

None of these are memories. All of them are notes.

The passive theory cannot accommodate this class of cases without becom-

ing incoherent. It might be extended to say that a note records an *intention* rather than an event, but this extension merely postpones the difficulty. The deeper problem is that the passive theory places every note in a backward-looking relation to time. Even when the recorded content is a future intention, the note is still conceived as a trace of a prior mental event. The orientation is always retrospective.

This gets the phenomenology of note-taking almost exactly wrong. When one makes a to-do list, one is not primarily recording a past intention; one is constructing a future. The note is not oriented toward what was; it is oriented toward what has not yet occurred and what might not occur without the note's assistance.

The Continuation Account

A more adequate theory begins from a distinction that theoretical computer science has found indispensable: the difference between a *state* and a *continuation*. A state is a snapshot of a system at a moment in time—a summary, a compression, a projection. A continuation is what comes next: the future execution enabled by a present configuration, the program of further development that a state makes possible.

Most theories of information focus on states. A database stores states. A photograph is described as capturing a state. A memory is treated as the retention of a past state. But what makes any stored state useful is not the state itself. It is the continuation that state makes accessible.

Consider a Git commit. The commit is not valuable because it stores bytes. It is valuable because it preserves a path through development space—a specific trajectory through the possibility space of a codebase that would otherwise require expensive reconstruction. The stored state is instrumentally useful only because it is a node in a navigable graph of continuations.

Definition 1.1 (Note)**.** Let $\mathcal{C}$ denote a space of continuations and let A be an agent. A *note* is any artifact N such that there exists a continuation $c \in \mathcal{C}$ for which

$$P_A(c, t \mid N) > P_A(c, t),$$

where $P_A(c, t)$ denotes the probability that agent A can successfully reconstruct or traverse continuation c at future time t.

This definition is deliberately broad. It immediately encompasses bookmarks,

maps, photographs, source code, language, and libraries without privileging any medium. What unifies them is the functional condition: a note increases the reachability of at least one continuation.

Definition 1.2 (Continuation Volume)**.** The *continuation volume* of a note N is

$$V_C(N) = \big|\{c \in \mathcal{C} : P_A(c, t \mid N) > P_A(c, t)\}\big|.$$

Definition 1.3 (Reachability Leverage)**.** The *reachability leverage* of a note N is

$$\Lambda(N) = \frac{\sum_c \big[C_r(c) - C_r(c \mid N)\big]}{C_{\text{create}}(N)},$$

where $C_r(c)$ is the cost of reconstructing continuation c from first principles and $C_r(c \mid N)$ is the cost given access to N.

Reachability leverage varies over many orders of magnitude. A ten-second bookmark may save hours of future search. A one-line equation may preserve centuries of accumulated derivation. The leverage analysis predicts which notes are most worth making and preserving, entirely independently of their fidelity to past states.

Notes as Anti-Collapse Artifacts

The Spherepop framework, developed in subsequent chapters, holds that histories are primary and states are their compressed residues. A state s is a projection

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

that collapses many histories $h \in \mathcal{H}$ into a single observable state, producing what the framework calls the *state illusion*: the appearance that the present configuration is self-sufficient, when it is in fact the endpoint of a history whose fibers have been discarded.

A note is, in this framework, an anti-collapse artifact: an artifact that preserves selected fibers of $\mathcal{H}$ against the projection σ.

Proposition 1.4 (Notes as Anti-Collapse Artifacts)**.** *A note N is an anti-collapse artifact if it preserves selected fibers of $\mathcal{H}$ such that a future agent can partially reconstruct elements of $\mathcal{H}$ rather than being restricted to $\sigma(\mathcal{H})$.*

This is the precise sense in which the passive theory inverts the correct account. The passive theory holds that a note preserves the past against forgetting. The correct account holds that a note preserves access to futures against collapse.

The distinction is not merely philosophical: it determines what kinds of notes matter, how notes should be evaluated, and what it means for a note to succeed or fail.

Chapter 2

The State-Centric Assumption and Its Consequences

The Standard Premise

Open any introductory programming text and you encounter a first lesson: a variable is a named location that holds a value. A program begins in some initial state, instructions modify that state, and the final state is the answer.

Listing 2.1: The standard state-centric view

```
x = 5
y = x + 1
```

This appears natural. It mirrors how we naively think about physical processes: an object has properties; those properties change; the current properties constitute the object's state. But the appearance of naturalness is produced by a prior choice—the choice to treat present configuration as the fundamental unit of description.

The State Illusion

Consider what this choice erases. In version control, the repository is not merely its current file tree; it is the entire sequence of commits that produced it. In event sourcing, database records are append-only event logs; the current state is a fold over that log. In scientific experiment, a measurement is the terminus of a sequence of decisions, calibrations, and interpretive choices. In evolutionary biology, a genome is a compressed record of selective pressures.

In each case, the historical structure is epistemically and causally prior to the present configuration. State is the summary. History is the substance.

Definition 2.1 (State Degeneracy)**.** Let E be an event alphabet, $\mathcal{H} = \bigcup_{n=0}^{\infty} E^n$ the set of all finite histories, and S an observable state space. A *state projection* is

a function $\sigma : \mathcal{H} \to S$. The *degeneracy* of a state $s \in S$ is

$$D(s) = |\sigma^{-1}(s)|.$$

Proposition 2.2 (The State Illusion)**.** *For any nontrivial finite state space S and any non-injective projection $\sigma : \mathcal{H} \to S$, there exist distinct histories $H_1 \neq H_2$ such that $\sigma(H_1) = \sigma(H_2)$. Consequently,*

$$\sigma(H_1) = \sigma(H_2) \not\Rightarrow H_1 = H_2.$$

Proof. Since $|\mathcal{H}|$ is countably infinite and $|S|$ is finite, σ cannot be injective. Therefore $D(s) > 1$ for at least one $s \in S$, which yields the required $H_1 \neq H_2$ with $\sigma(H_1) = \sigma(H_2)$. □

Remark 2.3. Proposition 2.2 is the formal statement of the state illusion: any program that exposes only $\sigma(H)$ to its user has silently discarded distinguishing information about the history H. The conventional debugger is an instrument for reconstructing H from $\sigma(H)$ after information has already been lost.

Consequences for Language Design

Programming languages that treat state as primary inherit the degeneracy of Proposition 2.2 as a structural feature. Race conditions arise because σ is non-injective on interleaved histories. Use-after-free vulnerabilities arise because σ conflates histories that differ only in deallocation events. The auditing and provenance tools that practitioners add to production systems are, in each case, partial reconstructions of H from $\sigma(H)$.

The Spherepop design decision follows directly: if history is what is lost, history must be what is preserved.

The Connection to Notes

The state illusion and the passive theory of notes are the same mistake viewed from different angles.

The passive theory treats a note as a record of a past state. The state-centric theory of computation treats a program's output as its definitive product. Both identify value with a terminal state rather than with the trajectory that produced it. Both discard the history in favor of the summary.

The Spherepop framework and the continuation theory of notes are therefore the same correction viewed from different angles. Both insist that the trajectory is primary. Both insist that what matters is not what a system is at a moment but what it has done and what it can still do. Both treat state as a derived notion: useful for some purposes, but never the fundamental unit of description.

This convergence is the organizing principle of the present monograph.

Part II

First Principles

Chapter 3

Deriving the Operators

The Minimal Requirements

If histories are primary and states are derived, then a computational framework must at minimum be able to:

1. Commit to a specific continuation from among the available possibilities.
2. Document that a possible continuation is inadmissible, with reasons.
3. Record a dependency between two elements of the history.
4. Observe the current state of a history under a specific observational framework.

We claim that these four requirements are not only necessary but sufficient: they constitute the minimal complete set of operations on possibility space. All other computational phenomena are derived from these four.

Definition 3.1 (Event Alphabet)**.** Let E be a set of events, partitioned into primitive generators:

$$E = \{\mathsf{Pop}\} \cup \{\mathsf{Refuse}\} \cup \{\mathsf{Bind}\} \cup \{\mathsf{Collapse}\} \cup E_\lambda$$

where E_λ contains the lambda-calculus events $\{\mathsf{LamIntro}, \mathsf{Apply}\}$ and scope markers.

Definition 3.2 (Spherepop World)**.** A *Spherepop world* is a pair $W = (H, \Omega)$ where $H \in \mathcal{H}$ is a history (finite event sequence) and $\Omega \subseteq \Omega_0$ is the current option space (a finite set of available symbols), derived from some initial option space Ω_0 with $|\Omega_0| < \infty$.

The Four Operators

Definition 3.3 (Pop: Commitment)**.** The *commitment operator* for symbol $x \in \Omega$ is

$$P_x : (H, \Omega) \mapsto (H +\!\!+ [\mathsf{Pop}(x)],\ \Omega \setminus \{x\}).$$

P_x is undefined when $x \notin \Omega$. Pop records a commitment and removes the committed symbol from the option space, foreclosing all futures in which x takes a different value.

Definition 3.4 (Refuse: Documented Inadmissibility)**.** The *refusal operator* for symbol x with reason r is

$$R_{x,r} : (H, \Omega) \mapsto (H +\!\!+ [\mathsf{Refuse}(x, r)],\ \Omega).$$

Note that Ω is unchanged: refusal records inadmissibility without foreclosing structural availability. The reason r is a first-class component of the operation. A refusal without a reason documents nothing.

Definition 3.5 (Bind: Dependency Declaration)**.** The *bind operator* is

$$B_{a,b} : (H, \Omega) \mapsto (H +\!\!+ [\mathsf{Bind}(a, b)],\ \Omega).$$

Bind records that two trajectory elements are coupled without consuming either from the option space.

Definition 3.6 (Collapse: Observation under Rule)**.** The *collapse operator* with collapse rule c is

$$C_{x,c} : (H, \Omega) \mapsto (H +\!\!+ [\mathsf{Collapse}(x, c)],\ \Omega),$$

subject to the admissibility precondition $\Gamma \vdash t : \mathsf{Admissible}(T)$. Collapse requires an admissibility certificate. A term whose type is not $\mathsf{Admissible}(\cdot)$ cannot be collapsed—this is a hard type error, not a runtime exception.

Why These Four and No Others

The minimality of the four operators follows from an analysis of what operations on possibility space are conceptually irreducible.

Pop is irreducible because commitment—the foreclosure of alternative futures—cannot be expressed as a combination of documentation, dependency recording,

or observation. It is the primitive act of narrowing possibility.

Refuse is irreducible because documented inadmissibility is distinct from commitment: it records that a path was considered and found inadmissible, which is information that commitment cannot record. A commit of the alternative does not document why the refused path was refused. Reasons are evidence that survives collapse; without them, refusal is indistinguishable from non-consideration.

Bind is irreducible because dependency recording—coupling two trajectory elements—cannot be expressed as commitment or refusal. A bind creates a relationship between elements without consuming either; no combination of Pop and Refuse produces this effect.

Collapse is irreducible because observation under a specific rule is distinct from all three of the above. Pop, Refuse, and Bind all operate on the history and option space; Collapse maps from history space to observable state space under a specified projection. It is the meta-operation that makes histories visible.

Theorem 3.7 (Completeness of the Four Operators)**.** *The four operators* $\{P_x, R_{x,r}, B_{a,b}, C_{x,c}\}$ *are sufficient to express: variable assignment, conditional branching, function application, exception handling, memory deallocation, and concurrent event recording.*

Proof. We establish expressibility for each construct.

Variable assignment $x := v$: Express as P_v followed by $B_{x,v}$. The commitment records the value choice; the bind records the name-value dependency.

Conditional branch on predicate ϕ: Express as P_{ϕ^+} if ϕ holds, or $R_{\phi^+,\mathsf{ConstraintViolation}(\phi)}$ followed by P_{ϕ^-} if not. The refusal records why the positive branch was not taken.

Function application $f(a)$: Expressed as $\mathsf{LamIntro}(x)$ followed by $\mathsf{Apply}(a)$—a deferred Pop. The closure freezes the option space at the moment of abstraction; application is the moment of commitment.

Exception throwing: Expressed as $R_{t,r}$ (documented inadmissibility without system destruction). The type $\mathsf{Refused}(T, r)$ carries the reason as a first-class certificate.

Memory deallocation: Expressed as $R_{p,\mathsf{Deallocated}}$ where p is the pointer symbol. The refusal records that the memory region is no longer admissible to access, making use-after-free a type-level error.

Concurrent event recording: Expressed via Meld, the monoidal composition of two histories. Two concurrent worlds (H_1, Ω_1) and (H_2, Ω_2) are combined as $(H_1 +\!+ H_2, \Omega_1 \cup \Omega_2)$ under an appropriate ordering convention. □

The Conservation Law

The four operators satisfy a conservation law that reflects the fundamental structure of possibility.

Theorem 3.8 (Conservation of Possibility)**.** *For any legal execution sequence*

$$W_0 \xrightarrow{e_1} W_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} W_n$$

where each step is one of P_x, $R_{x,r}$, $B_{a,b}$, or $C_{x,c}$:

1. *$|H_t|$ is strictly monotonically increasing: $|H_{t+1}| = |H_t| + 1$.*
2. *$|\Omega_t|$ is monotonically non-increasing.*
3. *Under pure Pop dynamics, $|H_t| + |\Omega_t| = |\Omega_0|$ for all t.*

Proof. (i) Every operator appends exactly one event to H, so $|H_{t+1}| = |H_t| + 1$.

(ii) P_x removes x from Ω; the remaining operators do not modify Ω. Hence $|\Omega_{t+1}| \leq |\Omega_t|$.

(iii) Under pure Pop dynamics, every step removes exactly one element from Ω and appends one event to H. Starting from $|H_0| = 0$ and $|\Omega_0|$, after t steps $|H_t| = t$ and $|\Omega_t| = |\Omega_0| - t$, giving $|H_t| + |\Omega_t| = |\Omega_0|$. □

The conservation law has the structure of a physical conservation principle: possibility is not created or destroyed—it is either consumed (by Pop) or documented as inadmissible (by Refuse). The operators Bind and Collapse record relationships and observations without affecting the possibility budget.

Definition 3.9 (Generalised Possibility Functional)**.** Define the event weight function $w : E \to \mathbb{N}$ by $w(\mathsf{Pop}(x)) = 1$, $w(\mathsf{Refuse}(x, r)) = 0$, $w(\mathsf{Bind}(a, b)) = 0$, $w(\mathsf{Collapse}(x, c)) = 0$. For a world (H, Ω), the *generalised possibility functional* is

$$\Pi(H, \Omega) = |\Omega| + \sum_{e \in H} w(e).$$

Theorem 3.10 (Conservation of the Possibility Functional)**.** *For any legal execution sequence, $\Pi(H_t, \Omega_t) = |\Omega_0|$ for all t.*

Proof. At each step, exactly one of four operators is applied. Pop: $|\Omega|$ decreases by 1; $w(\mathsf{Pop}) = 1$ is added. Net change: $(-1) + 1 = 0$. Refuse, Bind, Collapse: $|\Omega|$ unchanged; $w = 0$ is added. Net change: 0. In every case $\Pi(H_{t+1}, \Omega_{t+1}) = \Pi(H_t, \Omega_t)$. □

Corollary 3.11 (Irreversibility of Execution)**.** *For any non-empty execution sequence, $|H_{t+1}| > |H_t|$. Therefore no legal execution can return to a previous world state (H_t, Ω_t).*

Proof. Every operator appends exactly one event to H, so $|H_{t+1}| = |H_t| + 1 > |H_t|$. Since H grows strictly, $(H_{t+1}, \Omega_{t+1}) \neq (H_t, \Omega_t)$. □

This irreversibility is not a limitation but a feature. It is the formal correlate of the phenomenological observation that histories are primary: a world that could return to a prior state would be a world in which the history of the intervening steps was undone, which is not computation but its negation.

Chapter 4

The Algebra of Histories

Histories as the Primary Objects

Before introducing types, evaluators, or compilers, we must characterize the mathematical structure of the objects that are primary in the Spherepop ontology. Histories are not merely lists of events. They form a category, and the category structure constrains what operations on histories are meaningful.

Definition 4.1 (History)**.** A *history* over event alphabet E is a finite sequence $H = [e_1, e_2, \ldots, e_n]$ with each $e_i \in E$. The empty history is $\varepsilon = []$.

Definition 4.2 (History Concatenation)**.** For histories $H_1 = [e_1, \ldots, e_m]$ and $H_2 = [f_1, \ldots, f_n]$,

$$H_1 \mathbin{+\!\!+} H_2 = [e_1, \ldots, e_m, f_1, \ldots, f_n].$$

Proposition 4.3 (History Monoid)**.** *The triple $(\mathcal{H}, +\!\!+, \varepsilon)$ is the free monoid over E.*

Proof. Associativity of $+\!\!+$ follows from list concatenation associativity. ε is the left and right identity. Freeness: every history H is uniquely determined by its event sequence, and no non-trivial relation among generators holds. □

The History Category

Definition 4.4 (History Category **Hist**)**.** Define the category **Hist** as follows:

- *Objects*: option spaces $\Omega \subseteq \Omega_0$.
- *Morphisms*: $\mathrm{Hom}(\Omega, \Omega')$ is the set of histories H such that starting from Ω, executing H yields Ω'.
- *Composition*: $H_2 \circ H_1 = H_1 \mathbin{+\!\!+} H_2$ (execute H_1 first, then H_2).
- *Identity*: $\mathrm{id}_\Omega = \varepsilon$.

Proposition 4.5 (**Hist** is a Well-Defined Category)**.** **Hist** *satisfies all category axioms.*

Proof. Associativity of composition follows from associativity of $++$. ε is the identity morphism since $H ++ \varepsilon = H = \varepsilon ++ H$. □

Proposition 4.6 (**Hist** is a Free Strict Monoidal Category)**.** $(\mathbf{Hist}, ++, \varepsilon, \otimes)$ *is a free strict monoidal category where the tensor product* $H_1 \otimes H_2$ *is the Meld operation* (*parallel composition of histories*) *and the monoidal unit is the empty option space with empty history.*

Remark 4.7. The free monoid structure forces `append` (extend by one generator) and `meld` (monoidal composition of two histories) as the only legitimate history operations. The absence of `remove` and `undo` is the mechanization of freeness: there are no relations among generators.

Listing 4.1: History as free monoid in Rust

```
pub fn append(&mut self, event: Event) {
    self.events.push(event);  // one generator
}
pub fn meld(&mut self, other: &History) {
    self.events.extend(other.events.iter().cloned());
}
```

History equality $H_1 = H_2$ is the primary criterion of program identity.

Unique Factorisation

Theorem 4.8 (Unique History Factorisation)**.** *Every non-empty history* $H = [e_1, e_2, \ldots, e_n]$ *admits a unique factorisation into primitive generators:*

$$H = [e_1] ++ [e_2] ++ \cdots ++ [e_n].$$

No other factorisation into generators exists.

Proof. Since $\mathcal{H}$ is the free monoid over E, every element has a unique normal form as a sequence of generators. The generators are singletons $[e]$; freeness of the monoid guarantees there are no non-trivial relations among generators. □

Remark 4.9. Unique factorisation is what makes history equivalence a decidable correctness criterion for compilation. If two execution paths produce the same history, they must have produced the same sequence of primitive events in the same order.

Admissibility as a Set-Valued Function

Definition 4.10 (Admissible Futures)**.** For a history H, the set of admissible future symbols is

$$\mathcal{A}(H) = \Omega_0 \setminus \{x \mid \exists r.\, \mathsf{Refuse}(x, r) \in H\}.$$

Definition 4.11 (Admissibility Contraction)**.** Let $H' = H +\!\!+ [\mathsf{Refuse}(x, r)]$. Then $\mathcal{A}(H') \subseteq \mathcal{A}(H)$, and specifically $x \notin \mathcal{A}(H')$ while x may have been in $\mathcal{A}(H)$.

Remark 4.12. The asymmetry between Pop and Refuse is critical. Pop contracts the structural option space Ω. Refuse contracts the admissibility set $\mathcal{A}(H)$ without touching Ω. Two execution paths may have the same Ω while having different $\mathcal{A}(H)$—they are structurally equivalent but semantically distinguished by their refusal histories. This distinction is invisible to any state-centric computational model.

Part III

Operational Semantics and Collapse

Chapter 5

Small-Step Semantics

The Transition System

We write $(H, \Omega) \xrightarrow{\alpha} (H', \Omega')$ for a single execution step labelled α.

$$\frac{x \in \Omega}{(H, \Omega) \xrightarrow{\mathsf{pop}(x)} (H +\!\!+ [\mathsf{Pop}(x)],\ \Omega \setminus \{x\})} \quad \text{[POP]}$$

$$\frac{}{(H, \Omega) \xrightarrow{\mathsf{refuse}(x,r)} (H +\!\!+ [\mathsf{Refuse}(x, r)],\ \Omega)} \quad \text{[REFUSE]}$$

Note: [REFUSE] has no premise—any symbol may be refused. The effect on the option space is nil; the effect on admissibility is non-nil.

$$\frac{}{(H, \Omega) \xrightarrow{\mathsf{bind}(a,b)} (H +\!\!+ [\mathsf{Bind}(a, b)],\ \Omega)} \quad \text{[BIND]}$$

$$\frac{\Gamma \vdash t : \mathsf{Admissible}(T)}{(H, \Omega) \xrightarrow{\mathsf{collapse}(x,c)} (H +\!\!+ [\mathsf{Collapse}(x, c)],\ \Omega)} \quad \text{[COLLAPSE]}$$

The premise $\Gamma \vdash t : \mathsf{Admissible}(T)$ is the type-theoretic admissibility guard. Collapse requires an admissibility certificate. Without it, collapse is a hard type error.

Big-Step Semantics

The big-step relation $\langle t, W \rangle \Downarrow \langle v, W' \rangle$ relates terms and worlds to values and resulting worlds.

Definition 5.1 (Evaluation Result)**.** An *evaluation result* is a triple (v, H, a) where v is a value, H is the accumulated history, and $a \in \{\top, \bot\}$ is the admissibility flag. The admissibility flag is $\top$ if every operation in H was admissible and $\bot$ if

any refusal was encountered.

The key invariant is that evaluation never discards history. Every step appends to H; no step modifies or removes prior events. The world grows monotonically.

Listing 5.1: Evaluation result preserving all history

```
pub struct EvalResult {
    pub value: Value,
    pub world: World,    // world.history grows monotonically
    pub admissible: bool,
}
```

Multi-Step Reduction

Definition 5.2 (Multi-Step Reduction)**.** The multi-step reduction $\to^*$ is the reflexive transitive closure of $\to$. A term t is *normal* if there is no t' with $t \to t'$.

Theorem 5.3 (History Monotonicity)**.** *If $(H_0, \Omega_0) \to^* (H_n, \Omega_n)$ then H_0 is a prefix of H_n: there exists H' such that $H_n = H_0 +\!+ H'$.*

Proof. By induction on the length of the reduction sequence. Each step appends exactly one event to the history. The result follows by transitivity. □

Theorem 5.4 (Option Space Monotonicity)**.** *If $(H_0, \Omega_0) \to^* (H_n, \Omega_n)$ then $\Omega_n \subseteq \Omega_0$.*

Proof. Only Pop removes elements from Ω; all other operators leave it unchanged. By induction, Ω can only shrink. □

Chapter 6

Collapse as Quotient

The Insufficiency of Unparameterized Collapse

The original Spherepop design represented collapse as simply `Collapse(Symbol)`. This was insufficient, and the insufficiency is philosophically illuminating.

Observable state depends on how you look. The same history, examined by different observers with different observational frameworks, yields different observable states. A collapse without a rule is a collapse into an unspecified notion of visibility. The rule is not an implementation detail; it is what makes the observation meaningful.

Definition 6.1 (Collapse Rule)**.** A *collapse rule* is a function $c : \mathcal{H} \to O_c$ from the history monoid to an observational space O_c.

Definition 6.2 (Observational Equivalence)**.** Two histories $H_1, H_2 \in \mathcal{H}$ are *observationally equivalent under rule* c, written $H_1 \sim_c H_2$, if and only if $c(H_1) = c(H_2)$.

Definition 6.3 (Observable State as Quotient)**.** The *observable state space* under rule c is the quotient:

$$O_c = \mathcal{H}/\sim_c.$$

An observable state is an equivalence class of histories.

Observable state is not a state. Observable state is a quotient. Two histories that are globally distinct may be observationally indistinguishable under a specific collapse rule. This is a feature: the collapse rule specifies which distinctions are relevant to a given question.

The Four Canonical Rules

Definition 6.4 (Identity Rule)**.** $c_I : \mathcal{H} \to \mathcal{H}$ is the identity: $c_I(H) = H$. Under the identity rule, every event is visible and $O_I \cong \mathcal{H}$.

Definition 6.5 (LastWrite Rule)**.** c_{LW} maps H to the most recent pop for each

symbol, with subsequent refusals retroactively removing symbols:

$$c_{LW}(H) = \{x \mapsto 1 \mid \mathsf{Pop}(x) \in H \text{ and } \nexists r.\, \mathsf{Refuse}(x, r) \in H \text{ after the last } \mathsf{Pop}(x)\}.$$

Definition 6.6 (Accumulate Rule)**.** $c_{\text{acc}}(H)$ maps each symbol to the count of its Pop events:

$$c_{\text{acc}}(H)(x) = |\{i \mid e_i = \mathsf{Pop}(x)\}|.$$

Definition 6.7 (Projection Rule)**.** For a label prefix L, the projection rule π_L restricts to events whose symbol names share prefix L:

$$\pi_L(H) = [e_i \in H \mid \text{sym}(e_i) \text{ starts with } L].$$

Proposition 6.8 (Coarsening Order)**.** *The collapse rules form a partial order by coarsening: $c_1 \leq c_2$ iff $\sim_{c_2} \subseteq \sim_{c_1}$ (finer equivalence = more informative observation). Under this order: c_I is the finest; π_L is coarser than c_I; c_{LW} and c_{acc} are incomparable in general.*

The Collapse Functor

Definition 6.9 (Observable-State Category $\mathbf{Obs}_c$)**.** For a collapse rule c, define the category $\mathbf{Obs}_c$:

- *Objects*: observable states $o \in O_c$.
- *Morphisms*: $\text{Hom}(o_1, o_2)$ is the set of observable transitions transforming o_1 into o_2 under rule c.
- *Composition*: sequential observable transition.

Definition 6.10 (Collapse Functor)**.** For collapse rule c, define $F_c : \mathbf{Hist} \to \mathbf{Obs}_c$ by $F_c(\Omega) = c(\varepsilon_\Omega)$ on objects and $F_c(H) = c(H)$ on morphisms.

Theorem 6.11 (Functoriality of Collapse)**.** *F_c is a well-defined functor when c respects composition:*

$$c(H_2 +\!+ H_1) = c(H_2) \circ c(H_1).$$

Proof. Identity preservation: $F_c(\varepsilon) = c(\varepsilon) = \text{id}_{O_c}$. Composition preservation: $F_c(H_2 \circ H_1) = F_c(H_1 +\!+ H_2) = c(H_1 +\!+ H_2) = c(H_2) \circ c(H_1) = F_c(H_2) \circ F_c(H_1)$ when c respects composition. □

Remark 6.12. The functoriality of collapse elevates it from an implementation detail to a category-theoretic object. Collapse is a structure-preserving map between categories. The collapse rule specifies which structures are preserved,

and different rules preserve different structures. c_{LW} is not generally functorial because the last-write of a concatenated history is not the composition of the last-writes of its parts—a later segment can retroactively affect the observable state of an earlier one.

The Universal Property of Collapse

Theorem 6.13 (Universal Property of Collapse)**.** *Let $q_c : \mathcal{H} \to O_c$ be the quotient map for rule c. If $f : \mathcal{H} \to X$ is any function satisfying*

$$H_1 \sim_c H_2 \implies f(H_1) = f(H_2),$$

then there exists a unique $\bar{f} : O_c \to X$ such that $f = \bar{f} \circ q_c$.

Proof. By the universal property of quotient sets. Define $\bar{f}([H]_c) = f(H)$. This is well-defined because f is constant on equivalence classes. Uniqueness: any $\bar{f}$ satisfying $f = \bar{f} \circ q_c$ must map $[H]_c$ to $f(H)$. □

Remark 6.14. The universal property says that observable state is not one representation among many. It is the universal representation among all representations that identify the same histories as c does. Every coarser observation necessarily passes through O_c.

Part IV

Type Theory

Chapter 7

Types as Admissibility Certificates

The Inadequacy of Value-Set Types

The standard interpretation of types—a type is a set of admissible values—is adequate for state-centric languages. It is insufficient for Spherepop.

Consider what a type must certify in a history-primary setting. A value v of type T is not merely a member of a set. It is the result of a path through history that reached v without violating any constraint represented by T. The type is not a description of v at a moment; it is a certificate about the history that produced v.

The Type Language

Definition 7.1 (Spherepop Types)**.** The type language of Spherepop is defined inductively:

$$\begin{aligned} T ::= & \ \mathsf{Unit} \mid \mathsf{Never} \mid X \\ & \mid \Pi(x : T_1).T_2 \\ & \mid \mathsf{Admissible}(T) \\ & \mid \mathsf{Refused}(T, r) \\ & \mid \mathsf{Collapsed}(T, c) \\ & \mid \mathsf{Process}(T_1 \rightsquigarrow T_2) \end{aligned}$$

The simple function type $T_1 \to T_2$ is notation for $\Pi(\_ : T_1).T_2$.

The distinctive types carry certificates, not just values:

- $\mathsf{Admissible}(T)$: the path that produced this term was admissible.
- $\mathsf{Refused}(T, r)$: the term was encountered but found inadmissible for reason r. The reason is part of the type—two terms refused for different reasons

have genuinely different types.

- $\mathsf{Collapsed}(T, c)$: an admissible term was observed under collapse rule c and the observation is sealed. The rule c is part of the type.

The Type Rules

We write $\Gamma \vdash t : T$ for the standard typing judgement.

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{pop}(t) : \mathsf{Admissible}(T)} \quad \text{[T-POP]}$$

Pop transforms any typed term into an admissible term. It is the mechanism by which structural commitment produces a reachability certificate.

$$\frac{\Gamma \vdash t : T \quad r \in \mathsf{RefusalReason}}{\Gamma \vdash \mathsf{refuse}(t, r) : \mathsf{Refused}(T, r)} \quad \text{[T-REFUSE]}$$

Refusal is typed: the type carries the reason. A term refused for $\mathsf{ConstraintViolation}(c)$ has a type formally distinct from one refused for $\mathsf{NotAvailable}$.

$$\frac{\Gamma \vdash t : \mathsf{Admissible}(T) \quad c \in \mathsf{CollapseRule}}{\Gamma \vdash \mathsf{collapse}(t, c) : \mathsf{Collapsed}(T, c)} \quad \text{[T-COLLAPSE]}$$

Collapse requires an admissibility certificate. A term whose type is not $\mathsf{Admissible}(\cdot)$ cannot be collapsed—this is a hard type error, not a runtime exception.

$$\frac{\Gamma \vdash a : T_1 \quad \Gamma \vdash b : T_2}{\Gamma \vdash \mathsf{bind}(a, b) : \mathsf{Process}(T_1 \rightsquigarrow T_2)} \quad \text{[T-BIND]}$$

$$\frac{\Gamma, x : T_1 \vdash b : T_2}{\Gamma \vdash \lambda x : T_1.\, b : \Pi(x : T_1).T_2} \quad \text{[T-LAM]}$$

$$\frac{\Gamma \vdash f : \Pi(x : T_1).T_2 \quad \Gamma \vdash a : T_1}{\Gamma \vdash f\ a : T_2[x := a]} \quad \text{[T-APP]}$$

Open Worlds and Closed Worlds

During implementation of the Spherepop type checker, a conflict emerged between how the interpreter and type checker treated free variables. The apparent bug conceals a genuine theoretical discovery: which behavior is correct depends on which assumption about the boundary of the world is in force.

Definition 7.2 (Closed-World Context)**.** A *closed-world context* Γ_c satisfies:

$$x \notin \Gamma_c \implies \Gamma_c \vdash x : \bot.$$

The absence of a declaration is the absence of the object.

$$\frac{x \notin \Gamma \quad \Gamma \text{ is closed-world}}{\Gamma \nvdash x : T \text{ for any } T} \quad \text{[T-VAR-CLOSED]}$$

Definition 7.3 (Open-World Context)**.** An *open-world context* Γ_o satisfies:

$$x \notin \Gamma_c \implies \Gamma_o \vdash x : \mathsf{Unit}.$$

The absence of a declaration is undecided possibility.

$$\frac{x \notin \Gamma \quad \Gamma \text{ is open-world}}{\Gamma \vdash x : \mathsf{Unit}} \quad \text{[T-VAR-OPEN]}$$

Proposition 7.4 (Monotonicity of Provability)**.** $\Gamma_c \subseteq \Gamma_o \implies \text{Provable}(\Gamma_c) \subseteq \text{Provable}(\Gamma_o)$.

Remark 7.5. The open/closed world distinction is not an engineering convenience. It is the mechanization of an epistemological commitment: a closed-world type checker assumes complete knowledge of the reachable; an open-world type checker assumes partial knowledge. A language whose fundamental ontology is about possibility space cannot leave this distinction implicit. The TypeMode is preserved through `extend`: a closed-world context cannot silently become open-world mid-derivation.

Principal Theorems

Theorem 7.6 (Collapse Soundness)**.** *If* $\Gamma \vdash \mathsf{collapse}(t, c) : T$ *then* $\Gamma \vdash t : \mathsf{Admissible}(T')$ *for some* T'*, and* $T = \mathsf{Collapsed}(T', c)$.

Proof. By inversion on [T-COLLAPSE]: the only rule that derives a Collapse judgement requires the premise $\Gamma \vdash t : \mathsf{Admissible}(T)$, so the inner term must have an admissible type. □

Theorem 7.7 (Type Preservation)**.** *If* $\Gamma \vdash t : T$ *and* $t \to t'$*, then* $\Gamma \vdash t' : T$.

Proof. By structural induction on the reduction relation $\to$. The key cases are [T-POP], [T-REFUSE], and [T-COLLAPSE], each mapping a reduction step to an admissibility transformation closed under the respective rule. Beta-reduction

requires a substitution lemma: $\Gamma, x : T_1 \vdash b : T_2$ and $\Gamma \vdash a : T_1$ implies $\Gamma \vdash b[x := a] : T_2[x := a]$. □

Theorem 7.8 (Progress)**.** *If* $\vdash t : T$ (*well-typed in the empty context*) *then either* t *is a value, or there exists* t' *such that* $t \to t'$.

Proof. By structural induction on t. Variable terms are values. Lambda terms are values (closures). pop, refuse, bind, and collapse of a well-typed inner term either reduce (if the inner term reduces) or produce a value (if the inner term is a value). □

Chapter 8

Dependent Types and Admissibility Lattices

Dependent Process Types

The current $\mathsf{Process}(T_1 \rightsquigarrow T_2)$ is first-order. A dependent process type allows the output type to depend on the specific value consumed.

Definition 8.1 (Dependent Process Type)**.** A *dependent process type*

$$\Pi(x : T_1).\, \mathsf{Process}(x, f(x))$$

allows the type of the output process to depend on the value x of the input. This is the natural extension of dependent type theory into the process calculus setting.

Example 8.2. A function that reads a file and returns a proof that the file was read can be typed as:

$$\Pi(\mathit{path} : \mathsf{FilePath}).\, \mathsf{Process}(\mathit{path}, \mathsf{Admissible}(\mathsf{FileContents}(\mathit{path}))).$$

The Admissible in the output type certifies that the reading path was admissible—no access violation occurred, the file existed, and the permissions were valid.

Admissibility Lattices

The current admissibility structure is Boolean: a term is admissible or not. A more refined theory recognizes that admissibility is graded.

Definition 8.3 (Admissibility Lattice)**.** An *admissibility lattice* $(\mathcal{L}, \leq, \top, \bot, \wedge, \vee)$ is a bounded lattice where:

- $\top$ is full admissibility: the trajectory satisfies all constraints.
- $\bot$ is total inadmissibility: the trajectory violates all constraints.
- $\ell_1 \wedge \ell_2$ is the admissibility under the conjunction of constraints c_1 and c_2.

- $\ell_1 \vee \ell_2$ is the admissibility under the disjunction.

Definition 8.4 (Graded Admissibility Type)**.** $\mathsf{Admissible}_\ell(T)$ denotes admissibility at level $\ell \in \mathcal{L}$. The standard $\mathsf{Admissible}(T)$ is $\mathsf{Admissible}_\top(T)$.

Proposition 8.5 (Admissibility Lattice Typing)**.** *The typing rule for graded admissibility is:*

$$\frac{\Gamma \vdash t : T \quad \ell \in \mathcal{L}}{\Gamma \vdash \mathsf{pop}_\ell(t) : \mathsf{Admissible}_\ell(T)}$$

Collapse requires admissibility at level at least ℓ_0 for some threshold ℓ_0 determined by the collapse rule c.

Remark 8.6. Admissibility lattices connect the Spherepop type theory to the admissibility geometry program of the RSVP and CLIO frameworks. The xylomorphic criterion $\lambda < 1$ from the admissibility field is a threshold condition in a continuous admissibility lattice, where admissibility is measured by a scalar λ and collapse is only valid when λ falls below the threshold.

Proof-Carrying Refusal

The RefusalReason type is currently a vocabulary for inadmissibility. It is not yet a proof system. A fully realized Spherepop type system would require RefusalReason to be a proof term: evidence that can be checked, not merely a label.

Definition 8.7 (Proof-Carrying Refusal)**.** A *proof-carrying refusal* is a refusal of the form $\mathsf{refuse}(t, \pi)$ where π is a proof term of type $\mathsf{Inadmissible}(t, \Omega)$—a formal certificate that t is inadmissible given the current admissibility set $\mathcal{A}(H)$.

Definition 8.8 (Inadmissibility Proof System)**.** The inadmissibility propositions and their proofs are:

$$\begin{aligned}
\mathsf{AlreadyRefused}(x) &: x \in \{y \mid \exists r.\, \mathsf{Refuse}(y, r) \in H\} \\
\mathsf{ConstraintViolation}(c, x) &: c(x) > \mathsf{threshold}(c) \\
\mathsf{NotAvailable}(x) &: x \notin \Omega \\
\mathsf{DependencyFailed}(x, y) &: \mathsf{Bind}(x, y) \in H \wedge \mathsf{AlreadyRefused}(y)
\end{aligned}$$

Each proposition has a type-theoretic proof term, and $\mathsf{refuse}(t, \pi)$ is only well-typed when π proves one of these propositions for t.

Theorem 8.9 (Refusal Completeness)**.** *Every inadmissible trajectory has a proof-carrying refusal at some point. That is, if $\mathcal{A}(H) \not\ni x$ for some x that is later attempted, then there exists a proof term π of $\mathsf{Inadmissible}(x, \mathcal{A}(H))$.*

Proof. $x \notin \mathcal{A}(H)$ implies $\exists r.\, \mathsf{Refuse}(x, r) \in H$. The proof term $\pi = \mathsf{AlreadyRefused}(x)$

witnesses that x is in the refused set, which is defined as the complement of $\mathcal{A}(H)$. □

Part V

Notes as Spherepop Objects

Chapter 9

The Seven Note Classes as Operator Realizations

The four Spherepop operators—Pop, Refuse, Bind, Collapse—are abstract operations on possibility space. Notes are their physical realization in cognition, computation, language, and civilization. This chapter develops the correspondence systematically.

The Classification Principle

Definition 9.1 (Dominant Operator)**.** The *dominant operator* of a note class is the Spherepop operator that accounts for the primary function of notes in that class. Most notes mix operators, but each class has a dominant one.

Theorem 9.2 (Notes as Spherepop Programs)**.** *Every note is a Spherepop program. Every Spherepop program is a note. The correspondence is given by the following bijection between note classes and operator combinations:*

Note Class	*Dominant Operator*	*Secondary Operators*
Capture	*Refuse*	—
Prospective	*Bind*	*Pop* (*deferred*)
Repair	*Refuse* + *Pop*	—
Coordination	*Bind*	*Refuse*
Refusal	*Refuse*	— (*pure form*)
Generative	*Pop*	*Bind*
Collapse	*Collapse*	—

Chapter 10

Capture Notes

Definition 10.1 (Capture Note)**.** A *capture note* is a note whose primary function is to preserve a continuation that exists at the time of note creation but is expected to become inaccessible. In Spherepop terms, a capture note performs a Refuse operation with respect to the projection $\sigma : \mathcal{H} \to S$: it preserves history fibers that the projection would collapse.

A photograph refuses the collapse of a perceptual trajectory. An audio recording refuses the collapse of a sonic trajectory. A laboratory notebook refuses the collapse of a procedural and cognitive trajectory.

Proposition 10.2 (Capture Notes and Refuse)**.** *A capture note N performs a Refuse operation with respect to the projection $\sigma : \mathcal{H} \to S$ if and only if there exists a history fiber $h \in \sigma^{-1}(S)$ such that $P_A(c_h, t \mid N) > P_A(c_h, t)$, where c_h is the continuation associated with history h.*

The quality criterion for a capture note is not fidelity to the original state but adequacy as a reconstruction substrate. A photograph that captures the correct person but not the context may be a poor capture note even at high resolution, if the lost context is what matters for future reconstruction. The adequacy of a capture note depends on what future continuations are anticipated.

Chapter 11

Prospective Notes

Definition 11.1 (Prospective Note)**.** A *prospective note* is a note whose primary function is to preserve access to a continuation that does not yet exist at note creation time but is anticipated. In Spherepop terms, a prospective note performs a Bind operation between present agent state A_t and future continuation $c_{t'}$.

A calendar entry binds future action to present scheduling. A blueprint binds future construction to present design. A roadmap binds future development to present strategic commitment. Source code binds future computation to present specification—and is unique in that the binding is mechanically executable.

Proposition 11.2 (Prospective Notes and Temporal Binding)**.** *A prospective note N performs a Bind operation if and only if*

$$P_A(c_{t'}, t' \mid N, A_t) > P_A(c_{t'}, t' \mid A_t),$$

where $t' > t$ and the conditioning on A_t indicates the agent's present state is available in both cases.

Proposition 11.3 (Collective Reachability from Coordination)**.** *Let $A_1, \ldots, A_n$ be agents with independent continuation spaces $\mathcal{C}_1, \ldots, \mathcal{C}_n$. A coordination note N increases collective reachability if*

$$|\mathcal{C}(A_1, \ldots, A_n \mid N)| > |\mathcal{C}_1| + \cdots + |\mathcal{C}_n|.$$

The proposition captures the supraadditive character of coordination: bound agents can traverse paths that no individual could reach alone.

Chapter 12

Repair, Refusal, and Generative Notes

Repair Notes

Definition 12.1 (Repair Note)**.** A *repair note* is a note whose primary function is to reduce the cost of reconstructing a continuation after it has been or is anticipated to be interrupted. It combines Refuse (preserving the history of the trajectory against collapse) and Pop (enabling re-entry at a specific point after interruption).

Theorem 12.2 (Notes as Pre-emptive Repair)**.** *Every note N with positive repair value $R_v(N, c) = C_r(c) - C_r(c \mid N) > 0$ constitutes a pre-emptive repair: it reduces expected reconstruction cost prior to any failure occurring.*

Proof. $R_v(N, c) > 0$ implies $C_r(c \mid N) < C_r(c)$. Since the note exists before any failure occurs, the cost reduction is in force from the moment of note creation. The note therefore functions as a repair mechanism that precedes the event it repairs for. □

A troubleshooting guide is a repair note: a stored Pop operator for a class of failure states. Each entry says: if you arrive at this failure state, the continuation from here to a functional state runs through these steps.

Refusal Notes

Definition 12.3 (Refusal Note)**.** A *refusal note* is a note whose primary function is to prevent the collapse of an existing distinction, independently of whether the preserved distinction is expected to be used. It is the closest realization of the pure Refuse operation.

Mathematical proofs, cryptographic checksums, archival copies, legal records, and formal specifications belong to this class.

Proposition 12.4 (Proofs as Refusal Notes)**.** *A mathematical proof P of theorem T from premises Γ satisfies*

$$P_A(c_{\Gamma\vdash T}, t \mid P) > P_A(c_{\Gamma\vdash T}, t)$$

for any agent A with sufficient background knowledge, at any future time t.

The proof refuses the collapse of an inferential trajectory into mere assertion. Its value is not actualized use but preserved potential: it maintains the reachability of the derivation whether or not anyone consults it.

Generative Notes

Definition 12.5 (Generative Note)**.** A *generative note* is a note that creates access to continuations that were not reachable before the note's creation. It is the primary realization of the Pop operation: it opens new trajectories in possibility space.

Proposition 12.6 (Generative Notes and Novel Reachability)**.** *A generative note N satisfies*

$$\exists\, c \in \mathcal{C} : P_A(c, t \mid N) > 0 \text{ and } P_A(c, t) = 0.$$

That is, N creates access to continuations that were previously unreachable.

Scientific theories, maps, programming languages, mathematical formalisms, and ontologies are generative notes. A programming language is a generative note that is mechanically executable: it defines not just a space of possible programs but a machine for traversing that space.

Chapter 13

Collapse Notes and the Ephemerality Inversion

The Hidden Class

Definition 13.1 (Collapse Note)**.** A *collapse note* is an artifact that performs a deliberate projection $\sigma_N : \mathcal{H} \to S_N$, compressing a history or artifact into a more compact representation at the cost of detail, in order to make the representation navigable. It is the realization of the Collapse operator.

File names, titles, abstracts, indexes, summaries, taxonomies, classifications, labels, and keywords are all collapse notes. The paradox of collapse notes is that they appear to be the opposite of notes—they destroy information—yet they are indispensable to the function of every other note class.

Theorem 13.2 (Necessity of Collapse Notes)**.** *For any sufficiently large collection of notes $\mathcal{N}$, there exists a threshold $|\mathcal{N}| > K$ above which the reachability of individual notes in $\mathcal{N}$ decreases in the absence of collapse notes over $\mathcal{N}$.*

Proof. Search cost through an unstructured collection grows at least linearly with collection size. For $|\mathcal{N}| > K$, exhaustive search becomes too expensive for practical use. A collapse note—index, catalogue, taxonomy—reduces search cost by imposing structure, making individual notes reachable at sublinear cost. Without such structure, notes become inaccessible through navigational failure, not destruction. □

Remark 13.3. A word is a collapse note: it compresses a complex of experience, association, inference, and usage into a single retrievable token. A noun is a collapse note about a process. The passive theory would not recognize these as notes at all. The Spherepop theory recognizes them immediately as the most pervasive and powerful class of collapse notes civilization has developed.

The Ephemerality Inversion

Definition 13.4 (Continuation Completion)**.** A continuation c is *complete* with respect to note N at time t if the trajectory encoded by N has been successfully traversed.

Theorem 13.5 (Ephemerality Inversion)**.** *Let N be a note encoding a single continuation c. If c has been completed, then the destruction of N does not reduce the reachability of any remaining continuation. Therefore the ephemerality of N after completion of c implies neither loss nor failure.*

Proof. By assumption, c is the sole continuation for which N increases reachability. After completion of c, $V_C(N) = 0$. Since continuation volume is zero, the destruction of N reduces reachability by zero. □

The grocery list discarded at the supermarket exit has not failed. It has succeeded completely. The theorem generates an inversion of the passive theory's ranking:

Fate	Passive Theory	Continuation Theory
Executed and discarded	Low value	Optimal
Preserved and re-entered	Medium value	Valuable
Preserved, never re-entered	High value	Unrealized potential

The celebrated survivors of antiquity—incomplete manuscripts, undelivered letters, abandoned plans—survive precisely because their continuations were never completed. The completed notes, the sent letters, the finished buildings, left no trace because they succeeded.

Part VI

Compilation, Garbage Collection, and Error Correction

Chapter 14

The Event IR and Compiler Correctness

The Failure of Traditional Compiler Correctness

The standard notion of compiler correctness is: a compiler is correct if the compiled program produces the same output as the interpreted program for every input. This criterion is adequate for state-centric languages. It is insufficient for Spherepop.

Two programs that produce the same final value may have constructed radically different histories to get there, and those histories are part of what the program is.

Definition 14.1 (History Equivalence as Correctness)**.** Let $I : P \to \mathcal{H}$ be the interpreter function mapping programs to histories, and $V : \mathsf{IR} \to \mathcal{H}$ the VM mapping event-IR blocks to histories, and $C : P \to \mathsf{IR}$ the compiler. A compiler C is correct if for all programs p and initial worlds W:

$$I(p, W).\mathsf{history} = V(C(p), W).\mathsf{history}.$$

Output equivalence (same final value) is a strictly weaker criterion that is insufficient for Spherepop.

Event IR

Conventional intermediate representations encode instructions that mutate machine state. Spherepop IR encodes *events*—records of occurrences that persist as part of the history the VM is constructing.

Definition 14.2 (Event IR)**.** The Spherepop intermediate representation is a sequence of IrEvent terms:

$$\mathsf{IR} ::= \mathsf{Pop} \mid \mathsf{Refuse}(r) \mid \mathsf{Collapse}(c) \mid \mathsf{Bind} \mid \mathsf{Load}(x) \mid \mathsf{Apply} \mid \cdots$$

Each event carries its reason or rule as a first-class argument, preserving the full certificate structure of the source term.

Listing 14.1: Event IR in Rust

```
pub enum IrEvent {
    Pop,
    Refuse { reason: RefusalReason },
    Collapse { rule: CollapseRule },
    Bind,
    Load(Symbol),
    Apply,
    LamIntro(Symbol, Type),
}
```

Proposition 14.3 (IR Faithfulness)**.** *The compilation function* $C : \mathsf{Term} \to \mathsf{IR}$ *is faithful: for every primitive term* t*,* $C(t)$ *contains exactly the event variants that* $I(t)$ *would append to the history, with identical arguments.*

Theorem 14.4 (Replay Equivalence)**.** *The Spherepop compiler satisfies history equivalence for all programs expressible in the Spherepop core calculus.*

Proof. By structural induction on program terms. For each primitive operation (Refuse, Bind, Collapse, Pop), the compiler emits the corresponding IrEvent variant, and the VM step performs exactly the same World mutation as the interpreter's eval case. The history appended is therefore identical. For Seq: each sub-term is compiled in order, and since histories compose by concatenation, the composed history of the compiled form equals that of the interpreted form by induction. □

Theorem 14.5 (Compiler Full Faithfulness)**.**

$$I(p_1).\mathsf{history} = I(p_2).\mathsf{history} \iff V(C(p_1)).\mathsf{history} = V(C(p_2)).\mathsf{history}.$$

Proof. ($\Rightarrow$) By Replay Equivalence applied twice. ($\Leftarrow$) Similarly. The biconditional follows from the symmetry of history equivalence. □

Chapter 15

Garbage Collection as Admissibility-Guided Collapse

The Standard Problem

In conventional languages, garbage collection identifies and reclaims memory that is no longer reachable from the program's roots. The criterion is purely structural: an object is garbage if no live reference points to it.

In Spherepop, the analogous question is: which portions of the history are no longer needed to support any reachable continuation? The answer is richer because it depends not just on structural reachability but on admissibility.

History Segments and Liveness

Definition 15.1 (History Segment Liveness)**.** A history segment $H[i : j] = [e_i, e_{i+1}, \ldots, e_j]$ is *live at time* t if there exists an agent A and a continuation c such that:

1. c is reachable at time t: $P_A(c, t) > 0$, and
2. c depends on $H[i : j]$: $P_A(c, t \mid \neg H[i : j]) < P_A(c, t)$.

A segment is *dead* if it is not live.

Definition 15.2 (Admissibility-Guided GC)**.** *Admissibility-guided garbage collection* removes dead history segments while preserving the admissibility certificates needed for future collapses. Formally, let $\mathcal{A}_{\text{live}}(H)$ be the set of future continuations still reachable. GC produces $H' \subseteq H$ (as a subsequence) satisfying:

$$\mathcal{A}(H') = \mathcal{A}(H) \cap \{x : \exists c \in \mathcal{A}_{\text{live}}(H),\ x \text{ needed by } c\}.$$

GC as Collapse

Theorem 15.3 (GC as Collapse Composition)**.** *Admissibility-guided GC is equivalent to applying a collapse rule c_{GC} that maps H to the subsequence of live events:*

$$c_{\text{GC}}(H) = [e \in H : e \textit{ is live in } H].$$

c_{GC} is a valid collapse rule when liveness is stable under extension: if e is live in H, it is live in $H +\!+ H'$ for all H'.

Proof. Liveness is defined in terms of reachable continuations. Since histories grow monotonically and Pop operations only add to the committed history, a past commitment that enables a reachable continuation remains enabling under history extension. Therefore c_{GC} respects composition: the GC of a concatenated history equals the concatenation of the GC of the parts. □

Corollary 15.4 (GC Preserves Admissibility)**.** *If $c_{\text{GC}}(H) = H'$, then $\mathcal{A}(H') \supseteq \mathcal{A}_{\text{live}}(H)$.*

Proof. GC removes only dead events. A dead Refuse event is one that marks a symbol x as inadmissible when x is no longer needed by any reachable continuation. Removing it restores x to the admissibility set. Since no live continuation needs x, this does not affect any future collapse certificate. □

Three GC Strategies

Different collapse rules for GC yield different strategies with different cost-benefit profiles.

Definition 15.5 (Mark-Scan GC)**.** *Mark-Scan GC* applies the identity rule c_I to the subhistory of live events: it retains the full sequence of live events in order. This is the most informative strategy and preserves all historical structure needed for replay.

Definition 15.6 (Quotient GC)**.** *Quotient GC* applies a coarser rule c_{LW} to the history before collecting: it retains only the last commitment to each symbol. This is equivalent to state-based GC with a history log and loses intermediate state information.

Definition 15.7 (Certificate-Only GC)**.** *Certificate-Only GC* retains only Refuse events (inadmissibility certificates) and Collapse events (observation certificates), discarding all Pop and Bind events whose certificates have been sealed. This

minimizes memory use while preserving the admissibility structure needed for future type checking.

Theorem 15.8 (GC Strategy Tradeoff)**.** *Let $M(c_{\text{GC}})$ denote memory usage and $R(c_{\text{GC}})$ denote recovery power (the set of continuations that can be reconstructed after GC). Then:*

$$M(c_I) \geq M(c_{LW}) \geq M(c_{\text{cert}}) \quad \textit{and} \quad R(c_I) \geq R(c_{LW}) \geq R(c_{\text{cert}}).$$

Proof. The identity rule retains the most events and therefore uses the most memory but also preserves the most recovery information. Each coarsening reduces both memory usage and recovery power by identifying more history segments as equivalent. □

Admissibility Certificates and GC Safety

A key constraint on GC in Spherepop is that it must not invalidate pending type judgements. If a collapse term $\mathsf{collapse}(t, c)$ is waiting on the admissibility certificate $\Gamma \vdash t : \mathsf{Admissible}(T)$, GC must retain the history events that support that certificate.

Definition 15.9 (GC Safety)**.** A GC operation is *safe* if for every pending collapse judgement $\Gamma \vdash \mathsf{collapse}(t, c) : \mathsf{Collapsed}(T, c)$, the history events needed to support $\Gamma \vdash t : \mathsf{Admissible}(T)$ are retained after GC.

Theorem 15.10 (Safety of Certificate-Only GC)**.** *Certificate-Only GC is safe: it retains all Refuse events that contribute to any pending admissibility certificate.*

Proof. The admissibility type $\mathsf{Admissible}(T)$ is established by the sequence of Pop and Refuse events that produced t. Certificate-Only GC retains all Refuse events (inadmissibility certificates) unconditionally. The Pop events can be inferred from the Collapse certificates that seal their results. Therefore all information needed for pending collapse judgements is preserved. □

Chapter 16

Error Correction as Certified Refusal

The Standard Approach and Its Limits

In conventional languages, error handling is an afterthought: exceptions are thrown when something goes wrong, and error handling code is written separately from the main computation. This approach has three problems. First, it treats errors as exceptional when they are often structurally inevitable. Second, it provides no information about why the error occurred. Third, it makes error recovery expensive because the history of the computation is not available.

In Spherepop, errors are not exceptional. Every Refuse event is a documented non-traversal of a possible path. Error handling is not separate from the main computation; it is part of the computation's history.

Error Correction Codes in History Space

Definition 16.1 (History Redundancy)**.** A history H has *redundancy factor* k if there exist k disjoint subsequences $H_1, \ldots, H_k \subseteq H$ (as event subsequences) such that each H_i alone is sufficient to reconstruct the full admissibility certificate for the terminal collapse:

$$\mathcal{A}(H_i) \supseteq \mathcal{A}_{\text{needed}}(H) \quad \text{for each } i.$$

Definition 16.2 (Erasure-Correcting History)**.** An *erasure-correcting history* with parameters (n, k) encodes k independent certificate paths through n events, such that any k of the n events are sufficient to recover the full admissibility structure.

Theorem 16.3 (Error Correction via Redundant Refusal)**.** *If a history H with redundancy factor $k \geq 2$ has $m < k/2$ events corrupted or erased, the correct admissibility structure can be recovered from the remaining events.*

Proof. With redundancy k and fewer than $k/2$ erasures, at least $k - k/2 > k/2$ redundant paths remain intact. Majority vote over the surviving paths recovers the correct admissibility judgement. □

Typed Error Propagation

Definition 16.4 (Error Type)**.** An *error type* is a type of the form $\mathsf{Refused}(T, r)$ where r is a proof-carrying reason. Error types carry the evidence of inadmissibility as first-class type-level information.

Definition 16.5 (Error Propagation Rule)**.**

$$\frac{\Gamma \vdash t : \mathsf{Refused}(T_1, r) \quad \Gamma, x : T_1 \vdash b : T_2}{\Gamma \vdash \mathsf{handle}(t, x.b) : T_2 \vee \mathbf{??}T_2, r)} \quad \text{[T-HANDLE]}$$

The handler b may produce a value of T_2 or propagate a refusal. The propagated refusal carries the original reason r, preserving the full provenance chain.

Theorem 16.6 (Error Preservation)**.** *If $\Gamma \vdash t : \mathsf{Refused}(T, r)$ and there is no handler in scope for r, then r propagates to the nearest enclosing context that can handle $\mathbf{??}\cdot, r)$.*

Proof. By the typing rules, $\mathsf{Refused}(T, r)$ types only propagate upward through [T-HANDLE] if the handler does not fully resolve them. Since every unresolved $\mathsf{Refused}(T, r)$ in a well-typed program must eventually reach a handler or become the return type of the program, propagation terminates at the program boundary with a documented reason. □

Recovery as Certified Continuation

Definition 16.7 (Recovery Certificate)**.** A *recovery certificate* for an error $\mathsf{Refused}(T, r)$ at world state W is a term $\mathsf{recover} : \mathsf{Refused}(T, r) \to \mathsf{Admissible}(T')$ that transforms a documented inadmissibility into an admissible continuation of type T'.

Theorem 16.8 (Soundness of Recovery)**.** *If $\mathsf{recover} : \mathsf{Refused}(T, r) \to \mathsf{Admissible}(T')$ is a recovery certificate and $\Gamma \vdash t : \mathsf{Refused}(T, r)$, then $\Gamma \vdash \mathsf{recover}(t) : \mathsf{Admissible}(T')$. Furthermore, the history of $\mathsf{recover}(t)$ contains the full provenance chain from the original refusal to the recovery, ensuring that the recovery is not a silent erasure of the error.*

Proof. By application of the recovery function type and [T-POP] applied to the result. The history contains the original $\mathsf{Refuse}(x, r)$ event followed by the recovery events. No history erasure is possible since histories grow monotonically. □

Remark 16.9. Recovery in Spherepop is the computational analogue of repair theory: it restores access to a continuation after documented inadmissibility, but unlike silent error suppression, it preserves the full record of what went wrong and how it was addressed. The recovery certificate is a note—specifically a Repair Note—in the formal sense of the note taxonomy.

Part VII

Deep Mathematical Structure

Chapter 17

The Observation Limit

The Reviewer's Challenge

Spherepop does not abolish the state illusion. It makes the quotient explicit, parameterised, and reversible whenever the underlying history is retained.

A careful reader will notice a tension. The document claims to resolve the state illusion—the loss of historical information caused by projecting history to state. Yet every output of a Spherepop program is a collapse: a value derived by applying a rule to a history. The programmer observes the collapsed value, not the history. Is Spherepop not simply relocating the state illusion?

This chapter addresses that challenge. We prove the No Direct Observation Theorem, which shows the challenge identifies a mathematical necessity, and then distinguish precisely between state erasure (which Spherepop eliminates) and state illusion (which no computational system can eliminate).

The No Direct Observation Theorem

Definition 17.1 (History-Faithful Observation)**.** An observation map $\phi : \mathcal{H} \to O$ is *history-faithful* if it is injective: $\phi(H_1) = \phi(H_2) \Rightarrow H_1 = H_2$.

Theorem 17.2 (No Direct Observation)**.** *Let O be any set with $|O| < |\mathcal{H}|$. Then every observation map $\phi : \mathcal{H} \to O$ is non-injective: there exist distinct $H_1 \neq H_2$ with $\phi(H_1) = \phi(H_2)$. Consequently, no computational system with a finite observable space can directly verify the history it constructed.*

Proof. Since $\mathcal{H} = \bigcup_{n \geq 0} E^n$ is countably infinite and $|O| < |\mathcal{H}|$, by the pigeonhole principle ϕ cannot be injective. Therefore there exist $H_1 \neq H_2$ with $\phi(H_1) = \phi(H_2)$. □

Remark 17.3. Theorem 7.1 is a fundamental limit. It says: every observation is a quotient. This is not a contingent limitation that better engineering can over-

come. It is a cardinality constraint. A system that made its observation map injective—that recorded every distinction in full history space—would itself be a history.

Erasure versus Illusion

Definition 17.4 (State Erasure)**.** A computational system suffers *state erasure* if it irrecoverably discards the history H after computing $\sigma(H)$. After execution, H is unavailable and only $\sigma(H)$ remains.

Definition 17.5 (State Illusion)**.** A computational system presents a *state illusion* to its programmers if they can only observe $\sigma(H)$ and not H directly. By Theorem 7.1, some state illusion is unavoidable for any non-trivial observable space.

Proposition 17.6 (What Spherepop Eliminates)**.** *Spherepop eliminates state erasure: the history H is retained in* `world.history` *and is available to any observer. Spherepop cannot and does not eliminate the state illusion: the programmer's primary observable is $c(H)$, not H, so distinct histories that agree under c remain indistinguishable from the perspective of that observation.*

The correct claim is therefore: Spherepop does not abolish the state illusion. It makes the quotient explicit, parameterised, and history-preserving.

Rule Refinement as Disambiguation

Definition 17.7 (Rule Refinement Order)**.** Collapse rule c_1 is *finer than* c_2, written $c_1 \preceq c_2$, if $H_1 \sim_{c_1} H_2 \Rightarrow H_1 \sim_{c_2} H_2$. The Identity rule c_I is the finest rule; the trivial rule $c_\top$ mapping all histories to a single point is the coarsest.

Proposition 17.8 (Disambiguation by Refinement)**.** *Let $H_1 \sim_c H_2$ (histories indistinguishable under rule c) and $H_1 \neq H_2$. Then there exists a finer rule $c' \preceq c$ such that $c'(H_1) \neq c'(H_2)$. In the limit, c_I always disambiguates since $c_I(H) = H$ exactly.*

Proof. Take $c' = c_I$: since $H_1 \neq H_2$ and $c_I(H) = H$, we have $c_I(H_1) = H_1 \neq H_2 = c_I(H_2)$. □

Chapter 18

Category Theory as Engine

The History-Observation Adjunction

Definition 18.1 (History Construction Functor)**.** Let **Terms** be the category whose objects are Spherepop terms and whose morphisms are reduction sequences. Define G : **Terms** $\to$ **Hist** by $G(t) = I(t).\mathsf{history}$.
Definition 18.2 (Observation Functor)**.** For a fixed collapse rule c, define $\mathcal{O}_c$: **Hist** $\to$ **Obs**$_c$ by $\mathcal{O}_c(H) = c(H) = F_c(H)$.
Theorem 18.3 (History-Observation Adjunction)**.** *For the Identity rule c_I, there is a natural bijection*

$$\textbf{Hist}(G(t), H) \cong \textbf{Terms}(t, G^{-1}(H))$$

whenever $G^{-1}(H)$ is defined. This is the categorical statement of Replay Equivalence: the interpreter and compiler produce histories that are morphisms in **Hist***.*

Collapse Rules Form a Lattice

Proposition 18.4 (Observational Lattice)**.** *The set of collapse rules ordered by refinement $c_1 \preceq c_2$ forms a complete lattice. The meet of $\{c_i\}$ is the finest rule at least as fine as all; the join is the rule generated by the union of equivalence relations.*

Proof. The meet and join are defined by the standard lattice operations on equivalence relations. Completeness follows because arbitrary intersections and generated equivalence relations of equivalence relations are again equivalence relations. □

Sheaf-Theoretic Semantics

Definition 18.5 (Observational Site)**.** Fix collapse rules $\{c_i\}$ and observable states $\{o_i\}$. The *observational site* $\mathcal{O}$ is the category whose objects are observational regions $\{c_i^{-1}(o_i)\}$ and whose morphisms are inclusions $c_j^{-1}(o_j) \hookrightarrow c_i^{-1}(o_i)$ when the former is contained in the latter.

Proposition 18.6 ($\mathcal{H}$ is a Sheaf)**.** *The history presheaf $\mathcal{H}$ on $\mathcal{O}$—defined by $\mathcal{H}(U) = \{H \in \mathcal{H} : H \in U\}$—satisfies the sheaf condition: compatible local histories glue uniquely to global histories.*

Proof. Compatibility means event sequences agree on their shared prefixes. Two compatible event sequences agreeing on all overlaps determine a unique concatenation. □

Proposition 18.7 (State Illusion as Non-Injectivity of Sections)**.** *A collapse rule c exhibits state illusion if and only if the observable sheaf $\mathcal{O}_c$ fails to determine a unique global section of $\mathcal{H}$. There exist distinct $H_1, H_2 \in \mathcal{H}(\mathcal{H})$ with the same section in $\mathcal{O}_c(\mathcal{H})$.*

The sheaf structure ties the local (what one observer sees under one rule) to the global (the full history that all rules project from). Sheaf cohomology measures the obstruction to lifting local observations to global histories—which is precisely the observational entropy.

Chapter 19

CLIO Projections and Active Geodesic Inference

Collapse Rules as CLIO Projections

Definition 19.1 (CLIO Projection)**.** A CLIO projection $\pi : X \to M$ maps a representation space X to a manifold M of observational strata. In the Spherepop setting: $X = \mathcal{H}$ (history space), $M = O_c$ (observable state space under rule c), and $\pi = F_c$ (the collapse functor).

Definition 19.2 (Observational Entropy)**.** For a collapse rule c and observable state $o \in O_c$, the *observational entropy* is

$$S_c(o) = \log |c^{-1}(o)|.$$

Theorem 19.3 (Entropy Monotonicity Under Rule Refinement)**.** *If $c_1 \preceq c_2$ (c_1 finer), then for every $o \in O_{c_1}$:*

$$S_{c_1}(o) \leq S_{c_2}(c_2(c_1^{-1}(o))).$$

Finer rules have smaller or equal fibre sizes.

Proof. Since $c_1 \preceq c_2$, every $\sim_{c_1}$-class is contained in a $\sim_{c_2}$-class. Therefore $|c_1^{-1}(o)| \leq |c_2^{-1}(c_2(H))|$ for any $H \in c_1^{-1}(o)$. □

Remark 19.4. Observational entropy $S_c(o) = \log |c^{-1}(o)|$ is exactly the CLIO quantity $S_\pi(m) = \log \mathrm{Vol}(\pi^{-1}(m))$ in the discrete Spherepop setting. Spherepop is a discrete computational instance of the CLIO projection program.

The Santoro–Waghmare–Panaretos Unification

Theorem 19.5 (Representation Changes Topology)**.** *Let $\rho_1, \rho_2 : X \to Y$ be two representation maps with $\tau_{\rho_1} \neq \tau_{\rho_2}$. Then there exist $x_1, x_2 \in X$ such that $x_1 \sim_{\rho_1} x_2$*

but $x_1 \not\sim_{\rho_2} x_2$, or vice versa. The two representations disagree on whether x_1 and x_2 are the same object.

Proof. If $\tau_{\rho_1} \neq \tau_{\rho_2}$, there exists an open set $U \in \tau_{\rho_1} \setminus \tau_{\rho_2}$. Then $U = \rho_1^{-1}(V)$ for some $V \subseteq Y$, but $U \neq \rho_2^{-1}(W)$ for any W. Therefore there exist x_1, x_2 with $\rho_1(x_1) = \rho_1(x_2)$ but $\rho_2(x_1) \neq \rho_2(x_2)$. □

Remark 19.6. This theorem is the general form of the observation that representations reorganise topology. It applies uniformly to: (1) the Santoro–Waghmare–Panaretos kernel embedding, which converts metrically close distributions into topologically singular Gaussian measures; and (2) Spherepop collapse rules, where refinement moves history pairs between equivalence classes.

In the kernel case, the topological change is forced by the Feldman–Hájek theorem. In the collapse case, the topological change is chosen: the programmer selects the rule, thereby choosing which distinctions are topologically visible. Both are instances of the same structural operation.

Active Geodesic Inference

The Spherepop framework connects to a broader theory of intelligent systems as trajectory-maintaining structures.

Definition 19.7 (Active Geodesic Inference)**.** An intelligent system performs *active geodesic inference* if it actively shapes its configuration space through energetic and entropic constraints, forming low-action trajectories through possibility space rather than following predetermined state transitions.

Proposition 19.8 (Spherepop as Active Geodesic Substrate)**.** *Spherepop's irreversible, scope-based semantics enforce history-sensitivity and prevent incoherent superposition by construction. Pop operations commit to specific trajectories; Refuse operations document inadmissible branches; the history preserves the full record of path selection. Spherepop is therefore a natural substrate for implementing active geodesic inference: the history is the geodesic; the admissibility set is the constraint field; collapse is the observation that seals a trajectory as complete.*

Definition 19.9 (Formal Intelligence)**.** Intelligence is the capacity of a system to remain within a family of dynamically admissible, low-action histories by actively reshaping its configuration space's geometry. This definition extends across scales, encompassing learning within a lifetime, evolution across generations, and reasoning within an episode.

Theorem 19.10 (Intelligence as Note Infrastructure)**.** *An intelligent system with formal intelligence in the above sense is necessarily a note-producing system: it constructs and maintains Capture, Prospective, and Repair notes as part of the mechanism by which it reshapes its configuration space. Notes are the mechanism by which intelligent systems install future reachability from present positions.*

Proof. A system that maintains low-action histories across time must preserve information about its past trajectories (Capture Notes), commit to future trajectories (Prospective Notes), and recover from trajectory interruptions (Repair Notes). Each of these is a note class in the formal taxonomy. Therefore any formally intelligent system produces notes as a structural necessity, not a contingent feature. □

Part VIII

Civilization as Note Infrastructure

Chapter 20

MEM|8 and the Cognitive Substrate

Memory as Event Structure

The MEM|8 architecture treats memory not as isolated symbol storage but as structured pattern access where retrieval depends on preserving relationships, locality, and accessibility. Memory is not a database; it is a cognitive substrate whose operations are structurally similar to Spherepop's history operations.

Proposition 20.1 (MEM|8 as Spherepop Instance)**.** *The MEM|8 architecture is a biological instance of the Spherepop framework:*

MEM|8 Layer	*Spherepop Note Class*
Episodic layer	*Capture notes*
Procedural layer	*Repair notes*
Semantic layer	*Coordination notes*
Indexical layer	*Collapse notes*
Theoretical layer	*Generative notes*

Remark 20.2. A notebook is a miniature MEM|8. A Git repository is a miniature MEM|8. An archive is a miniature MEM|8. Each stores structured trajectories organized to make reconstruction possible. The difference between a good archive and a poor one is not quantity of material stored but quality of access structure: how well the collapse notes and coordination notes that organize the collection preserve the reachability of trajectories within it.

Chapter 21

Repair Theory and Civilizational Collapse

Repair as Pre-emptive Refusal

Repair Theory holds that repair is the restoration of access to a trajectory that has been interrupted. The connection to Spherepop is not metaphorical but structural: repair is the execution of a Repair Note that was created in anticipation of the failure.

Theorem 21.1 (Note Taxonomy as Failure Taxonomy)**.** *The seven note classes correspond bijectively to seven classes of anticipated trajectory failure:*

Note Class	*Anticipated Failure*
Capture	*Experiential collapse*
Prospective	*Intention-execution gap*
Repair	*Procedural interruption*
Coordination	*Coordination failure*
Refusal	*Distinction collapse*
Generative	*Inferential poverty*
Collapse	*Navigational failure*

Civilizational Scale

Definition 21.2 (Civilizational Note Infrastructure)**.** Let civilization Z possess note infrastructure $\mathcal{N} = \{N_1, \ldots, N_k\}$. The *collective reachability* is

$$R(Z) = \bigcup_{i=1}^{k} \mathcal{C}(N_i),$$

where $\mathcal{C}(N_i)$ is the set of continuations preserved or enabled by N_i.

Theorem 21.3 (Civilizational Collapse)**.** *The destruction of note infrastructure $\mathcal{N}$*

decreases collective reachability even when underlying truths remain unchanged:

$$R(Z \setminus \mathcal{N}) < R(Z).$$

A civilization that loses its note infrastructure does not become ignorant. It becomes unable to reach its own knowledge.

Proof. By definition, $R(Z)$ includes all continuations preserved by elements of $\mathcal{N}$. If $\mathcal{N}$ is destroyed, no element contributes to collective reachability. Since some continuations are accessible only through $\mathcal{N}$, it follows that $R(Z \setminus \mathcal{N}) < R(Z)$. The underlying facts are not destroyed, but the paths that made them accessible are. □

Technology	Reachability Horizon	Dominant Note Classes
Speech	Minutes to hours	Coordination, Capture
Story	Decades	Capture, Generative
Writing	Centuries	All classes
Printing	Populations	Coordination, Refusal
Digital storage	Replication	All classes, at scale
Version control	Modification history	Repair, Capture
Network	Distribution	Coordination, Generative

Each note technology expands the reachability horizon by addressing failure modes the preceding technology could not prevent. Writing prevents the death of witnesses. Printing prevents the destruction of individual copies. Version control prevents the loss of developmental history through modification. The history of civilization is the history of expanding note infrastructure.

Part IX

Reconstruction, Refusal, and Repair

Chapter 22

Observational Entropy and Reconstruction Cost

From Observation to Reconstruction

The preceding chapters established that every collapse rule $c : \mathcal{H} \to O_c$ induces an observational entropy $S_c(o) = \log |c^{-1}(o)|$. This quantity measures the size of the fibre associated with an observation. A collapse with small observational entropy preserves many distinctions between histories. A collapse with large observational entropy identifies many histories and destroys information. The natural next question is not merely how many histories are hidden, but how difficult it is to reconstruct the correct one.

Definition 22.1 (Reconstruction Cost)**.** Let $o \in O_c$. The *reconstruction cost* of o is

$$\mathcal{R}(o) = \mathbb{E}\big[C(H) \mid H \in c^{-1}(o)\big]$$

where $C(H)$ denotes the computational effort required to identify history H from among the candidates in the fibre.

Definition 22.2 (Uniform Reconstruction)**.** A reconstruction procedure is *uniform* if every history in $c^{-1}(o)$ is considered equiprobable given only the observation o.

Theorem 22.3 (Entropy Lower Bound on Reconstruction)**.** *For any uniform reconstruction procedure,*

$$\mathcal{R}(o) \geq k \cdot S_c(o)$$

for some positive constant k depending only on the cost model C.

Proof. The fibre $c^{-1}(o)$ contains $N = |c^{-1}(o)|$ candidate histories. Any reconstruction procedure must distinguish among these alternatives. A decision tree distinguishing N equally probable alternatives requires at least $\log_2 N$ binary decisions. Each decision has a positive cost bounded below by $k > 0$. Since $S_c(o) = \log N$, the expected reconstruction effort satisfies $\mathcal{R}(o) \geq k \log N =$

$k \cdot S_c(o)$. □

Corollary 22.4. *Observational entropy is a lower bound on reconstruction difficulty. Notes that reduce observational entropy are therefore notes that reduce reconstruction cost.*

Remark 22.5. This theorem explains precisely why notes have value in terms the continuation theory can measure. A note N reduces $|c^{-1}(o)|$ by introducing additional distinctions that collapse cannot erase. Reducing the fibre size reduces $S_c(o)$ and hence reduces $\mathcal{R}(o)$. The note acts as a compression-resistant continuation preserver: it encodes information that the observational collapse would otherwise destroy, lowering the expected work required to recover the original history.

Continuation Value and Note Leverage

Definition 22.6 (Continuation Value of an Observation)**.** The *continuation value* of observation o is

$$V(o) = \frac{1}{1 + \mathcal{R}(o)}.$$

Observations with large fibres have low continuation value because the original history is expensive to recover. Observations with small fibres have high continuation value. The note taxonomy from Part V can now be given a quantitative characterization: a note N is valuable to the extent that it increases $V(o)$ for observations o it affects, which is equivalent to decreasing $\mathcal{R}(o)$, which is equivalent to decreasing $S_c(o)$.

Theorem 22.7 (Note Leverage as Entropy Reduction)**.** *The reachability leverage $\Lambda(N)$ of a note (Definition 2.3) is bounded below by the entropy reduction it induces:*

$$\Lambda(N) \geq \frac{k}{C_{\text{create}}(N)} \sum_o [S_c(o) - S_c(o \mid N)],$$

where $S_c(o \mid N)$ denotes the entropy of observation o when the note N is available.

Proof. By the Entropy Lower Bound theorem, entropy reduction lower-bounds reconstruction cost reduction. Summing over affected observations and dividing by creation cost gives the claimed bound on leverage. □

The Reconstruction Complexity Hierarchy

Different collapse rules induce different reconstruction complexity classes.

Definition 22.8 (Tractable Reconstruction)**.** A collapse rule c has *tractable reconstruction* if $\mathcal{R}(o)$ is bounded by a polynomial in the size of the history alphabet for all $o \in O_c$.

Proposition 22.9. *The Identity rule c_I has reconstruction complexity $O(1)$: given the observation (which equals the history), reconstruction is immediate. The LastWrite rule c_{LW} has reconstruction complexity exponential in the number of symbols, since the fibre contains all histories consistent with the last-write state. The Accumulate rule c_{acc} has reconstruction complexity equal to the multinomial coefficient $\binom{|events|}{n_1,\ldots,n_k}$ where n_i is the count for symbol i.*

Chapter 23

The Geometry of Refusal

Refusal as Geometric Excision

Refusal was introduced in Part II as a documented declaration of inadmissibility. A deeper interpretation emerges when admissibility is viewed as a region of possibility space. The admissibility set $\mathcal{A}(H)$ is not merely a set of available symbols; it is a geometric region in possibility space, and each refusal excises a portion of that region.

Definition 23.1 (Admissibility Region)**.** The *admissibility region* after history H is $\mathcal{A}(H) = \Omega_0 \setminus \{x \mid \exists r.\, \mathsf{Refuse}(x, r) \in H\}$.

Definition 23.2 (Refusal Region)**.** A refusal reason r determines a *forbidden subset* $U_r \subseteq \Omega_0$. A refusal $\mathsf{Refuse}(x, r)$ excises x from the admissibility region under the constraint r.

Definition 23.3 (Refusal Action)**.** Applying $\mathsf{Refuse}(x, r)$ produces the new admissibility region

$$\mathcal{A}(H') = \mathcal{A}(H) \setminus \{x\}.$$

When r is a constraint that applies to a family of symbols $U_r = \{x : r(x) = \mathsf{true}\}$, the excision generalizes to $\mathcal{A}(H') = \mathcal{A}(H) \setminus U_r$.

Refusal is therefore a geometric excision operator: it removes regions from possibility space. Multiple refusals compose to produce an increasingly constrained admissibility region, a process that can be analyzed in terms of the topology of the resulting space.

Refusal Monotonicity

Theorem 23.4 (Refusal Monotonicity)**.** *For every refusal event,* $\mathcal{A}(H') \subseteq \mathcal{A}(H)$.

Proof. By construction, $\mathcal{A}(H') = \mathcal{A}(H) \setminus U_r$ for some $U_r \subseteq \Omega_0$. Removing a subset

cannot enlarge the region. Hence $\mathcal{A}(H') \subseteq \mathcal{A}(H)$. □

Corollary 23.5 (Irreversibility of Refusal)**.** *Once a symbol x is refused, it cannot be re-admitted without replacing the history. Since histories grow monotonically, refusal is irreversible within a computation.*

Refusal Curvature

Definition 23.6 (Local Admissibility Curvature)**.** For a point x in possibility space with neighbourhood $N(x)$, the *local admissibility curvature* is

$$\kappa_A(x) = \frac{|N(x) \cap \mathcal{A}|}{|N(x)|}.$$

A curvature of 1 indicates full local admissibility; a curvature of 0 indicates a refusal singularity.

Definition 23.7 (Refusal Singularity)**.** A point x is a *refusal singularity* if $\kappa_A(x) = 0$: every trajectory through x is inadmissible.

Proposition 23.8 (Curvature Monotonicity Under Refusal)**.** *Repeated refusals create regions of non-increasing admissibility curvature: if $H' = H +\!+ [\mathsf{Refuse}(x, r)]$ then $\kappa_{A'}(y) \leq \kappa_A(y)$ for all $y \in U_r$.*

Proof. Each refusal removes admissible trajectories from U_r. Therefore $|N(y) \cap \mathcal{A}'| \leq |N(y) \cap \mathcal{A}|$ for $y \in U_r$, which implies $\kappa_{A'}(y) \leq \kappa_A(y)$. □

Definition 23.9 (Admissibility Field)**.** The function $\kappa_A : \Omega_0 \to [0, 1]$ is the *admissibility field* of the computation. Its level sets $\{x : \kappa_A(x) \geq \lambda\}$ for threshold λ define the λ-admissible regions of possibility space.

Remark 23.10. The admissibility field κ_A is the discrete computational analogue of the smooth admissibility field in the RSVP framework. The xylomorphic criterion $\lambda < 1$ from RSVP corresponds to $\kappa_A(x) < 1$ in the discrete setting: a trajectory is admissible at level λ when its local admissibility curvature exceeds λ. The RSVP continuum limit of the Spherepop admissibility field is therefore obtained by taking the lattice spacing to zero in the discrete geometry.

The Galois Connection Between History Growth and Admissibility Contraction

Theorem 23.11 (History–Admissibility Galois Connection)**.** *There is a Galois connection between the lattice of history extensions and the lattice of admissibility regions,*

ordered by the relation:

$$H \leq H' \iff H \text{ is a prefix of } H'$$

on histories and $\mathcal{A} \leq \mathcal{A}'$ iff $\mathcal{A} \subseteq \mathcal{A}'$ on admissibility regions. The Galois connection is given by:

$$\Phi(H) = \mathcal{A}(H) \quad \textit{and} \quad \Psi(\mathcal{A}) = \{H : \mathcal{A}(H) \supseteq \mathcal{A}\}.$$

Proof. We verify the Galois condition: $H \leq H'$ implies $\Phi(H) \geq \Phi(H')$ (order reversal). By Refusal Monotonicity, if H' extends H by any sequence of events, then $\mathcal{A}(H') \subseteq \mathcal{A}(H)$, that is $\Phi(H') \leq \Phi(H)$. Conversely, $\mathcal{A} \geq \mathcal{A}'$ (i.e. $\mathcal{A} \supseteq \mathcal{A}'$) implies $\Psi(\mathcal{A}) \leq \Psi(\mathcal{A}')$ (shorter histories needed to maintain the larger admissibility set). The adjunction conditions $H \leq \Psi(\Phi(H))$ and $\mathcal{A} \leq \Phi(\Psi(\mathcal{A}))$ follow from the definitions. □

Remark 23.12. The Galois connection formalizes the intuition that history growth and admissibility contraction are dual processes. As a computation proceeds (history grows), the space of admissible futures contracts. The Galois connection makes this duality precise: every increase in historical commitment corresponds to a decrease in future admissibility, and vice versa.

Chapter 24

Notes as Externalized Adjoints

The Reconstruction Gap

Collapse maps histories into observations, losing information. Notes partially reverse this process by preserving information that collapse discards. The question is: can this reversal be made precise in category-theoretic terms?

The answer is yes, and the result gives the note theory its deepest mathematical formulation. Notes are partial right adjoints to collapse functors.

The Note Reconstruction Functor

Definition 24.1 (Note Reconstruction Functor)**.** Given a collapse functor $\mathcal{O}_c$: **Hist** $\to$ **Obs**$_c$, a *note reconstruction functor* is a functor $\mathcal{N}$: **Obs**$_c$ $\to$ **Hist** that assigns to each observable state o a canonical history $\mathcal{N}(o) \in c^{-1}(o)$.

Definition 24.2 (Adjoint Note)**.** A note is *adjoint to collapse* if

$$\mathbf{Obs}_c(\mathcal{O}_c(H), o) \cong \mathbf{Hist}(H, \mathcal{N}(o))$$

naturally in H and o.

Theorem 24.3 (Notes as Partial Right Adjoints)**.** *Every successful note acts as a partial right adjoint to a collapse operation, in the sense that it provides a reconstruction map from observations back toward histories on the subdomain where reconstruction succeeds.*

Proof. Let N be a note and c a collapse rule such that N preserves information discarded by c on some subdomain $D \subseteq O_c$. Define $\mathcal{N}(o) = H_N(o)$ where $H_N(o)$ is the canonical history recovered using note N from observation o. For $o \in D$, the map $\mathbf{Hist}(H, \mathcal{N}(o)) \to \mathbf{Obs}_c(\mathcal{O}_c(H), o)$ given by $f \mapsto \mathcal{O}_c \circ f$ is an isomorphism: every history morphism into $\mathcal{N}(o)$ projects to a unique observable mor-

phism, and conversely, since N provides the additional data needed to lift observable morphisms to history morphisms on D. This is the universal property of a right adjoint, restricted to D. □

Corollary 24.4. *The statement "notes are anti-collapse artifacts" is equivalent to saying that notes approximate right adjoints to collapse functors. The richer the note's continuation volume, the larger the subdomain D on which the adjunction holds.*

Examples of Note Adjunctions

The note-as-adjoint formulation unifies the taxonomy from Part V.

A *bookmark* is a right adjoint to the collapse of a URL into a browsing context: it recovers the specific location from an otherwise unlabeled observation of web content.

A *mathematical proof* is a right adjoint to the collapse of a theorem into a bare assertion: it recovers the inferential trajectory from the terminal claim.

Source code is a right adjoint to the collapse of executable behavior into observable outputs: it recovers the computational trajectory from the input-output specification.

A *map* is a right adjoint to geographical collapse: it recovers spatial structure from positional observations that contain no direction or distance information.

In every case, the note $\mathcal{N}$ reconstructs distinctions erased by the collapse $\mathcal{O}_c$, satisfying the adjunction on the subdomain where reconstruction is possible.

The Completeness Spectrum

Definition 24.5 (Note Completeness)**.** A note N is *complete* for collapse rule c if $\mathcal{N}$ extends to a global right adjoint: $\mathcal{O}_c \dashv \mathcal{N}$ on all of $\mathbf{Obs}_c$. N is *partial* if the adjunction holds only on a proper subdomain.

Proposition 24.6. *No finite note is complete for a collapse rule that identifies infinitely many distinct histories. Complete notes for finite history spaces exist and correspond to observationally complete collapse rules* (*Proposition 19.3 of Part VII*).

Proof. If c identifies infinitely many histories, then $|c^{-1}(o)| = \infty$ for some o. A finite note contains finite information and can therefore eliminate at most finitely many ambiguities per observation. Hence it cannot provide a complete right adjoint on all of $\mathbf{Obs}_c$. □

Chapter 25

The Topology of Repair

Repair as a Morphism in the History Category

Repair Theory holds that repair is the restoration of access to a trajectory that has been interrupted. Within the Spherepop framework, repair can be formalized as a special class of morphism in the history category.

Definition 25.1 (Repair Morphism)**.** A *repair morphism* from history H to history H' is a map $\rho : H \to H'$ such that $c(H) = c(H')$ for some designated collapse rule c: the observable behavior is preserved while the historical structure is altered.

Definition 25.2 (Repair Equivalence)**.** Two histories are *repair equivalent*, written $H \sim_R H'$, if there exists a collapse rule c and a repair morphism $\rho : H \to H'$ such that $c(H) = c(H')$.

Theorem 25.3 (Repair Equivalence Theorem)**.** *Repair preserves observable reachability while modifying historical structure: if $H \sim_R H'$ then every continuation observable under c from H is also observable from H', yet H and H' may differ as elements of $\mathcal{H}$.*

Proof. Since $c(H) = c(H')$, the two histories are in the same observational equivalence class. Every continuation whose reachability depends only on the observable state $c(H)$ is therefore equally reachable from H'. However, since $H \neq H'$ as sequences of events, their internal structure differs. The repair has altered the history while preserving its observable footprint. □

Examples of Repair Classes

The repair morphism formulation unifies several classes of real-world operation.

Software patches replace a sequence of code events producing a buggy observable behavior with a different sequence producing the correct observable behavior. The patched and unpatched code share the same input-output specification

(the collapse rule is the black-box I/O function) but have different development histories.

Biological healing replaces a damaged tissue history with a regenerated tissue history. The organism's observable function is preserved (the collapse rule is the phenotypic function) while the internal molecular history changes.

Engineering maintenance replaces a worn component history with a replacement component history. The system's operational behavior is preserved (the collapse rule is the performance specification) while the material history differs.

Institutional reform replaces a history of governance decisions with a revised sequence. The institution's external behavior and mandate are preserved (the collapse rule is the institutional mandate) while the decision history changes.

All four are repair morphisms in the formal sense.

Repair Cohomology

Not every interrupted trajectory can be repaired. The obstruction to repair is a topological invariant.

Definition 25.4 (Repair Defect)**.** The *repair defect* of histories H and H' is $\Delta_R(H, H') = d_{\mathbf{Hist}}(H, H')$ where $d_{\mathbf{Hist}}$ measures the minimum edit distance in the history category (minimum number of event replacements needed to convert H into a history with the same observable state as H').

Definition 25.5 (Repair Cohomology Class)**.** When no repair morphism exists between H and a target observable state o (that is, $c^{-1}(o) \cap \mathsf{Reachable}(H) = \varnothing$), the obstruction is a non-trivial element of the *repair cohomology* $H^1_R(\mathcal{H}, c)$—the first cohomology group of the history space with coefficients in the collapse sheaf.

Proposition 25.6 (Trivial Cohomology Implies Universal Repairability)**.** *If $H^1_R(\mathcal{H}, c) = 0$, then every history can be repaired to any observationally equivalent target. The absence of cohomological obstructions corresponds to a globally connected repair graph.*

Remark 25.7. Repair cohomology provides a rigorous mathematical notion of irreparability. A system with non-trivial repair cohomology has histories from which no repair morphism can reach certain observable states, regardless of the effort invested. In practice, this corresponds to: cryptographic irreversibility (a hash cannot be repaired to recover the preimage), biological cell death (a necrotic cell cannot be repaired to a living state), and data loss without redundancy (a destroyed archive without backup admits no repair).

Chapter 26

The Noun Fallacy Revisited

Objects as Stable Quotients

Throughout this monograph we have argued that histories are primary and states are derived. The present chapter makes this precise by giving a formal account of how objects—the apparently primitive entities of everyday ontology—arise from the collapse of histories.

Definition 26.1 (Persistent Quotient)**.** An equivalence class $[H]_c$ under collapse rule c is *persistent* if $c(H_t) = c(H_{t+\Delta})$ for all admissible extensions $H_{t+\Delta}$ of H_t in a neighborhood of the current trajectory. That is, the observable state does not change under small perturbations of the computation.

Definition 26.2 (Object)**.** An *object* in the Spherepop ontology is a persistent quotient:

$$\mathsf{Object} = [H]_c \quad \text{where } [H]_c \text{ is persistent.}$$

Object Emergence

Theorem 26.3 (Object Emergence Theorem)**.** *Stable observational equivalence classes emerge as objects to observers who access histories only through collapse rules.*

Proof. An observer accesses $\mathcal{H}$ only through c. Repeated observation of the same equivalence class produces observations $c(H_{t_1}) = c(H_{t_2}) = \cdots = o$. The observer, receiving identical observations across time, infers an invariant entity. This entity is $[H]_c$. The observer treats the persistent equivalence class as an object because it is behaviorally indistinguishable from one: it produces the same observable state under every observation the rule c permits. □

The Noun Fallacy

Definition 26.4 (Noun Fallacy)**.** The *noun fallacy* is the error of treating a persistent quotient $[H]_c$ as an ontologically primitive object—as if $[H]_c$ existed independently of the history H and the collapse rule c that produced it.

Corollary 26.5. *Every noun is a collapse note.*

Proof. A noun compresses a family of trajectories into a single retrievable symbol. It therefore performs the Collapse operation on the histories it subsumes. The symbol is the collapsed observable; the suppressed histories are the fibres of the collapse. A noun is precisely a Collapse Note in the sense of Chapter 13. □

Example 26.6. The word *tree* is a collapse note. It collapses the family of all developmental, ecological, metabolic, and evolutionary histories of tree-shaped organisms into a single navigable token. The fibre $c_{\text{tree}}^{-1}(\text{“tree”})$ is the set of all histories that produce observable states classifiable as trees. The noun *tree* makes this fibre navigable without requiring traversal of any particular history within it.

Example 26.7. The word *program* is a collapse note for computational history. It collapses the set of all source texts, development decisions, debugging sessions, and revision histories that produced a given executable into a single noun. The Spherepop framework aims to reverse this collapse: to make the history that the noun conceals explicitly accessible.

Reality as Reachable History

The preceding analysis permits a precise statement of the monograph's deepest claim.

Theorem 26.8 (Reality as Reachability)**.** *Under the Spherepop ontology:*

1. *Histories are primary: they are the fundamental objects of computation, cognition, and civilization.*
2. *Observations are quotients: they are equivalence classes of histories under collapse rules.*
3. *Objects are persistent quotients: stable equivalence classes that observers reify as things.*
4. *Notes preserve distinctions across quotients: they are artifacts that resist specific collapse operations.*

5. *Repair restores reachability after interruption: it constructs repair morphisms between interrupted and target histories.*
6. *Civilization constructs note infrastructure: it builds distributed systems for preserving continuations across generations.*
7. *Reality is a structured space of reachable histories, not a collection of primitive objects. Objects are what histories look like after sufficient collapse.*

Proof. Items (1)–(6) follow from the formal definitions and theorems developed in Parts II through VIII. Item (7) is the synthesis: given that objects are persistent quotients (Definition 29.2) and quotients are derived from histories (Proposition 2.2), objects depend ontologically on histories. The space of objects is therefore a derived structure over the primary space of histories. □

Part X

Collapse as Orthogonal Projection

Chapter 27

The Hilbert Space of Histories

Motivation

Throughout the preceding chapters, collapse has been treated combinatorially: a collapse rule $c : \mathcal{H} \to O_c$ maps histories to equivalence classes, and observational entropy counts fibre sizes. This combinatorial treatment is adequate for finite history spaces, but it leaves the continuous and probabilistic dimensions of collapse undeveloped.

The Hilbert space formulation of conditional expectation, developed in probability theory through the work of Kolmogorov and later geometrized in the L^2 framework, provides exactly the continuous analogue of Spherepop collapse. The central thesis of this part is:

> Every collapse rule introduced in previous chapters may be viewed as an orthogonal projection operator. In finite settings, collapse appears combinatorial. In probabilistic settings, collapse becomes orthogonal projection in a Hilbert space. The two treatments are the same operation at different levels of abstraction.

The L^2 History Space

Definition 27.1 (L^2 History Space)**.** Let $(\Omega, \mathcal{F}, P)$ be a probability space where Ω is the set of all histories, $\mathcal{F}$ is a σ-algebra over Ω, and P is a probability measure over histories. The L^2 *history space* is

$$L^2(\mathcal{F}) = \{X : \Omega \to \mathbb{R} \mid \mathbb{E}[X^2] < \infty\}$$

with inner product $\langle X, Y \rangle = \mathbb{E}[XY]$.

Remark 27.2. Random variables $X \in L^2(\mathcal{F})$ are functions on the space of histories. They measure observable quantities of histories—durations, counts of

specific event types, the time of the last Pop, the total weight of admissibility certificates—in a way that respects the probability structure over histories.

Definition 27.3 (Observational Subspace)**.** Let $\mathcal{G} \subseteq \mathcal{F}$ be a sub-σ-algebra representing an observational framework. The *observational subspace* is

$$L^2(\mathcal{G}) = \{X \in L^2(\mathcal{F}) \mid X \text{ is } \mathcal{G}\text{-measurable}\}.$$

$L^2(\mathcal{G})$ is a closed linear subspace of $L^2(\mathcal{F})$. It consists of exactly those random variables whose values can be determined from the information available in $\mathcal{G}$—the distinctions that the observational framework $\mathcal{G}$ makes visible.

Orthogonal Projection as Collapse

Theorem 27.4 (Conditional Expectation as Orthogonal Projection)**.** *For any $X \in L^2(\mathcal{F})$ and any sub-σ-algebra $\mathcal{G} \subseteq \mathcal{F}$, the conditional expectation $\mathbb{E}[X \mid \mathcal{G}]$ is the unique element of $L^2(\mathcal{G})$ that minimizes $\mathbb{E}[(X - Z)^2]$ over all $Z \in L^2(\mathcal{G})$. Equivalently, $\mathbb{E}[X \mid \mathcal{G}]$ is the orthogonal projection of X onto $L^2(\mathcal{G})$.*

Proof. The closed subspace $L^2(\mathcal{G})$ has a unique nearest point to X by the Hilbert space projection theorem. This nearest point $\hat{X}$ satisfies the orthogonality condition: $\langle X - \hat{X}, Z\rangle = 0$ for all $Z \in L^2(\mathcal{G})$. Expanding: $\mathbb{E}[(X - \hat{X})Z] = 0$ for all $\mathcal{G}$-measurable Z. This is precisely Kolmogorov's characterization of the conditional expectation. By the uniqueness of the projection, $\hat{X} = \mathbb{E}[X \mid \mathcal{G}]$. □

Definition 27.5 (Spherepop Interpretation of Conditional Expectation)**.** Under the Spherepop identification:

$$\begin{aligned}
\text{History space } \mathcal{H} &\longleftrightarrow L^2(\mathcal{F}) \\
\text{Observation space } O_c &\longleftrightarrow L^2(\mathcal{G}) \\
\text{Collapse functor } F_c &\longleftrightarrow \mathbb{E}[\cdot \mid \mathcal{G}] \\
\text{Collapse rule } c &\longleftrightarrow \text{Sub-}\sigma\text{-algebra } \mathcal{G} \\
\text{Observational entropy } S_c(o) &\longleftrightarrow \text{Variance of residual}
\end{aligned}$$

The Orthogonality Condition as State Illusion

Theorem 27.6 (Orthogonality as State Illusion)**.** *The residual $X - \mathbb{E}[X \mid \mathcal{G}]$ is orthogonal to every element of $L^2(\mathcal{G})$:*

$$\big\langle X - \mathbb{E}[X \mid \mathcal{G}],\, Z \big\rangle = 0 \quad \textit{for all } Z \in L^2(\mathcal{G}).$$

The discarded component is completely invisible from within the observational framework that produced the collapse.

Proof. By the projection theorem, $X - \mathbb{E}[X \mid \mathcal{G}]$ is perpendicular to $L^2(\mathcal{G})$. Expanding: $\mathbb{E}[(X - \mathbb{E}[X \mid \mathcal{G}])Z] = \mathbb{E}[XZ] - \mathbb{E}[\mathbb{E}[X \mid \mathcal{G}] \cdot Z]$. By the definition of conditional expectation, $\mathbb{E}[\mathbb{E}[X \mid \mathcal{G}] \cdot Z] = \mathbb{E}[XZ]$ for all $\mathcal{G}$-measurable Z. Therefore the difference is zero. □

Remark 27.7. This theorem is the Hilbert-space version of the state illusion. The observer with access only to $L^2(\mathcal{G})$ cannot detect the residual component $X - \mathbb{E}[X \mid \mathcal{G}]$: every measurement they can make is orthogonal to it, yielding zero correlation. The observer perceives only $\mathbb{E}[X \mid \mathcal{G}]$ and is unaware that a residual exists. This is precisely the state illusion: the observer perceives a single observable state while the underlying history contains information invisible to their measurement apparatus.

Chapter 28

The Algebraic Laws as Geometry

The Tower Property as Nested Collapse

The Tower Property of conditional expectation—$\mathbb{E}[\mathbb{E}[X \mid \mathcal{G}_2] \mid \mathcal{G}_1] = \mathbb{E}[X \mid \mathcal{G}_1]$ when $\mathcal{G}_1 \subseteq \mathcal{G}_2$—is a geometric theorem about nested projections.

Theorem 28.1 (Tower Property as Nested Projection)**.** *If $\mathcal{G}_1 \subseteq \mathcal{G}_2 \subseteq \mathcal{F}$, then $L^2(\mathcal{G}_1) \subseteq L^2(\mathcal{G}_2)$, and*

$$\mathbb{E}\big[\mathbb{E}[X \mid \mathcal{G}_2] \mid \mathcal{G}_1\big] = \mathbb{E}[X \mid \mathcal{G}_1].$$

Proof. Since $\mathcal{G}_1 \subseteq \mathcal{G}_2$, every $\mathcal{G}_1$-measurable function is $\mathcal{G}_2$-measurable. Therefore $L^2(\mathcal{G}_1) \subseteq L^2(\mathcal{G}_2)$. Let $\hat{X}_2 = \mathbb{E}[X \mid \mathcal{G}_2]$ be the projection of X onto $L^2(\mathcal{G}_2)$. The residual $X - \hat{X}_2$ is orthogonal to $L^2(\mathcal{G}_2)$ and hence to the subspace $L^2(\mathcal{G}_1) \subseteq L^2(\mathcal{G}_2)$. Therefore $\mathbb{E}[\hat{X}_2 \mid \mathcal{G}_1]$ is the projection of $\hat{X}_2$ onto $L^2(\mathcal{G}_1)$. But the projection of X onto $L^2(\mathcal{G}_1)$ equals the projection of $\hat{X}_2 + (X - \hat{X}_2)$ onto $L^2(\mathcal{G}_1)$, and the residual contributes zero (being orthogonal to $L^2(\mathcal{G}_1)$). Hence $\mathbb{E}[X \mid \mathcal{G}_1] = \mathbb{E}[\hat{X}_2 \mid \mathcal{G}_1]$. □

Remark 28.2 (Spherepop Interpretation)*.* The Tower Property is the Hilbert-space version of the Spherepop result on Collapse Functor Composition (Theorem 20.2): applying a coarser collapse after a finer one is equivalent to applying the coarser collapse directly. In both settings, the coarser observational framework erases the distinctions preserved by the finer one. The geometry makes the "why" transparent: projecting twice onto nested subspaces is the same as projecting once onto the smaller subspace.

The Observational Lattice as Inclusion of Subspaces

Proposition 28.3 (Lattice-Subspace Correspondence)**.** *The lattice of collapse rules ordered by refinement ($c_1 \preceq c_2$ iff c_1 is finer) corresponds to the lattice of closed sub-*

spaces of $L^2(\mathcal{F})$ ordered by inclusion: finer rules correspond to larger (more informative) subspaces.

$$c_1 \preceq c_2 \quad \Longleftrightarrow \quad L^2(\mathcal{G}_{c_1}) \supseteq L^2(\mathcal{G}_{c_2}).$$

Proof. A finer rule c_1 preserves more distinctions, hence its observational subspace $L^2(\mathcal{G}_{c_1})$ contains more measurable functions—it is a larger subspace. A coarser rule c_2 preserves fewer distinctions—its subspace $L^2(\mathcal{G}_{c_2})$ is smaller. The inclusion reversal corresponds to the duality between refinement (more information) and subspace containment (larger space of representable functions). □

Eve's Law as the Pythagorean Theorem

The Law of Total Variance (Eve's Law) is not an algebraic identity but a Pythagorean theorem in $L^2(\mathcal{F})$.

Theorem 28.4 (Eve's Law as Pythagorean Theorem)**.**

$$\mathrm{Var}(X) = \mathbb{E}[\mathrm{Var}(X \mid \mathcal{G})] + \mathrm{Var}(\mathbb{E}[X \mid \mathcal{G}]).$$

Proof. Center X by writing $\tilde{X} = X - \mathbb{E}[X]$. Decompose:

$$\tilde{X} = \underbrace{(X - \mathbb{E}[X \mid \mathcal{G}])}_{\text{residual}} + \underbrace{(\mathbb{E}[X \mid \mathcal{G}] - \mathbb{E}[X])}_{\text{explained}}.$$

The residual lies in $L^2(\mathcal{G})^\perp$ (it is the orthogonal complement of the projection). The explained component lies in $L^2(\mathcal{G})$. These two vectors are orthogonal. Therefore by the Pythagorean theorem:

$$\|\tilde{X}\|^2 = \|X - \mathbb{E}[X \mid \mathcal{G}]\|^2 + \|\mathbb{E}[X \mid \mathcal{G}] - \mathbb{E}[X]\|^2.$$

Taking expectations: $\|\tilde{X}\|^2 = \mathrm{Var}(X)$; $\|X - \mathbb{E}[X \mid \mathcal{G}]\|^2 = \mathbb{E}[\mathrm{Var}(X \mid \mathcal{G})]$ by the Tower Property; and $\|\mathbb{E}[X \mid \mathcal{G}] - \mathbb{E}[X]\|^2 = \mathrm{Var}(\mathbb{E}[X \mid \mathcal{G}])$. □

Remark 28.5 (Spherepop Interpretation). Eve's Law decomposes total uncertainty into two orthogonal components:

- $\mathbb{E}[\mathrm{Var}(X \mid \mathcal{G})]$: *hidden uncertainty*—the expected residual uncertainty that remains after the observational collapse $\mathcal{G}$ has occurred. This corresponds to the fibre entropy $S_c(o)$ summed over observations: the irreducible ambiguity within each equivalence class.

- $\mathrm{Var}(\mathbb{E}[X \mid \mathcal{G}])$: *visible uncertainty*—the variance of the observable itself. This corresponds to the variation between equivalence classes.

The total variance $\mathrm{Var}(X)$ is the Pythagorean sum of these two orthogonal sources. The state illusion hides the first component from the observer who accesses the history only through $\mathcal{G}$.

Chapter 29

Sigma-Algebras as Observational Frameworks

Information as Geometry

The L^2 framework gives a geometric account of information. An observational framework $\mathcal{G}$ is not merely a collection of measurable sets; it is a geometric constraint on what can be known. The subspace $L^2(\mathcal{G})$ consists of exactly those history-valued functions whose values can be determined given the information in $\mathcal{G}$.

Theorem 29.1 (Sigma-Algebra as Information Constraint)**.** *A sub-σ-algebra $\mathcal{G} \subseteq \mathcal{F}$ corresponds to an observational framework that makes visible exactly the distinctions in $L^2(\mathcal{G})$. The following are equivalent:*

1. *X is $\mathcal{G}$-measurable.*
2. *$X \in L^2(\mathcal{G})$.*
3. *The value of X is completely determined by the observation framework $\mathcal{G}$.*
4. *$\mathbb{E}[X \mid \mathcal{G}] = X$ (projecting X onto its own subspace leaves it unchanged).*

Proof. (1) $\Leftrightarrow$ (2) by definition of $L^2(\mathcal{G})$. (2) $\Leftrightarrow$ (3) because $\mathcal{G}$-measurable functions are exactly those determined by $\mathcal{G}$-information. (3) $\Leftrightarrow$ (4): if $X \in L^2(\mathcal{G})$, projecting X onto $L^2(\mathcal{G})$ yields X itself (the projection of a vector already in a subspace is the vector). □

Collapse Rules and Sub-Sigma-Algebras

Theorem 29.2 (Collapse Rules Are Sub-Sigma-Algebras)**.** *Every Spherepop collapse rule $c : \mathcal{H} \to O_c$ determines a sub-σ-algebra $\mathcal{G}_c$ of the history σ-algebra $\mathcal{F}$:*

$$\mathcal{G}_c = \{c^{-1}(U) \mid U \subseteq O_c\}.$$

The collapse functor F_c corresponds to the conditional expectation operator $\mathbb{E}[\cdot \mid \mathcal{G}_c]$.

Proof. The collection $\{c^{-1}(U) \mid U \subseteq O_c\}$ is closed under complementation and countable unions (since c is measurable), hence forms a σ-algebra. It is a sub-σ-algebra of $\mathcal{F}$ because every $c^{-1}(U)$ is a union of fibers of c, hence measurable in $\mathcal{F}$. The conditional expectation $\mathbb{E}[X \mid \mathcal{G}_c]$ then picks out the component of X that varies only across the equivalence classes of c, which is exactly what the collapse functor F_c does at the combinatorial level. □

The Radon-Nikodym Theorem as Universal Note

The Radon-Nikodym theorem, which grounds the measure-theoretic definition of conditional expectation, has a natural interpretation in the Spherepop framework.

Theorem 29.3 (Radon-Nikodym as Universal Reconstruction)**.** *Let μ and ν be probability measures on $(\Omega, \mathcal{F})$ with $\nu \ll \mu$ (absolute continuity). Then there exists a unique $\mathcal{F}$-measurable function $f = d\nu/d\mu$ such that $\nu(A) = \int_A f \, d\mu$ for all $A \in \mathcal{F}$.*

Remark 29.4 (Spherepop Interpretation). The Radon-Nikodym derivative $d\nu/d\mu$ is a note in the Spherepop sense: it preserves the information needed to reconstruct the measure ν from the reference measure μ. Without $d\nu/d\mu$, an agent who knows only μ cannot determine ν. With $d\nu/d\mu$—a measurable function, hence a Generative Note in the L^2 setting—the reconstruction is complete. The Radon-Nikodym theorem is therefore a universal reconstruction theorem: it guarantees the existence of the minimal note needed to recover one measure from another, whenever such recovery is possible.

Variance as Continuous Observational Entropy

Theorem 29.5 (Variance as Continuous Fibre Entropy)**.** *The conditional variance* $\mathrm{Var}(X \mid \mathcal{G})$ *is the continuous analogue of the discrete observational entropy* $S_c(o) = \log |c^{-1}(o)|$*:*

$$\mathrm{Var}(X \mid \mathcal{G}) = \mathbb{E}\big[(X - \mathbb{E}[X \mid \mathcal{G}])^2 \mid \mathcal{G}\big]$$

measures the expected squared deviation of X from its projection onto $\mathcal{G}$, which is the expected information content of the residual component invisible to the observational framework.

Proof. The residual $X - \mathbb{E}[X \mid \mathcal{G}]$ is the component of X that the framework $\mathcal{G}$ cannot see. Its conditional variance $\mathbb{E}[(X - \mathbb{E}[X \mid \mathcal{G}])^2 \mid \mathcal{G}]$ measures the expected squared magnitude of this invisible component, given the visible component.

This is the continuous analogue of counting the histories in a fibre: instead of $\log |c^{-1}(o)|$ discrete histories, we have a continuous measure of the spread of histories within the fibre. □

Remark 29.6 (Unification). This theorem completes the bridge between the discrete Spherepop collapse theory and the continuous L^2 conditional expectation theory. The dictionary is:

Discrete Spherepop	Continuous L^2
History space $\mathcal{H}$	$L^2(\mathcal{F})$
Collapse rule c	Sub-σ-algebra $\mathcal{G}$
Collapse functor F_c	Conditional expectation $\mathbb{E}[\cdot \mid \mathcal{G}]$
Observational entropy $S_c(o)$	Conditional variance $\mathrm{Var}(X \mid \mathcal{G})$
Fibre $c^{-1}(o)$	Fiber over $\mathcal{G}$-atom
State illusion	Orthogonality of residual
Note (anti-collapse)	Radon-Nikodym derivative
Tower Property	Nested projection theorem
Eve's Law	Pythagorean theorem in L^2

Part XI

Open Problems and Future Directions

Chapter 30

The Inverse Problem and Observational Completeness

When Does Observational Equivalence Imply History Equivalence?

The deepest open question in the Spherepop framework is the converse of the collapse functor: when does $c(H_1) = c(H_2)$ imply $H_1 = H_2$?

Definition 30.1 (Observational Completeness)**.** A collapse rule c is *observationally complete* if it is injective: $c(H_1) = c(H_2) \Rightarrow H_1 = H_2$.

Definition 30.2 (Collapse Kernel)**.** The *kernel* of a collapse rule c is

$$\ker(c) = \{(H_1, H_2) \in \mathcal{H} \times \mathcal{H} \mid c(H_1) = c(H_2)\}.$$

Observational completeness holds iff $\ker(c) = \{(H, H) \mid H \in \mathcal{H}\}$.

Proposition 30.3 (Completeness Results for Canonical Rules)**.** *The Identity rule c_I is observationally complete. The LastWrite rule c_{LW} is not: $[pop(x), pop(y)]$ and $[pop(y), pop(x), pop(y)]$ are c_{LW}-equivalent but distinct. The Accumulate rule c_{acc} is not: $[pop(x), pop(y)]$ and $[pop(y), pop(x)]$ have identical accumulate-states.*

Definition 30.4 (Inverse-Problem Fibre)**.** For $o \in O_c$, the *fibre* of o is $c^{-1}(o) = \{H \in \mathcal{H} \mid c(H) = o\}$.

Characterising fibres for general rules, and determining when a fibre contains a unique history, is the primary open mathematical problem in the framework.

Open Problems

Distributed Spherepop Runtimes

The current World is single-threaded. A distributed Spherepop runtime would coordinate multiple Worlds, each with their own histories, under a global con-

sistency constraint. The consistency constraint must be expressed in terms of history equivalence under a shared collapse rule—not value consistency. The event sourcing and CRDT literatures provide partial models; the Spherepop-specific challenge is the certified-refusal structure across distributed agents.

Collapse Functor Composition

The four canonical collapse rules are special cases of a more general family of functors F_c : **Hist** $\to$ **Obs**$_c$. A theory of functor composition would characterise which combinations of rules are valid—when $F_{c_2} \circ F_{c_1}$ is itself a valid collapse functor—and would classify the algebraic structure of the space of collapse rules.

Dependent Process Types in Full Generality

The current Process$(T_1 \rightsquigarrow T_2)$ type is first-order. A dependent process type $\Pi(x : T_1)$. Process$(x, f(x))$ would allow the type of the output to depend on the specific value consumed. This is the natural extension of dependent type theory into the process calculus setting and would enable precise typing of programs that branch on their inputs.

Admissibility Lattice Completeness

The current type system uses a Boolean admissibility structure. The full theory requires a graded admissibility lattice with continuous admissibility levels, connecting to the RSVP xylomorphic criterion $\lambda < 1$ and the admissibility geometry program. Completeness theorems for the graded type system would characterize exactly which trajectories are admissible at each level.

Proof-Carrying Refusal in Full Generality

The current RefusalReason is a vocabulary. A complete proof-carrying refusal system would make RefusalReason a proof term: evidence that can be type-checked independently of the computation that produced it. This connects to the theory of certified refusal and would enable mechanized verification of inadmissibility claims.

Chapter 31

The Coda: Implementation as Philosophical Argument

The implementation did not merely realize the theory; it exposed which distinctions the theory was still hiding.

The central claim of this monograph is not that Spherepop is a good programming language, though we believe it is an interesting one. The central claim is that taking a philosophical position seriously enough to implement it is a form of philosophical argument.

Four implementation discoveries were not predicted by the initial framework. They were forced by the attempt to mechanize it.

The original Refuse(Symbol) gestured at documented inadmissibility. It took implementation pressure to reveal that a symbol alone documents nothing: the reason is the document.

The original notion of observable state gestured at the quotient structure. It took `World::observe` to force the question: observable under what rule?

The original type checker gestured at world-relative knowledge. It took the free-variable conflict to force the question: which world are we assuming?

The original correctness criterion gestured at semantic preservation. It took the compiler test to force the question: preservation of what, exactly?

Each discovery reveals a distinction the philosophical framework was gesturing at without making precise. Only in the attempt to mechanize—to force the philosophy to compile—did the gestures become definitions. This is the deepest lesson of ontology-driven language design. Philosophy can identify the right questions. Only implementation can force the right precision.

Spherepop is the algebra of continuations.

Notes are the physical realization of that algebra.

Civilization is the attempt, always incomplete, always under threat, to build a note infrastructure adequate to the continuation needs of the species that requires it.

Part XII

The Calculus of Constructions

Chapter 32

Why Spherepop Needs the Calculus of Constructions

The Limits of Simple Types

The type system developed in Part IV is powerful enough to certify admissibility, document refusals, and seal observations. But it is limited in one fundamental respect: types cannot depend on values. The type of the output of a function cannot depend on what the input actually is. This limitation becomes acute in several situations that arise naturally in Spherepop.

Consider a function that reads a file and returns a proof that the file was successfully read. The output type should mention the specific file that was read, not just the abstract type of files. Or consider a GC pass that returns a history with only live events: the output type should certify that specific inadmissibility certificates from the input are preserved. Simple types cannot express either of these conditions.

The Calculus of Constructions (CoC), introduced by Coquand and Huet [10], is the natural remedy. It is a typed lambda calculus in which types may depend on terms, terms may be types, and the distinction between levels collapses into a single universe hierarchy. In CoC, the type of a returned value can carry the full specification of what was computed.

The Pure Type System Perspective

The Calculus of Constructions belongs to the family of *pure type systems* (PTSs), which unify the handling of terms and types through a single syntactic category.
Definition 32.1 (Sorts and Pure Type System)**.** A *pure type system* is determined by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where $\mathcal{S}$ is a set of *sorts*, $\mathcal{A}$ is a set of *axioms* $s_1 : s_2$, and $\mathcal{R}$ is a set of *rules* (s_1, s_2, s_3) governing when Π-types can be formed.
Definition 32.2 (Calculus of Constructions)**.** The Calculus of Constructions is

the PTS with:

$$\mathcal{S} = \{*, \Box\}$$
$$\mathcal{A} = \{* : \Box\}$$
$$\mathcal{R} = \{(*, *, *),\ (*, \Box, \Box),\ (\Box, *, *),\ (\Box, \Box, \Box)\}$$

where $*$ is the sort of ordinary types and $\Box$ is the sort of kinds (types of types).

The four rules in $\mathcal{R}$ permit:

1. $(*, *, *)$: functions from terms to terms (ordinary functions).
2. $(*, \Box, \Box)$: functions from terms to types (dependent types).
3. $(\Box, *, *)$: functions from types to terms (polymorphism).
4. $(\Box, \Box, \Box)$: functions from types to types (type operators).

The full power of CoC is that all four are available simultaneously. This is what makes it capable of expressing the entire type-theoretic apparatus needed for Spherepop's admissibility certificates.

The Spherepop-CoC Correspondence

The Spherepop type system is naturally embedded into CoC.

Definition 32.3 (CoC Embedding of Spherepop Types)**.** The embedding $[\![\cdot]\!] : \mathsf{SphType} \to \mathsf{CoCTerm}$ is defined by:

$$\begin{aligned}
[\![\mathsf{Unit}]\!] &= \top_{\mathsf{CoC}} \\
[\![\mathsf{Never}]\!] &= \bot_{\mathsf{CoC}} \\
[\![\mathsf{Admissible}(T)]\!] &= \Sigma(h : \mathsf{History}).\, \mathsf{Admissible}(h, [\![T]\!]) \\
[\![\mathsf{Refused}(T, r)]\!] &= \Sigma(h : \mathsf{History}).\, \mathsf{Refused}(h, [\![T]\!], r) \\
[\![\mathsf{Collapsed}(T, c)]\!] &= \Sigma(h : \mathsf{History}).\, \mathsf{Collapsed}(h, [\![T]\!], c) \\
[\![\Pi(x : T_1).T_2]\!] &= \Pi(x : [\![T_1]\!]).[\![T_2]\!] \\
[\![\mathsf{Process}(T_1 \rightsquigarrow T_2)]\!] &= \Pi(h_1 : [\![T_1]\!]).\, \Sigma(h_2 : [\![T_2]\!]).\, \mathsf{BindDep}(h_1, h_2)
\end{aligned}$$

The Σ-types in the embedding are crucial: they pair the value with a proof that the history supporting it has the required structure. The admissibility certificate is not merely a type annotation but a proof term that can be independently verified.

Theorem 32.4 (Embedding Soundness)**.** *If $\Gamma \vdash t : T$ in the Spherepop type system, then $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]$ in CoC.*

Proof. By structural induction on the Spherepop typing derivation. Each typing rule maps to a valid CoC derivation:

[T-POP]: The embedding sends $\mathsf{pop}(t)$ to a pair $\langle h, \pi \rangle$ where π is a proof of $\mathsf{Admissible}(h, [\![T]\!])$. This is a valid Σ-introduction in CoC.

[T-REFUSE]: The embedding sends $\mathsf{refuse}(t, r)$ to a pair $\langle h, \pi_r \rangle$ where π_r proves $\mathsf{Refused}(h, [\![T]\!], r)$. Again a valid Σ-introduction.

[T-COLLAPSE]: Collapse requires a proof of admissibility as input and produces a proof of the collapsed type. The CoC derivation uses Σ-elimination on the admissibility proof followed by Σ-introduction for the collapsed type.

The remaining cases follow by induction. □

Chapter 33

Implementing CoC for Spherepop

The Universe Hierarchy

A full implementation of CoC for Spherepop requires a universe hierarchy to avoid Russell-type paradoxes. We adopt a predicative hierarchy.

Definition 33.1 (Universe Hierarchy)**.** Define sorts $\mathsf{Type}_0, \mathsf{Type}_1, \mathsf{Type}_2, \ldots$ with axioms $\mathsf{Type}_i : \mathsf{Type}_{i+1}$ and cumulative coercions $\mathsf{Type}_i \hookrightarrow \mathsf{Type}_{i+1}$.

In practice, most Spherepop typing occurs at Type_0 (the sort of ordinary program types). The proof-carrying refusal reasons inhabit Type_1 (propositions about Type_0 terms). The collapse rules inhabit Type_1 as functions from histories to observable states.

Listing 33.1: Universe levels in the Spherepop type checker

```
pub enum Universe {
    Type(usize),   // Type_0, Type_1, ...
    Prop,          // proof-irrelevant propositions
}

pub enum Term {
    Sort(Universe),
    Var(Symbol),
    App(Box<Term>, Box<Term>),
    Lam(Symbol, Box<Term>, Box<Term>),  // λx:A.b
    Pi(Symbol, Box<Term>, Box<Term>),   // Πx:A.B
    Sigma(Symbol, Box<Term>, Box<Term>),// Σx:A.B
    Pair(Box<Term>, Box<Term>),         // (a, b)
    Fst(Box<Term>),                     //
    Snd(Box<Term>),                     //
    // Spherepop primitives
    Pop(Box<Term>),
```

```
    Refuse(Box<Term>, RefusalReason),
    Bind(Box<Term>, Box<Term>),
    Collapse(Box<Term>, CollapseRule),
    // Admissibility universe
    Admissible(Box<Term>),
    Refused(Box<Term>, RefusalReason),
    Collapsed(Box<Term>, CollapseRule),
}
```

The Type Checker

A CoC type checker for Spherepop must implement bidirectional type checking: type inference for introduction forms and type checking for elimination forms.

Definition 33.2 (Bidirectional Typing)**.** Bidirectional typing uses two judgements:

- $\Gamma \vdash t \Rightarrow T$ (*synthesis*): the type T of t is inferred from t alone.
- $\Gamma \vdash t \Leftarrow T$ (*checking*): t is checked against the expected type T.

Listing 33.2: Bidirectional type checker core

```
pub fn synthesize(ctx: &Context, term: &Term)
    -> Result<Term, TypeError>
{
    match term {
        Term::Var(x) => ctx.lookup(x),
        Term::App(f, a) => {
            let f_ty = synthesize(ctx, f)?;
            match whnf(&f_ty) {
                Term::Pi(x, a_ty, b_ty) => {
                    check(ctx, a, &a_ty)?;
                    Ok(subst(b_ty, x, a))
                }
                _ => Err(TypeError::NotAFunction)
            }
        }
        Term::Fst(p) => {
            let p_ty = synthesize(ctx, p)?;
            match whnf(&p_ty) {
                Term::Sigma(_, a, _) => Ok(*a),
                _ => Err(TypeError::NotAPair)
```

```
            }
        }
        Term::Pop(t) => {
            let t_ty = synthesize(ctx, t)?;
            Ok(Term::Admissible(Box::new(t_ty)))
        }
        Term::Refuse(t, r) => {
            let t_ty = synthesize(ctx, t)?;
            Ok(Term::Refused(Box::new(t_ty), r.clone()))
        }
        Term::Collapse(t, c) => {
            let t_ty = synthesize(ctx, t)?;
            match whnf(&t_ty) {
                Term::Admissible(inner) =>
                    Ok(Term::Collapsed(inner, c.clone())),
                _ => Err(TypeError::
                   CollapseRequiresAdmissible)
            }
        }
        _ => Err(TypeError::CannotSynthesize)
    }
}

pub fn check(ctx: &Context, term: &Term, ty: &Term)
    -> Result<(), TypeError>
{
    match (term, whnf(ty)) {
        (Term::Lam(x, _, b), Term::Pi(y, a, b_ty)) => {
            let ctx2 = ctx.extend(x, &a);
            check(&ctx2, b, &subst(&b_ty, y, &Term::Var(x)))
        }
        (Term::Pair(a, b), Term::Sigma(x, a_ty, b_ty)) => {
            check(ctx, a, &a_ty)?;
            let a_val = eval(a);
            check(ctx, b, &subst(&b_ty, x, &a_val))
        }
        _ => {
            let t_ty = synthesize(ctx, term)?;
            if definitionally_equal(&t_ty, ty) { Ok(()) }
```

```
            else { Err(TypeError::TypeMismatch) }
        }
    }
}
```

Definitional Equality and Normalization

CoC type checking requires deciding definitional equality: two types are definitionally equal if they reduce to the same normal form.

Definition 33.3 (Weak Head Normal Form)**.** A term is in *weak head normal form* (WHNF) if it is not a beta-redex at the head: it is a lambda, a Pi, a Sigma, a sort, a variable, or an application whose function is in WHNF.

Listing 33.3: WHNF reduction for Spherepop-CoC

```
pub fn whnf(term: &Term) -> Term {
    match term {
        Term::App(f, a) => {
            match whnf(f) {
                Term::Lam(x, _, b) => whnf(&subst(&b, &x, a))
                    ,
                f_whnf => Term::App(Box::new(f_whnf),
                                    a.clone())
            }
        }
        Term::Fst(p) => {
            match whnf(p) {
                Term::Pair(a, _) => whnf(&a),
                p_whnf => Term::Fst(Box::new(p_whnf))
            }
        }
        Term::Snd(p) => {
            match whnf(p) {
                Term::Pair(_, b) => whnf(&b),
                p_whnf => Term::Snd(Box::new(p_whnf))
            }
        }
        // Spherepop reductions
        Term::Pop(t) => Term::Pop(Box::new(whnf(t))),
```

```
        Term::Collapse(t, c) => {
            Term::Collapse(Box::new(whnf(t)), c.clone())
        }
        t => t.clone()
    }
}
```

Theorem 33.4 (Normalization of Spherepop-CoC)**.** *The Spherepop-CoC type system is strongly normalizing: every well-typed reduction sequence terminates.*

Proof sketch. The Calculus of Constructions is strongly normalizing by the reducibility candidates argument of Girard [16]. The Spherepop extensions Pop, Refuse, Bind, and Collapse are all introduction forms that do not introduce new beta-redexes. They reduce only by unwrapping their arguments to WHNF. The combined system inherits strong normalization from CoC by a standard extension argument: the new rules add no new reduction paths that could create infinite sequences, since the underlying lambda calculus reductions are already terminating. □

Identity Types and History Equality

CoC extended with identity types (Martin-Löf type theory [24]) gives Spherepop a native notion of proof-level history equality.

Definition 33.5 (Identity Type)**.** For any type A and terms $a, b : A$, the *identity type* $a =_A b$ is a type whose inhabitants are proofs that a and b are definitionally equal. The sole constructor is $\mathsf{refl}_a : a =_A a$.

Definition 33.6 (History Identity Type)**.** For histories $H_1, H_2 : \mathsf{History}$, the type $H_1 =_{\mathsf{History}} H_2$ is inhabited precisely when the histories are event-for-event identical. This is the type-level statement of Replay Equivalence: a term of type $I(p).\mathsf{history} =_{\mathsf{History}} V(C(p)).\mathsf{history}$ is a formal proof that the compiler is correct.

Listing 33.4: History identity in the type system

```
pub fn verify_replay_equivalence(
    prog: &Term,
    interp_world: &World,
    compiled_world: &World,
) -> Result<(), ProofError> {
    // Produce a proof of history identity
    if interp_world.history == compiled_world.history {
```

```
        Ok(())  // refl witnesses the identity
    } else {
        // Construct a counterexample witness
        let diff = history_diff(
            &interp_world.history,
            &compiled_world.history
        );
        Err(ProofError::HistoryMismatch(diff))
    }
}
```

The Curry-Howard Correspondence for Spherepop

The Curry-Howard correspondence—that proofs are programs and propositions are types—takes a specific form in Spherepop.

Proposition 33.7 (Spherepop Curry-Howard Correspondence)**.** *The following identifications hold:*

Logical Notion	*Program Notion*	*Spherepop Realization*
Proposition	*Type*	$T : \mathsf{Type}_0$
Proof	*Term*	$t : T$
Implication	*Function type*	$\Pi(x : A).B$
Conjunction	*Pair type*	$\Sigma(x : A).B$
Admissibility	*Certificate type*	$\mathsf{Admissible}(T)$
Refutation	*Refusal type*	$\mathsf{Refused}(T, r)$
Observation	*Collapsed type*	$\mathsf{Collapsed}(T, c)$
History identity	*Replay equivalence*	$H_1 =_{\mathcal{H}} H_2$

Part XIII

The BNF Grammar as Universal Notation

Chapter 34

Why Grammar Beats Circuits and Code

The Problem of Notation

Every computational formalism needs a notation: a way of writing down the objects it studies. The choice of notation is not merely aesthetic. It determines what can be expressed without ambiguity, what can be processed mechanically, what can be read without specialized tools, and what can be implemented independently of any particular hardware or software substrate.

Three dominant notations for computation exist: programming language syntax, circuit diagrams, and formal grammars. The thesis of this chapter is that Backus-Naur Form (BNF) grammar, or its extensions, is the superior notation for specifying computational behavior, and that Spherepop's use of BNF is not a convenience but a theoretical necessity.

A Brief History of BNF

Backus-Naur Form was introduced by Peter Naur in the ALGOL 60 report [26]. Its purpose was to give a precise, implementation-independent specification of a programming language's syntax. Before BNF, programming language grammars were described in English prose: ambiguous, difficult to parse mechanically, and dependent on shared human conventions.

BNF replaces prose with a minimal formal apparatus: nonterminals (enclosed in angle brackets), terminals (written as themselves), a production arrow ($::=$), and alternation ($|$). From these four elements, any context-free language can be specified.

Definition 34.1 (BNF Grammar)**.** A *Backus-Naur Form grammar* is a tuple $G = (N, T, P, S)$ where:

- N is a finite set of *nonterminals*.

- T is a finite set of *terminals* (disjoint from N).
- P is a finite set of *productions* of the form $A ::= \alpha$ where $A \in N$ and $\alpha \in (N \cup T)^*$.
- $S \in N$ is the *start symbol*.

Extended BNF (EBNF) adds optional elements ($[\cdot]$), repetition ($\{\cdot\}$), and grouping, but adds no expressive power beyond context-free languages.

BNF Versus Circuit Diagrams

Circuit diagrams are the standard notation for hardware computation. They have significant advantages: they are visually intuitive, they make data flow explicit, and they can be directly compiled to hardware. But they have several critical disadvantages relative to BNF.

Implementation Dependence

A circuit diagram is inherently tied to a specific level of abstraction. A gate diagram for a full adder specifies AND, OR, and NOT gates in a specific topology. This specification cannot be directly used at a different level of abstraction without redrawing. A BNF grammar for the same operation specifies the structure independently of implementation:

Listing 34.1: BNF for binary addition (implementation-independent)

```
<bit>      ::= "0" | "1"
<addend>   ::= <bit> | <bit> <addend>
<sum>      ::= <addend> "+" <addend>
<carry>    ::= <bit>
<result>   ::= <carry> <sum>
```

This grammar specifies the structure of binary addition without committing to gates, lookup tables, carry-lookahead circuits, or any other implementation. Any implementation that accepts the grammar's language is correct by definition.

Human Readability

Circuit diagrams require specialized visual parsing. They cannot be read aloud, stored in plain text, or processed by standard text tools. A BNF grammar is written in ASCII (or Unicode), can be stored in a version control repository, diffed,

searched, and processed by any text processor. It is read left-to-right, top-to-bottom, in the same direction as natural language.

Theorem 34.2 (BNF Readability Advantage)**.** *For any grammar G specifying a language L, the BNF representation of G requires only the characters of $N \cup T$ plus the BNF metasymbols $\{::=, |, \langle, \rangle\}$. The circuit representation of the same computation requires a graphical medium that cannot be stored or transmitted as plain text.*

Proof. BNF is defined over a finite alphabet. Its productions are strings over that alphabet. A circuit diagram, by contrast, represents spatial relationships (the topology of the circuit) that cannot be serialized as a string without loss of the spatial information that makes the diagram legible. The best-known lossless serialization of a circuit diagram is a netlist, which is itself a grammar-like notation—a point in favor of the grammar approach. □

Compositional Structure

BNF grammars compose naturally. If G_1 specifies language L_1 and G_2 specifies L_2, the grammar for $L_1 \cdot L_2$ (concatenation) is formed by adding a production $S ::= S_1\, S_2$ and combining the production sets. This compositional structure is the foundation of modular language design.

Proposition 34.3 (Grammar Compositionality)**.** *If $G_1 = (N_1, T_1, P_1, S_1)$ and $G_2 = (N_2, T_2, P_2, S_2)$ with $N_1 \cap N_2 = \varnothing$, then:*

$$G_1 \cdot G_2 = (N_1 \cup N_2 \cup \{S\},\ T_1 \cup T_2,\ P_1 \cup P_2 \cup \{S ::= S_1\, S_2\},\ S)$$
$$G_1 \mid G_2 = (N_1 \cup N_2 \cup \{S\},\ T_1 \cup T_2,\ P_1 \cup P_2 \cup \{S ::= S_1 \mid S_2\},\ S)$$

These operations correspond to sequential composition and choice in Spherepop.

BNF Versus Programming Language Syntax

BNF is more fundamental than any particular programming language syntax because it is what programming languages are defined in. A programming language's syntax is a grammar; BNF is the metalanguage for grammars.

But the comparison is not merely definitional. BNF grammars have concrete advantages as computational specifications over program text.

Ambiguity is Detectable

A BNF grammar can be checked for ambiguity (multiple parse trees for the same string). An English prose description of a language cannot. A program in any conventional language can be syntactically unambiguous but semantically ambiguous in ways that BNF would surface if the semantics were given as a grammar.

Separation of Concerns

BNF cleanly separates the *shape* of a program (its grammar) from its *meaning* (its semantics). This separation is enforced by the formalism: a BNF grammar says nothing about what programs mean, only about what strings are valid programs. This is not a limitation but a virtue: it allows the same grammar to be given multiple semantics (operational, denotational, axiomatic) without changing the specification of valid programs.

Formal Verification Target

A BNF grammar is a mathematical object that can serve as the specification for formal verification. A Spherepop program can be verified to be within the language of a grammar by a parser; verification at the semantic level then proceeds over the parse tree. This two-stage verification is not possible when the language is specified in prose or implicitly in a reference implementation.

The Spherepop BNF as Computational Universal

The Spherepop BNF grammar from Part III is not merely a syntactic convenience. It is a specification from which an implementation in any language, on any substrate, can be mechanically derived.

Definition 34.4 (Spherepop Core Grammar)**.** The Spherepop core language is

given by the following BNF:

$$
\begin{aligned}
\langle term \rangle &::= \langle var \rangle \mid \langle lambda \rangle \mid \langle app \rangle \mid \langle let \rangle \mid \langle pop \rangle \mid \langle refuse \rangle \mid \langle bind \rangle \mid \langle collapse \rangle \mid \langle seq \rangle \\
\langle var \rangle &::= \langle ident \rangle \\
\langle lambda \rangle &::= (\lambda \mid \mathsf{fn})\ \langle ident \rangle\ [:\ \langle type \rangle]\ (\rightarrow \mid .)\ \langle term \rangle \\
\langle app \rangle &::= \langle term \rangle\ \langle term \rangle \\
\langle let \rangle &::= \mathsf{let}\ \langle ident \rangle\ =\ \langle term \rangle\ \mathsf{in}\ \langle term \rangle \\
\langle pop \rangle &::= \mathsf{pop}\ \langle term \rangle \\
\langle refuse \rangle &::= \mathsf{refuse}\ \langle term \rangle\ [\langle reason \rangle] \\
\langle bind \rangle &::= \mathsf{bind}\ \langle term \rangle\ \langle term \rangle \\
\langle collapse \rangle &::= \mathsf{collapse}\ \langle term \rangle\ [\langle rule \rangle] \\
\langle seq \rangle &::= \mathsf{seq}\ [\langle term \rangle\ \{; \langle term \rangle\}^*] \\
\langle reason \rangle &::= \mathsf{violation}(\langle ident \rangle) \mid \mathsf{explicit}(\langle ident \rangle) \\
\langle rule \rangle &::= \mathsf{id} \mid \mathsf{last\_write} \mid \mathsf{accumulate} \mid \mathsf{proj}(\langle ident \rangle) \\
\langle type \rangle &::= \mathsf{Unit} \mid \mathsf{Never} \mid \langle type\_var \rangle \mid \mathsf{Admissible}(\langle type \rangle) \mid \langle type \rangle \rightarrow \langle type \rangle
\end{aligned}
$$

Theorem 34.5 (Grammar Determines Implementation)**.** *Given the Spherepop core BNF, a parser, type checker, and evaluator are mechanically derivable for any target platform.*

Proof. The grammar is context-free and therefore parseable by any LALR(1) or PEG parser generator. The type checker follows from the typing rules (which are themselves a kind of grammar over typing contexts). The evaluator follows from the operational semantics (which are also a grammar of world-state transitions). Each of these derivations is mechanical and platform-independent: the same BNF yields implementations in Rust, Python, Haskell, JavaScript, or any other host language without modification to the grammar. □

Grammar as Interface Contract

When the Spherepop grammar is fixed, it constitutes an interface contract between: (1) program authors, who write terms in the grammar's language; (2) parsers, which accept exactly the grammar's language; (3) type checkers, which verify admissibility of parsed terms; (4) evaluators, which execute well-typed terms; (5) compilers, which translate well-typed terms to Event IR.

Any implementation satisfying this contract is a correct Spherepop imple-

mentation. The grammar is the single source of truth for what the language is, independent of any particular reference implementation.

BNF as Anti-Collapse Artifact

The relationship between BNF grammars and the note theory of this monograph is not accidental.

A BNF grammar is a Refusal Note of the highest generativity. It specifies which strings are admissible (members of the language) and which are inadmissible. It refuses the collapse of the language into any particular implementation. And it generates the full continuation space of all programs in the language: any string in the language is a reachable continuation, and the grammar specifies exactly the set of reachable continuations.

Theorem 34.6 (BNF Grammars as Generative Refusal Notes)**.** *A BNF grammar G is simultaneously:*

1. *A* Generative Note*: it creates the continuation space $L(G)$ of all programs in the language, which did not exist before the grammar was specified.*
2. *A* Refusal Note*: it refuses all strings not in $L(G)$ as inadmissible, with the implicit reason* ParseError*.*
3. *A* Coordination Note*: it couples the behaviors of all implementations by binding them to the same accepted and rejected strings.*
4. *A* Capture Note*: it preserves the structure of valid programs across implementations, allowing programs written for one implementation to be run on another.*

Proof. (1) The language $L(G)$ is the set of all strings derivable from S in G. Before G exists, this set is undefined; after G exists, it is precisely specified. New programs in $L(G)$ are continuations that G makes reachable.

(2) A string $w \notin L(G)$ has no parse tree under G. The parser refuses it with a parse error. This is a Refuse operation with reason ParseError applied to the inadmissible string.

(3) Any two implementations I_1 and I_2 of G accept the same strings and reject the same strings. They are bound by the grammar to the same admissibility decisions. This is a Bind coupling their trajectory spaces.

(4) A program $p \in L(G)$ has an implementation-independent parse tree. The parse tree captures the structure of p independently of what evaluator processes it. This is a Capture Note for the program's syntactic structure. □

Chapter 35

Grammar, Circuits, and the Efficiency Hierarchy

Measuring Expressiveness Per Symbol

We want a precise way to compare notations for expressing computational structure. Let the *expressiveness per symbol* of a notation $\mathcal{N}$ for specification S be the ratio of the information content of S to the number of symbols required to express S in $\mathcal{N}$.

Definition 35.1 (Specification Complexity)**.** The *specification complexity* $K_{\mathcal{N}}(S)$ of specification S in notation $\mathcal{N}$ is the minimum number of symbols in $\mathcal{N}$ required to express S unambiguously. The *relative expressiveness* of $\mathcal{N}_1$ over $\mathcal{N}_2$ for S is

$$E(S, \mathcal{N}_1, \mathcal{N}_2) = \frac{K_{\mathcal{N}_2}(S)}{K_{\mathcal{N}_1}(S)}.$$

$E > 1$ means $\mathcal{N}_1$ is more expressive per symbol.

Theorem 35.2 (BNF Specification Efficiency)**.** *For context-free languages, BNF is asymptotically more symbol-efficient than circuit diagrams and at least as efficient as any other notation that is implementation-independent and mechanically parseable.*

Proof sketch. A circuit diagram for a function $f : \{0,1\}^n \to \{0,1\}^m$ specifies f by encoding its topology as a directed acyclic graph of gates. The minimum size of such a diagram is $\Omega(C(f))$ where $C(f)$ is the circuit complexity of f. A BNF grammar for the same function, specified as a language recognizer, can be written in $O(\log n + \log m + K(f))$ productions where $K(f)$ is the Kolmogorov complexity of f's description. For functions with short descriptions but high circuit complexity (such as sorting networks), BNF can be exponentially more compact. □

The Readability Spectrum

Different notations occupy different positions on the spectrum from machine-efficient to human-efficient.

Notation	Human Readable	Machine Parseable	Impl.-Independent
Machine code	No	Yes	No
Assembly	Partial	Yes	No
Circuit diagram	Yes (visual)	Partial	No
C/Rust program	Yes	Yes	Partial
Lambda calculus	Yes	Yes	Yes
BNF grammar	Yes	Yes	Yes
Natural language	Yes	No	Yes

BNF occupies the unique position of being simultaneously human-readable, mechanically parseable, and fully implementation-independent. No other common notation achieves all three.

Grammar as the Missing Level of Abstraction

Programming languages, circuit diagrams, and prose all operate at specific levels of abstraction. Grammar operates at the level of *specification*: it describes the structure of all possible instances of a computational artifact without committing to any particular instance.

Definition 35.3 (Abstraction Level Hierarchy)**.** The abstraction levels of computational description form a hierarchy:

1. *Physical*: transistor layouts, voltages, timing.
2. *Circuit*: gate-level topology.
3. *Machine*: instruction set architecture.
4. *Language*: program syntax and semantics.
5. *Grammar*: specification of the language itself.
6. *Meta-grammar*: specification of the notation for grammars.

Proposition 35.4 (Grammar as Implementation Firewall)**.** *A grammar at level n of the hierarchy is implementation-independent with respect to all levels below n. A Spherepop BNF grammar is therefore independent of all language implementations, all machine instruction sets, all circuit designs, and all physical substrates.*

This independence is exactly what makes BNF appropriate as the specifi-

cation language for Spherepop. The Spherepop grammar specifies what computations the language can express without committing to how they are executed. The execution details—Rust runtime, WASM compilation, hardware acceleration—are below the abstraction level at which the grammar operates.

Literate Grammar and Documentation

Knuth's literate programming [22] proposed that programs should be written as literature: prose explanations interleaved with code. The Spherepop approach inverts this slightly: grammar should be written as literature, with prose explanations interleaved with BNF productions.

Definition 35.5 (Literate Grammar)**.** A *literate grammar* is a BNF grammar augmented with:

1. Prose explanations of the intent of each production.
2. Examples of derivations in the grammar.
3. Semantic notes indicating what well-formed strings mean.
4. Transformation rules mapping productions to type-checking and evaluation rules.

The Spherepop BNF in this monograph is a literate grammar in this sense: each production is accompanied by the operational semantics rule it corresponds to and the type-checking rule it enables. This makes the grammar simultaneously a specification, a tutorial, and a formal verification target.

Part XIV

Supplementary Derivations

Chapter 36

Derivation Compendium

This chapter collects all supplementary derivations, filling the mathematical gaps indicated in the preceding parts. Each derivation is labeled by the chapter it supports.

Chapter 1: Notes Increase Reachability

For each continuation $c \in \mathcal{C}$, define the *reachability gain* of note N by

$$G_A(N, c, t) = P_A(c, t \mid N) - P_A(c, t).$$

Then N is a note precisely when $\exists c \in \mathcal{C},\ G_A(N, c, t) > 0$.

Proposition 36.1 (Nontrivial Notes Are Reachability-Increasing)**.** *If N is a note and all non-note effects are nonnegative, then the total reachability gain $G_A(N, t) = \sum_{c \in \mathcal{C}} G_A(N, c, t)$ is strictly positive.*

Proof. By definition, there exists at least one c_0 such that $G_A(N, c_0, t) > 0$. If all other terms satisfy $G_A(N, c, t) \geq 0$, then the total sum contains one strictly positive term and no negative terms. Hence $G_A(N, t) > 0$. □

Chapter 2: State as Quotient

Proposition 36.2 (State Is a Quotient of History)**.** *Every surjective state projection $\sigma : \mathcal{H} \to S$ factors uniquely through the quotient map $q : \mathcal{H} \to \mathcal{H}/\!\sim_\sigma$.*

Proof. Define $\bar{\sigma}([H]) = \sigma(H)$. This is well-defined because $H_1 \sim_\sigma H_2$ implies $\sigma(H_1) = \sigma(H_2)$. Surjectivity makes $\bar{\sigma}$ onto; injectivity follows because distinct classes have distinct σ-values. Thus $S \cong \mathcal{H}/\!\sim_\sigma$. □

Chapter 3: Operator Minimality

Theorem 36.3 (Operator Minimality)**.** *No one of* Pop, Refuse, Bind, Collapse *is definable from the other three while preserving its effect on history, option space, admissibility, and observation.*

Proof. Pop uniquely decreases Ω; the other three leave Ω unchanged. Refuse uniquely changes the admissibility set $\mathcal{A}(H)$ without decreasing Ω; Pop changes Ω, while Bind and Collapse do not record inadmissibility. Bind uniquely introduces a dependency relation between two elements without consuming or refusing either; no combination of the other operators produces a pure dependency edge. Collapse uniquely maps a history through an observational rule into a quotient space; the other operators extend history internally but produce no observation. Hence all four are irreducible. □

Chapter 4: Free Category

Theorem 36.4 (Histories Form the Free Category on the Event Graph)**.** *Let G be the directed graph whose edges are primitive events. Then* **Hist** *is the free category generated by G.*

Proof. Objects are option spaces; morphisms are finite paths. Given any category $\mathcal{D}$ and graph morphism $F : G \to U(\mathcal{D})$, the unique functor $\bar{F} : \mathbf{Hist} \to \mathcal{D}$ extending F sends $[e_1, \ldots, e_n]$ to $F(e_n) \circ \cdots \circ F(e_1)$. This is the universal property of the free category. □

Chapter 5: Small-Step Determinacy

Proposition 36.5 (Determinacy Under Fixed Rule Selection)**.** *If the next operator and its arguments are fixed, Spherepop small-step reduction is deterministic.*

Proof. Each transition rule has a unique conclusion for fixed inputs. No fixed operator application can reduce to two distinct worlds. □

Chapter 6: Collapse Entropy Monotonicity

Theorem 36.6 (Collapse Increases Observational Entropy Under Coarsening)**.** *If c_2 is coarser than c_1, then every c_2-observation has entropy at least as large as the entropy of any c_1-observation contained inside it.*

Proof. Coarsening means each c_1-class is contained in some c_2-class. The fibre of the coarser observation contains the fibre of the finer observation, so $|c_2^{-1}(o_2)| \geq |c_1^{-1}(o_1)|$ for contained observations. Taking logarithms gives the entropy inequality. □

Chapter 7: Type Soundness

Theorem 36.7 (Spherepop Type Soundness)**.** *If $\vdash t : T$ and $t \to^* t'$, then either t' is a value of type T, or there exists t'' such that $t' \to t''$.*

Proof. By repeated application of Preservation and Progress. Preservation gives $\vdash t' : T$ after any finite reduction sequence. Progress implies t' is either a value or can take another step. □

Chapter 8: Lattice Collapse Threshold

Proposition 36.8 (Monotonicity of Collapse Permission)**.** *If $\ell_1 \leq \ell_2$ and $\Gamma \vdash t : \mathsf{Admissible}_{\ell_2}(T)$, then every collapse permitted at level ℓ_1 is also permitted at level ℓ_2.*

Proof. A higher admissibility level satisfies at least the constraints of a lower level. Since $\ell_1 \leq \ell_2$, any rule whose threshold is met by ℓ_1 is also met by ℓ_2. □

Note Composition Theory

Define note composition by

$$\mathcal{C}(N_1 \otimes N_2) = \mathcal{C}(N_1) \cup \mathcal{C}(N_2) \cup \mathcal{C}_{\text{bind}}(N_1, N_2),$$

where $\mathcal{C}_{\text{bind}}(N_1, N_2)$ is the set of continuations reachable only because both notes are jointly available.

Theorem 36.9 (Coordination Surplus)**.** *If $\mathcal{C}_{\text{bind}}(N_1, N_2) \neq \varnothing$, then*

$$V_C(N_1 \otimes N_2) > V_C(N_1) + V_C(N_2) - |\mathcal{C}(N_1) \cap \mathcal{C}(N_2)|.$$

Proof. The continuation set of the composite contains the union of the individual sets plus at least one additional bound continuation. The ordinary union has size $V_C(N_1) + V_C(N_2) - |\mathcal{C}(N_1) \cap \mathcal{C}(N_2)|$. Adding the nonempty bound set strictly increases this. □

Theorem 36.10 (Note Entropy)**.** *Define the* note entropy *of N relative to agent A at time t as*

$$H_A(N,t) = -\sum_{c\in\mathcal{C}} P_A(c,t \mid N)\log P_A(c,t \mid N).$$

A note N' is more focused *than N if $H_A(N',t) < H_A(N,t)$: it concentrates reachability on a smaller set of continuations. A note N' is more* generative *than N if $V_C(N') > V_C(N)$.*

Remark 36.11. Focus and generativity trade off: collapse notes are highly focused (low entropy) while generative notes are high-entropy. The note taxonomy orders by this tradeoff.

Note Class Derivations

Proposition 36.12 (Capture Adequacy)**.** *A capture note N is adequate for task family T iff for every continuation $c \in T$ there exists recoverable data $d \subseteq N$ such that $P_A(c,t \mid d) > P_A(c,t)$.*

Theorem 36.13 (Prospective Notes Are Deferred Pops)**.** *Every prospective note determines a deferred commitment operator. At time t the continuation is not yet executed (no immediate Pop occurs); at time t' the agent follows the note, selecting the future continuation via a Pop.*

Proof. A prospective note binds present state A_t to future continuation $c_{t'}$. The selection at t' is a Pop of $c_{t'}$ from the available alternatives. The note stores a commitment whose execution is deferred. □

Theorem 36.14 (Repair Notes Minimize Reconstruction Cost)**.** *A repair note N is optimal in family $\mathcal{F}$ when $C_r(c \mid N) = \min_{M\in\mathcal{F}} C_r(c \mid M)$.*

Proof. Repair value is $C_r(c) - C_r(c \mid N)$. Since $C_r(c)$ is fixed for the continuation, maximizing repair value is equivalent to minimizing $C_r(c \mid N)$. □

Proposition 36.15 (Checksums as Pure Refusal Notes)**.** *A checksum $\chi(h)$ refuses informational collapse by distinguishing corrupted histories from admissible ones. If $\chi(h') \neq \chi(h)$, then h' is refused as a valid continuation of h. The checksum is a pure refusal note: it stores only the boundary between admissible and inadmissible continuations.*

Theorem 36.16 (Generativity Is Strict Reachability Expansion)**.** *A note N is generative iff $\exists c : P_A(c,t) = 0$ and $P_A(c,t \mid N) > 0$.*

Proof. If such c exists, N creates access to a previously unreachable trajectory. Conversely, generativity requires opening at least one previously unavailable trajectory, which is exactly this condition. □

Theorem 36.17 (Collapse Notes Trade Detail for Access)**.** *Let $\mathcal{N}$ be a finite note collection. A collapse note K partitioning $\mathcal{N}$ into m classes reduces search cost from $O(|\mathcal{N}|)$ to $O(m + \max_i |\mathcal{N}_i|)$.*

Proof. Without organizing structure, worst-case search examines every note. With a partition of m classes, the agent selects one class (m steps) then searches that class ($\max_i |\mathcal{N}_i|$ steps worst case). □

Compilation and GC Derivations

Theorem 36.18 (Event IR Is History-Preserving)**.** *For every primitive source event e, compilation followed by VM execution appends the same event to history as interpretation.*

Proof. By case analysis: Pop compiles to `IrEvent::Pop`, which the VM appends as Pop; Refuse(r) compiles to `IrEvent::Refuse{r}`, appended as Refuse(r); analogously for Bind and Collapse(c). □

Theorem 36.19 (GC Is Safe Exactly When It Preserves Live Fibres)**.** *A GC operation $g : \mathcal{H} \to \mathcal{H}$ is safe iff for every live continuation c, $P_A(c, t \mid g(H)) = P_A(c, t \mid H)$.*

Proof. If the equality holds for all live continuations, GC has removed no history needed for any reachable future—hence safe. Conversely, safety by definition preserves all history required for live continuations. □

Error Correction Derivation

Theorem 36.20 (Certified Refusal Prevents Silent Error Absorption)**.** *If every refusal carries a proof term π, then no error can be erased without either being handled or remaining visible in the output type.*

Proof. A refusal has type Refused(T, π). The typing rules allow this type to be handled by a recovery term or propagated. There is no rule converting Refused(T, π) into T while discarding π. Hence the error cannot be silently absorbed. □

Category Theory Derivations

Theorem 36.21 (Collapse as Reflection)**.** *If* $\mathbf{Obs}_c$ *is the full subcategory of* **Hist** *consisting of histories saturated under* $\sim_c$*, then the quotient functor* $F_c : \mathbf{Hist} \to \mathbf{Obs}_c$ *is left adjoint to the inclusion* $I_c : \mathbf{Obs}_c \hookrightarrow \mathbf{Hist}$*.*

Proof. For each history H and saturated observable O, every morphism $H \to I_c(O)$ constant on $\sim_c$-classes factors uniquely through $F_c(H)$. This gives the natural bijection $\mathbf{Obs}_c(F_c(H), O) \cong \mathbf{Hist}(H, I_c(O))$, establishing $F_c \dashv I_c$. □

Corollary 36.22 (Perfect Observation Requires History)**.** *An observation map is perfectly history-faithful iff it is isomorphic to the identity observation on* $\mathcal{H}$*.*

Proof. Perfect history-faithfulness requires injectivity. A bijective observation establishes an isomorphism between histories and observations, hence is isomorphic to the identity. □

Sheaf Derivations

Theorem 36.23 (State Illusion as Failure of Unique Gluing)**.** *Given a cover of observations* $\{U_i\}$*, state illusion occurs exactly when compatible local sections* $s_i \in \mathcal{H}(U_i)$ *admit more than one global section.*

Proof. Compatible local sections agree on overlaps. If more than one global history restricts to the same local observations, the observations do not determine a unique history—precisely the state illusion. □

CLIO Derivation

Theorem 36.24 (Spherepop Observational Entropy Is Discrete CLIO Entropy)**.** *The CLIO fibre entropy* $S_\pi(o) = \log \mathrm{Vol}(\pi^{-1}(o))$ *reduces in the discrete setting to* $S_c(o) = \log |c^{-1}(o)|$*.*

Proof. In the discrete setting, volume is counting measure. Therefore $\mathrm{Vol}(\pi^{-1}(o)) = |\pi^{-1}(o)| = |c^{-1}(o)|$. □

Active Geodesic Derivation

Proposition 36.25 (Histories Are Discrete Geodesics)**.** *Let* $L(e_i)$ *be the local cost of event* e_i *and define the action* $\mathcal{S}(H) = \sum_i L(e_i)$*. A Spherepop execution is a discrete*

geodesic when it minimizes $\mathcal{S}(H)$ among admissible histories with the same boundary conditions.

Proof. In discrete variational mechanics, a geodesic minimizes action between fixed endpoints. A Spherepop history is a path in event space; restricting to admissible histories and minimizing the additive cost functional gives the discrete geodesic condition. □

MEM|8 Locality Derivation

Theorem 36.26 (Locality as Continuation Preservation)**.** *If related memory components are physically or logically local, then the expected cost of reconstructing their shared continuation is lower than under random placement.*

Proof. Let $d(x, y)$ be access distance between memory components. Reconstruction cost is monotone in aggregate distance among components required by a continuation. Local placement lowers expected pairwise distance relative to random placement, hence lowers expected reconstruction cost. □

Civilizational Collapse Derivation

Theorem 36.27 (Reachability Loss Under Archive Destruction)**.** *Let $c \in R(Z)$ be reachable only through some $N \in \mathcal{N}$. Destroying $\mathcal{N}$ strictly decreases collective reachability.*

Proof. Since c is reachable only through N, removing N removes c from $R(Z)$. The post-destruction reachability is a proper subset of the original. □

Inverse Problem Derivation

Proposition 36.28 (Kernel Triviality Characterizes Complete Observation)**.** *A collapse rule c is observationally complete iff* $\ker(c) = \Delta_{\mathcal{H}} = \{(H, H) : H \in \mathcal{H}\}$.

Proof. Observational completeness means $c(H_1) = c(H_2) \Rightarrow H_1 = H_2$, so the kernel contains only diagonal pairs. Conversely, a diagonal kernel gives injectivity. □

Final Unification Theorem

Theorem 36.29 (Notes as Reachability Artifacts)**.** *An artifact N is a note iff it is a reachability artifact: there exists an agent A, a continuation c, and a future time t such that $P_A(c, t \mid N) > P_A(c, t)$. Spherepop programs are notes because they preserve, restrict, bind, refuse, and collapse continuations in history space.*

Proof. The first claim is the definition of a note under the continuation account. For the second: every Spherepop program constructs a history from primitive events. Pop selects continuations; Refuse preserves inadmissibility distinctions; Bind couples trajectories; Collapse makes histories observable under rules. A Spherepop program is therefore an artifact that changes which continuations remain reachable to future agents. Hence it is a note. □

Corollary 36.30 (Universal Reachability Artifact Thesis)**.** *Programs, proofs, archives, maps, protocols, libraries, institutions, languages, and civilizations are all special cases of the same abstract object: a reachability artifact. They differ in the medium through which they preserve continuations and in the scale at which they operate. The abstract structure is identical in every case.*

Proof. By Theorem 36.29, any artifact that increases the probability of traversing at least one continuation is a note. Each of the listed artifacts does this:

A *program* preserves a computational continuation. A *proof* preserves an inferential continuation. An *archive* preserves historical continuations against decay. A *map* creates navigational continuations. A *protocol* binds communicative continuations between agents. A *library* is a repository of continuations across domains. An *institution* binds social and coordination continuations across generations. A *language* is a system of semantic continuations shared by a community. A *civilization* is the totality of all of the above, organized into a distributed hierarchy of note infrastructure.

In every case, the note increases $P_A(c, t)$ for some continuation c. □

Part XV

Process Algebra and Concurrency

Chapter 37

Spherepop as a Process Algebra

Motivation

The four Spherepop operators—Pop, Refuse, Bind, Collapse—were derived in Part II as the minimal complete set for managing possibility space in sequential computation. But computation is rarely purely sequential. Processes run in parallel, communicate, and synchronize. This chapter develops the concurrent extension of Spherepop and shows that the history-primary ontology handles concurrency more naturally than state-centric models.

The key insight is that concurrency is a source of state illusion. When two processes P and Q run in parallel and access shared state, the observable outcome depends on the interleaving order—the specific history of which events occurred first. A state-centric model can only see the final state, losing the interleaving information that explains race conditions, deadlocks, and non-deterministic behavior. A history-primary model retains the interleaving and makes it explicit.

Parallel Composition

Definition 37.1 (Parallel World)**.** A *parallel world* is a pair $W_{\parallel} = (H_1 \parallel H_2, \Omega_1 \cup \Omega_2)$ where H_1 and H_2 are histories of concurrent processes with disjoint option spaces $\Omega_1 \cap \Omega_2 = \varnothing$.

Definition 37.2 (Interleaving Semantics)**.** The *interleaving semantics* of $W_{\parallel}$ is the set of all histories H that are consistent interleavings of H_1 and H_2:

$$\mathsf{IL}(H_1, H_2) = \{H \in \mathcal{H} \mid H \text{ is a shuffle of } H_1 \text{ and } H_2\},$$

where a *shuffle* of H_1 and H_2 is any sequence containing all events of both, with the relative order within each preserved.

Proposition 37.3 (Interleaving Count)**.** *The number of distinct interleavings of histories of lengths m and n is $\binom{m+n}{m}$.*

Proof. An interleaving is determined by choosing which m of the $m + n$ positions are occupied by events from H_1; the remaining n positions are occupied by events from H_2 in order. There are $\binom{m+n}{m}$ such choices. □

Remark 37.4. The combinatorial explosion in $\binom{m+n}{m}$ is the formal content of the observation that concurrent programs are hard to test. A state-centric test observes only one interleaving; the Spherepop history preserves which one was executed, making the interleaving explicit and auditable.

Synchronization via Bind

Definition 37.5 (Synchronization Event)**.** A *synchronization event* between processes P and Q at symbols $a \in \Omega_1$ and $b \in \Omega_2$ is the event $\mathsf{Bind}(a, b)$ appended to the history when both processes commit to their respective symbols simultaneously.

Theorem 37.6 (Bind as Synchronization)**.** *In a parallel world $W_\parallel$, the event $\mathsf{Bind}(a, b)$ with $a \in \Omega_1$ and $b \in \Omega_2$ is semantically equivalent to a channel communication in the π-calculus: process P sends a on channel ab while process Q receives b.*

Proof. In the π-calculus [25], the communication $\bar{a}b.P \mid a(x).Q$ reduces to $P \mid Q[b/x]$, recording that b was transmitted on channel a. In Spherepop, $\mathsf{Bind}(a, b)$ records the dependency between a and b in the history without consuming either from their option spaces. After the bind, any future collapse rule can verify that a and b are coupled. The structural correspondence holds: both formalisms record the communication event without immediately collapsing its participants. □

Race Conditions as Non-Deterministic Collapse

Definition 37.7 (Race Condition)**.** A *race condition* occurs in a parallel world $W_\parallel$ when two or more processes attempt to commit (Pop) to the same symbol $x \in \Omega_1 \cap \Omega_2$ simultaneously, and the observable outcome depends on which commits first.

Theorem 37.8 (Race Conditions Are Collapse Artifacts)**.** *Race conditions are not intrinsic to the computation but arise from applying a collapse rule that identifies interleavings differing only in which process committed first.*

Proof. Let H_1 be the history where process P commits to x first, and H_2 the history where Q commits first. The histories $H_1 \neq H_2$ are distinct. However, under a collapse rule c that discards ordering information (such as c_{acc}), $c(H_1) = c(H_2)$. The observable state is identical despite the different histories. A programmer who observes only $c(H_1)$ cannot determine whether P or Q committed first—the race is invisible. But in the Spherepop world, $H_1 \neq H_2$ is a fact preserved in the history, accessible by switching to the Identity rule c_I. □

Deadlock as Inadmissibility Cycle

Definition 37.9 (Dependency Graph)**.** The *dependency graph* G_H of a history H has one node for each symbol in Ω_0 and a directed edge $a \to b$ for each $\mathsf{Bind}(a, b) \in H$.

Theorem 37.10 (Deadlock as Graph Cycle)**.** *A parallel computation is deadlocked at state $(H, \Omega_1 \cup \Omega_2)$ if and only if the dependency graph G_H contains a directed cycle $a_1 \to a_2 \to \cdots \to a_k \to a_1$ where each a_i is waiting on the next in its current admissibility set.*

Proof. Deadlock occurs when no process can proceed: every process is waiting for a resource held by another. In Spherepop, a process holds resource x by having executed $\mathsf{Pop}(x)$ (committing to x) and waits for resource y when $y \notin \mathcal{A}(H)$ due to a dependency recorded by $\mathsf{Bind}(x, y)$. A cycle $a_1 \to a_2 \to \cdots \to a_k \to a_1$ in G_H means a_1 waits for a_2, a_2 waits for a_3, and so on cyclically, which is precisely the formal definition of deadlock. □

Corollary 37.11 (Deadlock Detection)**.** *Deadlock detection in Spherepop reduces to cycle detection in the dependency graph G_H, which is computable in $O(|V| + |E|)$ time by depth-first search.*

Chapter 38

The Process Calculus of Spherepop

Terms and Their Meanings

The Spherepop process calculus extends the core term language with parallel composition, restriction, and replication.

Definition 38.1 (Extended Term Language)**.**

$$\langle \mathit{proc} \rangle ::= \langle \mathit{term} \rangle \mid \langle \mathit{proc} \rangle \parallel \langle \mathit{proc} \rangle \mid \nu x.\langle \mathit{proc} \rangle \mid !\langle \mathit{proc} \rangle \mid \mathbf{0}$$

where $P \parallel Q$ is parallel composition, $\nu x.P$ restricts the scope of symbol x to P, $!P$ is the replication of P (an unbounded number of copies), and $\mathbf{0}$ is the terminated process.

Definition 38.2 (Structural Congruence)**.** Processes are *structurally congruent*, written $P \equiv Q$, if they differ only in syntactic form:

$$P \parallel Q \equiv Q \parallel P$$
$$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$$
$$P \parallel \mathbf{0} \equiv P$$
$$\nu x.\mathbf{0} \equiv \mathbf{0}$$
$$!P \equiv P \parallel !P$$

Theorem 38.3 (Structural Congruence Preserves History Equivalence)**.** *If $P \equiv Q$, then for any initial world W, the sets of histories produced by P and Q are equal:*

$$\mathsf{Hist}(P, W) = \mathsf{Hist}(Q, W).$$

Proof. By induction on the structural congruence rules. Commutativity of $\parallel$ permutes the order of interleaved events but does not change which events occur. Associativity regroups parallel composition without affecting the event set. The

identity law $P \parallel \mathbf{0} \equiv P$ holds because $\mathbf{0}$ contributes no events. The replication law $!P \equiv P \parallel !P$ reflects the coinductive nature of replication: history sets of both sides are identical by the fixed-point equation. □

Reduction Rules for Processes

Definition 38.4 (Process Reduction)**.** Process reduction $P \to P'$ is defined by:

$$\mathsf{pop}(a) \parallel \mathsf{bind}(a, b).Q \to Q[\text{linked}(a, b)] \quad \text{[SYNC]}$$

$$\frac{P \to P'}{P \parallel Q \to P' \parallel Q} \quad \text{[PAR]}$$

$$\frac{P \to P'}{P \parallel Q \to P \parallel Q'} \quad \text{[SYMM]}$$

$$\frac{P \equiv P' \to Q' \equiv Q}{P \to Q} \quad \text{[STRUCT]}$$

Theorem 38.5 (Subject Reduction for Processes)**.** *If $\Gamma \vdash P : T$ and $P \to P'$, then $\Gamma \vdash P' : T$.*

Proof. By case analysis on the reduction rules. The key case is [SYNC]: $\mathsf{pop}(a)$ has type $\mathsf{Admissible}(\mathsf{Unit})$ by [T-POP]; $\mathsf{bind}(a, b)$ has type $\mathsf{Process}(T_a \rightsquigarrow T_b)$ by [T-BIND]. After synchronization, the linked state $\text{linked}(a, b)$ is typed by the composition of these certificates. □

Bisimulation and Behavioral Equivalence

Definition 38.6 (History Bisimulation)**.** A binary relation $\mathcal{R}$ on processes is a *history bisimulation* if whenever $(P, Q) \in \mathcal{R}$:

1. For every step $P \xrightarrow{H} P'$, there exists Q' such that $Q \xrightarrow{H} Q'$ and $(P', Q') \in \mathcal{R}$.
2. Symmetrically with P and Q swapped.

where $P \xrightarrow{H} P'$ means P produces history H and reduces to P'.

Theorem 38.7 (Bisimulation Is Congruence)**.** *History bisimulation $\sim$ is a congruence: if $P \sim Q$ then $P \parallel R \sim Q \parallel R$ for all R.*

Proof. Let $\mathcal{R} = \{(P \parallel R, Q \parallel R) \mid P \sim Q\}$. We show $\mathcal{R}$ is a bisimulation. Any step of $P \parallel R$ is either a step of P alone, a step of R alone, or a synchronization between P and R. For steps of P alone: by $P \sim Q$, there is a matching step of Q, giving $Q \parallel R$ a matching step. For steps of R: R matches itself. For synchronizations: the Bind event is recorded identically in both histories, so $Q \parallel R$ can

perform the same synchronization. □

Chapter 39

Distributed Spherepop

Multiple Worlds and Consistency

A distributed Spherepop system consists of multiple worlds $(H_1, \Omega_1), \ldots, (H_n, \Omega_n)$ running at different locations, each maintaining its own history. Consistency requires that the histories agree on shared observations.

Definition 39.1 (Distributed World System)**.** A *distributed world system* is a tuple $\mathcal{W} = (W_1, \ldots, W_n, \sim_{\mathsf{glob}})$ where each $W_i = (H_i, \Omega_i)$ is a local world and $\sim_{\mathsf{glob}}$ is a global consistency relation specifying which pairs of events across worlds must agree.

Definition 39.2 (Global Consistency)**.** A distributed world system $\mathcal{W}$ is *globally consistent* under collapse rule c if

$$c(H_i)|_{\Omega_i \cap \Omega_j} = c(H_j)|_{\Omega_i \cap \Omega_j} \quad \text{for all } i \neq j,$$

where $c(H)|_\Omega$ denotes the restriction of the observation to symbols in Ω.

Theorem 39.3 (CAP as Collapse Tradeoff)**.** *In a distributed world system subject to network partitions, no system can simultaneously guarantee:*

1. Consistency*: all worlds observe the same collapse $c(H)$.*
2. Availability*: every non-failed world responds to requests.*
3. Partition tolerance*: the system continues operating during network failures.*

At most two of three can hold at any given time.

Proof sketch. During a network partition, worlds W_i and W_j cannot communicate. If both must remain available, they each process independent events, producing histories H_i and H_j that diverge. At partition resolution, $c(H_i)$ and $c(H_j)$ may disagree on shared symbols, violating consistency. To maintain consistency, one world must refuse to process events (sacrificing availability). This is the formal Spherepop statement of the CAP theorem [32]. □

CRDTs as Admissibility-Preserving Merges

Conflict-free replicated data types (CRDTs) achieve eventual consistency by ensuring that concurrent updates commute. In Spherepop terms, CRDTs are merge operations that preserve admissibility.

Definition 39.4 (History Merge)**.** A *history merge* of H_1 and H_2 is any history $H_1 \sqcup H_2$ such that both H_1 and H_2 are subsequences of $H_1 \sqcup H_2$, preserving the relative order of events within each.

Definition 39.5 (CRDT in Spherepop)**.** A *Spherepop CRDT* is a data type whose update operations are Pop events that commute under the collapse rule: for any two updates P_x and P_y,

$$c(H +\!\!+ [P_x, P_y]) = c(H +\!\!+ [P_y, P_x]).$$

Theorem 39.6 (CRDTs Achieve Eventual Consistency)**.** *A Spherepop CRDT system achieves eventual consistency: after all updates propagate, all worlds reach the same observable state.*

Proof. Since updates commute under c, any merge $H_1 \sqcup H_2$ produces the same observable state $c(H_1 \sqcup H_2)$ regardless of the interleaving chosen. After all updates have propagated to all worlds, each world's history is a permutation of the same multiset of events. By commutativity, all produce the same observable state. □

Example 39.7 (Grow-Only Counter as Accumulate CRDT)**.** A grow-only counter in Spherepop is a collection of Pop events on a symbol x, with the Accumulate collapse rule $c_{\text{acc}}(H)(x) = |\{i : e_i = \mathsf{Pop}(x)\}|$. Any two concurrent increments commute under c_{acc} because addition is commutative: regardless of the order $\mathsf{Pop}(x), \mathsf{Pop}(x)$ are interleaved, the count is the same. This is the formal Spherepop account of the G-Counter CRDT.

Part XVI

Full Type-Theoretic Metatheory

Chapter 40

The Complete Proof System

Motivation

The type system of Part IV established Progress, Preservation, and Collapse Soundness with proof sketches. A type theorist demands the full metatheory: canonical forms, a weakening lemma, an exchange lemma, a substitution lemma, and the complete proofs of Progress and Preservation deriving from them. This chapter provides that apparatus following the syntactic approach of Wright and Felleisen [36].

Canonical Forms

Lemma 40.1 (Canonical Forms)**.** *If* $\vdash v : T$ (*well-typed value in empty context*) *then:*

1. *If* $T = \mathsf{Unit}$ *then* $v = ()$.
2. *If* $T = \Pi(x : T_1).T_2$ *then* $v = \lambda x : T_1.b$ *for some* b.
3. *If* $T = \mathsf{Admissible}(T')$ *then* $v = \mathsf{pop}(v')$ *for some* v' *with* $\vdash v' : T'$.
4. *If* $T = \mathsf{Refused}(T', r)$ *then* $v = \mathsf{refuse}(v', r)$ *for some* v' *with* $\vdash v' : T'$.
5. *If* $T = \mathsf{Collapsed}(T', c)$ *then* $v = \mathsf{collapse}(v', c)$ *for some* v' *with* $\vdash v' : \mathsf{Admissible}(T')$.
6. *If* $T = \mathsf{Process}(T_1 \rightsquigarrow T_2)$ *then* $v = \mathsf{bind}(v_1, v_2)$ *for some* $v_1 : T_1$ *and* $v_2 : T_2$.

Proof. By induction on the typing derivation $\vdash v : T$ in the empty context. Since v is a value, it is not a redex. Each type is introduced by exactly one rule, and values are the results of those introduction rules. We proceed case by case.

Case $T = \mathsf{Unit}$: The only introduction form for Unit is the unit value $()$, so $v = ()$.

Case $T = \Pi(x : T_1).T_2$: The only introduction form for Π-types is [T-LAM], which produces $v = \lambda x : T_1.b$.

Case $T = \mathsf{Admissible}(T')$: The only rule producing $\mathsf{Admissible}(T')$ is [T-POP],

which requires $v = \mathsf{pop}(v')$.

Case $T = \mathsf{Refused}(T', r)$: Only [T-REFUSE] produces this type, requiring $v = \mathsf{refuse}(v', r)$.

Case $T = \mathsf{Collapsed}(T', c)$: Only [T-COLLAPSE] produces this type, requiring the inner term to have type $\mathsf{Admissible}(T')$.

Case $T = \mathsf{Process}(T_1 \rightsquigarrow T_2)$: Only [T-BIND] produces this type. □

Structural Lemmas

Lemma 40.2 (Weakening)**.** *If* $\Gamma \vdash t : T$ *and* $x \notin \mathrm{dom}(\Gamma)$*, then* $\Gamma, x : S \vdash t : T$ *for any type* S*.*

Proof. By induction on the derivation $\Gamma \vdash t : T$.

Base case [T-VAR]: If $t = y$ and $(y : T) \in \Gamma$, then $(y : T) \in \Gamma, x : S$ (since $x \neq y$). Apply [T-VAR].

Case [T-LAM]: $t = \lambda y : T_1.b$ with $\Gamma, y : T_1 \vdash b : T_2$. By induction hypothesis (choosing fresh x), $\Gamma, x : S, y : T_1 \vdash b : T_2$. By exchange (Lemma 40.3), $\Gamma, y : T_1, x : S \vdash b : T_2$. Apply [T-LAM].

Cases [T-POP], [T-REFUSE], [T-BIND], [T-COLLAPSE]: Apply the induction hypothesis to all premises, then re-apply the rule.

Case [T-APP]: Apply induction to both premises. □

Lemma 40.3 (Exchange)**.** *If* $\Gamma, x : S, y : T, \Delta \vdash t : U$*, then* $\Gamma, y : T, x : S, \Delta \vdash t : U$*.*

Proof. By induction on the derivation. All rules are insensitive to the order of entries in Γ that do not match the current variable being looked up. The only sensitive case is [T-VAR], which looks up exactly one variable; swapping two different bindings leaves the lookup result unchanged. □

Lemma 40.4 (Substitution)**.** *If* $\Gamma, x : S \vdash t : T$ *and* $\Gamma \vdash s : S$*, then* $\Gamma \vdash t[x := s] : T[x := s]$*.*

Proof. By induction on the derivation $\Gamma, x : S \vdash t : T$.

Case [T-VAR] with $t = x$, $T = S$: $t[x := s] = s$ and $T[x := s] = S[x := s] = S$ (since $x \notin \mathrm{FV}(S)$ in well-formed contexts). We need $\Gamma \vdash s : S$, which is given.

Case [T-VAR] with $t = y \neq x$: $t[x := s] = y$ and the binding $(y : T) \in \Gamma$ (not involving x). Apply [T-VAR].

Case [T-LAM]: $t = \lambda y : T_1.b$ with $\Gamma, x : S, y : T_1 \vdash b : T_2$. Choose y fresh so $y \neq x$. By induction, $\Gamma, y : T_1 \vdash b[x := s] : T_2[x := s]$. Then $t[x := s] = \lambda y : T_1[x := s].b[x := s]$ and apply [T-LAM].

Case [T-APP]: $t = f\,a$ with $\Gamma, x : S \vdash f : \Pi(y : T_1).T_2$ and $\Gamma, x : S \vdash a : T_1$. By induction: $\Gamma \vdash f[x := s] : (\Pi(y : T_1).T_2)[x := s]$ and $\Gamma \vdash a[x := s] : T_1[x := s]$. Apply [T-APP].

Case [T-POP]: $t = \mathsf{pop}(t')$. By induction $\Gamma \vdash t'[x := s] : T'[x := s]$. Apply [T-POP].

Cases [T-REFUSE], [T-BIND], [T-COLLAPSE]: Analogous to the Pop case. □

Full Progress and Preservation

Theorem 40.5 (Full Type Preservation)**.** *If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.*

Proof. By induction on the derivation of $\Gamma \vdash t : T$, with case analysis on the reduction step $t \to t'$.

Case [T-APP] *with* $[\beta]$*-reduction*: $t = (\lambda x : T_1.b)\,a \to b[x := a] = t'$. From [T-APP]: $\Gamma \vdash \lambda x : T_1.b : \Pi(x : T_1).T_2$ and $\Gamma \vdash a : T_1$. From [T-LAM]: $\Gamma, x : T_1 \vdash b : T_2$. By Substitution (Lemma 40.4): $\Gamma \vdash b[x := a] : T_2[x := a] = T$.

Case [T-POP] *with inner reduction*: $t = \mathsf{pop}(t_0)$ with $\Gamma \vdash t_0 : T_0$ and $t_0 \to t_0'$. By induction hypothesis, $\Gamma \vdash t_0' : T_0$. Apply [T-POP] to obtain $\Gamma \vdash \mathsf{pop}(t_0') : \mathsf{Admissible}(T_0)$.

Cases [T-REFUSE], [T-BIND], [T-COLLAPSE]: Analogous, reducing the inner term.

Case [T-COLLAPSE] *with inner Pop value*: $t = \mathsf{collapse}(\mathsf{pop}(v), c)$ where $\mathsf{pop}(v)$ is a value of type $\mathsf{Admissible}(T')$. This reduces to a sealed observation value of type $\mathsf{Collapsed}(T', c)$. By [T-COLLAPSE], the type is preserved. □

Theorem 40.6 (Full Progress)**.** *If $\vdash t : T$ (well-typed in empty context), then either t is a value, or there exists t' such that $t \to t'$.*

Proof. By induction on the derivation $\vdash t : T$.

Case [T-VAR]: Impossible in empty context.

Case [T-LAM]: $t = \lambda x : T_1.b$ is a value.

Case [T-APP]: $t = f\,a$. By induction, f is either a value or can step. If f can step to f', then $f\,a \to f'\,a$. If f is a value, by Canonical Forms (Lemma 40.1) with type $\Pi(x : T_1).T_2$, we have $f = \lambda x : T_1.b$. Now by induction on a: if a steps to a', then $(\lambda x : T_1.b)\,a \to (\lambda x : T_1.b)\,a'$. If a is a value, then $(\lambda x : T_1.b)\,a \to b[x := a]$ by $[\beta]$-reduction.

Case [T-POP]: $t = \mathsf{pop}(t_0)$. If t_0 is a value, then $\mathsf{pop}(t_0)$ is a value. If $t_0 \to t_0'$, then $\mathsf{pop}(t_0) \to \mathsf{pop}(t_0')$.

Case [T-REFUSE]: Analogous to Pop.

Case [T-COLLAPSE]: $t = \mathsf{collapse}(t_0, c)$ with $\vdash t_0 : \mathsf{Admissible}(T')$. By induction on t_0: if t_0 is a value, by Canonical Forms it is $\mathsf{pop}(v)$, and $\mathsf{collapse}(\mathsf{pop}(v), c)$ reduces to the sealed observation—a value. If t_0 steps, then t steps by reducing inside.

Case [T-BIND]: $t = \mathsf{bind}(t_1, t_2)$. By induction, each component either steps or is a value. If any component steps, t steps. If both are values, $\mathsf{bind}(v_1, v_2)$ is a value. □

Corollary 40.7 (Type Soundness)**.** *If $\vdash t : T$, then evaluation of t either produces a value of type T or diverges. It does not get stuck.*

Proof. By repeated application of Progress and Preservation. At each step, either t is a value (done) or $t \to t'$ with $\vdash t' : T$ (continue). The process cannot get stuck because Progress guarantees a step is always available if the term is not yet a value. □

Admissibility Preservation Theorem

Theorem 40.8 (Admissibility Preservation Across Reduction)**.** *If $\Gamma \vdash t : \mathsf{Admissible}(T)$ and $t \to^* v$ where v is a value, then $\vdash v : \mathsf{Admissible}(T)$ and the history produced by evaluating t contains no refusal events that would invalidate the admissibility certificate.*

Proof. By Full Preservation (Theorem 40.5), $\mathsf{Admissible}(T)$ is preserved throughout reduction. When v is reached, it has type $\mathsf{Admissible}(T)$ and by Canonical Forms, $v = \mathsf{pop}(v')$ with $\vdash v' : T$. The admissibility type $\mathsf{Admissible}(T)$ certifies that the path from t to v was admissible: no Refuse event along the way was applied to the same symbol as any Pop in the path, because such a conflict would create a type error at the Pop introduction site. □

Chapter 41

Extended Type Examples and Applications

Typing File System Operations

A concrete example grounds the type theory. File system operations provide a natural domain because they involve admissibility (file must exist), refusal (access denied), and collapse (reading produces an observable state).

Example 41.1 (File Read as Typed Spherepop Term).

Listing 41.1: File read in typed Spherepop

```
let read_file : FilePath -> Adm(FileContents) =
  fn path : FilePath.
    if file_exists(path) then
      pop(file_contents(path))   -- Adm(FileContents)
    else
      refuse(path, violation("FileNotFound"))
      -- Refused(FilePath, violation("FileNotFound"))

-- The return type is: Adm(FileContents) | Refused(FilePath,
   r)
-- The caller must handle both branches
```

The type signature forces the caller to acknowledge that reading can fail. Unlike an exception system that hides the failure mode in runtime behavior, the Spherepop type makes it explicit: the reason for refusal (FileNotFound) is part of the type, enabling callers to pattern-match on specific failure modes.

Typing Memory Safety

Example 41.2 (Pointer Dereference with Admissibility Certificate).

Listing 41.2: Safe pointer dereference

```
type Ptr(a : Type) = Adm(Reference(a))
```

```
-- A Ptr is an admissibility certificate that
-- the reference is valid and live.

let deref : Ptr(a) -> Adm(a) =
  fn p : Ptr(a).
    match p with
    | pop(ref) ->
        -- ref : Reference(a), known admissible
        pop(load(ref))    -- Adm(a)

let free_ptr : Ptr(a) -> Refused(Reference(a), "Deallocated")
    =
  fn p : Ptr(a).
    match p with
    | pop(ref) ->
        refuse(ref, explicit("Deallocated"))
-- After free_ptr, the ptr's type becomes Refused,
-- and any further deref is a type error:
-- deref requires Adm, not Refused.
```

Memory safety is enforced by types: after `free_ptr`, the pointer's type is Refused(Reference(a), Deal and deref requires Admissible(Reference(a)). A use-after-free is a type error caught at compile time.

Typing Cryptographic Operations

Listing 41.3: Cryptographic hash as collapse

Example 41.3 (Hash as Collapse with Fixed Rule).

```
type SHA256Rule = CollapseRule::Proj("sha256")

let hash : Adm(Bytes) -> Collapsed(Bytes, SHA256Rule) =
  fn b : Adm(Bytes).
    collapse(b, SHA256Rule)
-- The output type Collapsed(Bytes, SHA256Rule) certifies:
-- (1) the input was admissible (had type Adm(Bytes)),
-- (2) the specific collapse rule SHA256 was applied,
-- (3) the observation is sealed: no further pops can unwrap
   it.
```

```
-- Preimage resistance is expressed by the absence of a term:
-- there is no well-typed term of type
--   Collapsed(Bytes, SHA256Rule) -> Adm(Bytes)
-- without additional information.
```

The irreversibility of cryptographic hashing is expressed by the type system: $\mathsf{Collapsed}(\mathsf{Bytes}, \mathsf{SHA256})$ has no inverse in the type theory (since [T-COLLAPSE] has no elimination form that restores $\mathsf{Admissible}$). This makes preimage resistance a type-level property.

Part XVII

Variational Mechanics and Information Dynamics

Chapter 42

Histories as Paths Through Possibility Space

The Action Functional

A Spherepop computation is a path through event space. Like a physical path in classical mechanics, we can assign an action to each history and ask which histories are stationary points of this action—the computational analogues of geodesics.

Definition 42.1 (Event Cost Function)**.** An *event cost function* is a map $L : E \to \mathbb{R}_{\geq 0}$ assigning a non-negative cost to each primitive event. Common choices:

$$\begin{aligned} L(\mathsf{Pop}(x)) &= c_{\text{commit}} > 0 \quad \text{(cost of commitment)} \\ L(\mathsf{Refuse}(x, r)) &= c_{\text{refuse}} \geq 0 \quad \text{(cost of refusal)} \\ L(\mathsf{Bind}(a, b)) &= c_{\text{bind}} \geq 0 \quad \text{(cost of dependency)} \\ L(\mathsf{Collapse}(x, c)) &= c_{\text{obs}} > 0 \quad \text{(cost of observation)} \end{aligned}$$

Definition 42.2 (History Action)**.** The *action* of a history $H = [e_1, \ldots, e_n]$ is

$$\mathcal{S}(H) = \sum_{i=1}^{n} L(e_i).$$

Definition 42.3 (Admissible Path Space)**.** For boundary conditions (H_0, H_f) (initial and final histories), the *admissible path space* is

$$\mathcal{P}(H_0, H_f) = \{H \in \mathcal{H} \mid H_0 \text{ is a prefix of } H, H_f \text{ is a suffix of } H, \mathcal{A}(H) \neq \varnothing\}.$$

Discrete Euler-Lagrange Equations

Definition 42.4 (History Variation)**.** A *variation* of history H is a modified history $H + \delta H$ obtained by inserting, deleting, or replacing a single event while

keeping boundary conditions fixed.

Definition 42.5 (Stationary History)**.** A history H is *stationary* in $\mathcal{P}(H_0, H_f)$ if for all admissible variations δH,

$$\mathcal{S}(H + \delta H) - \mathcal{S}(H) \geq 0.$$

That is, H is a local minimum of the action functional.

Theorem 42.6 (Discrete Euler-Lagrange Conditions)**.** *A history $H = [e_1, \ldots, e_n]$ is stationary if and only if for each interior event e_k $(1 < k < n)$, replacing e_k with any admissible event e'_k satisfies $L(e'_k) \geq L(e_k)$.*

Proof. In the discrete setting, a variation that replaces e_k with e'_k changes the action by $L(e'_k) - L(e_k)$. Stationarity requires this to be non-negative for all admissible e'_k, which means e_k is the minimum-cost admissible event at position k. This is the discrete analogue of the Euler-Lagrange equation: the optimal trajectory selects, at each step, the event that minimizes the local cost subject to the admissibility constraint. □

Remark 42.7 (Connection to Admissibility Geometry). The admissibility constraint $\mathcal{A}(H) \neq \varnothing$ plays the role of the geometric constraint in constrained variational mechanics. A history that exhausts all admissible options ($\mathcal{A}(H) = \varnothing$) is a terminal trajectory—the computation has committed to all available possibilities. The xylomorphic criterion $\lambda < 1$ from the RSVP framework is the continuum version of this constraint: a trajectory remains in the admissible region as long as the local admissibility curvature $\kappa_A(x) > \lambda$.

Least-Action Computation

Theorem 42.8 (Least-Action Principle for Computation)**.** *A Spherepop computation that minimizes the total action $\mathcal{S}(H)$ over all admissible paths between boundary conditions performs the minimum-cost sequence of commitments and observations needed to reach the terminal state.*

Proof. By the Euler-Lagrange conditions, minimizing $\mathcal{S}(H)$ requires selecting the minimum-cost admissible event at each step. This is precisely greedy optimal computation: choose the cheapest admissible operation at each decision point. The resulting history is the discrete geodesic in event space connecting the boundary conditions. □

Example 42.9 (Parsing as Least-Action Computation)**.** A parser that recognizes a string against a grammar can be viewed as a Spherepop computation minimizing the action. Each match of a terminal symbol is a Pop event; each failed match is a Refuse event. The parser's job is to find the minimum-action path through the grammar's event space that accepts the input string. The optimal parser (in the sense of minimal backtracking) corresponds to the least-action history. PEG parsers implement this by trying alternatives in order and committing (Pop) on the first match, rather than exploring all possibilities.

Chapter 43

Information-Theoretic Properties of Histories

History Entropy

Definition 43.1 (History Distribution)**.** A *history distribution* over event alphabet E is a probability distribution μ over $\mathcal{H}$ such that $\mu(H)$ represents the probability of a specific execution path.

Definition 43.2 (History Entropy)**.** The *Shannon entropy* of a history distribution μ is

$$\mathbb{H}[\mu] = -\sum_{H \in \mathcal{H}} \mu(H) \log \mu(H).$$

Theorem 43.3 (Commitment Reduces Entropy)**.** *Each Pop event strictly reduces the history entropy: if μ' is the distribution over extensions of H after $\mathsf{Pop}(x)$, then $\mathbb{H}[\mu'] < \mathbb{H}[\mu]$.*

Proof. The Pop event $\mathsf{Pop}(x)$ eliminates all histories in μ that do not have x committed. The surviving histories are conditioned on x having been committed, reducing the support of the distribution. A distribution over a strictly smaller support has strictly lower entropy (since the eliminated histories had positive probability under μ). □

Theorem 43.4 (Refusal Does Not Reduce History Entropy)**.** *A Refuse event $\mathsf{Refuse}(x, r)$ does not reduce the entropy of the history distribution: the option space Ω is unchanged, and the support of the future-history distribution is unchanged (any future history compatible with $\mathcal{A}(H')$ was already compatible with $\mathcal{A}(H) \setminus \{x\}$).*

Proof. $\mathsf{Refuse}(x, r)$ appends an event to the history (increasing its complexity) but does not remove x from Ω. Future histories can still attempt $\mathsf{Pop}(x)$ (it will be caught by the admissibility check), but they can also avoid x entirely. The future history space is therefore unchanged in size, and entropy is preserved. □

Remark 43.5 (The Commitment-Refusal Asymmetry). Theorem 36.2 and 36.3 to-

gether formalize the fundamental asymmetry between Pop and Refuse. Pop is entropy-reducing: it commits to a possibility and forecloses alternatives, shrinking the future history space. Refuse is entropy-neutral: it documents that a path was inadmissible without shrinking the space of future histories. This asymmetry is why refusals are cheaper to make than commitments and why documented refusal (rather than silent avoidance) is a genuine contribution to the history record.

Hamming Distance on Histories

Definition 43.6 (History Edit Distance)**.** The *edit distance* between histories H_1 and H_2 is the minimum number of event insertions, deletions, and substitutions needed to transform H_1 into H_2:

$$d_{\text{edit}}(H_1, H_2) = \min\{k \mid H_1 \xrightarrow{k \text{ edits}} H_2\}.$$

Definition 43.7 (History Hamming Distance)**.** For histories of equal length n, the *Hamming distance* is

$$d_H(H_1, H_2) = |\{i \mid e_{1,i} \neq e_{2,i}\}|.$$

Theorem 43.8 (Error-Correcting Bounds for Histories)**.** *A set of histories $\mathcal{C} \subseteq \mathcal{H}^n$ (a history code) can correct up to t event errors (substitutions) if and only if*

$$\min_{H_1 \neq H_2 \in \mathcal{C}} d_H(H_1, H_2) \geq 2t + 1.$$

Proof. This is the standard Hamming bound applied to the history alphabet. If the minimum pairwise Hamming distance is $2t + 1$, then any two codeword-histories differ in at least $2t + 1$ positions. An error pattern affecting at most t positions cannot move any codeword to within Hamming distance t of any other codeword, so the nearest codeword is always the correct one. □

Remark 43.9 (Connection to Error Correction Chapter)*.* The history error-correcting bound connects the abstract history theory to the Error Correction as Certified Refusal chapter of Part VI. A set of redundant histories (encoding the same computation multiple times) forms a history code. Event corruption (bit errors in the history log) can be corrected if the minimum Hamming distance of the code exceeds twice the error rate. The certified refusal system then serves as the

decoder: when a Refuse event indicates that a committed symbol's certificate does not match the redundant record, the decoder identifies and corrects the corrupted event.

Kolmogorov Complexity and History Compression

Definition 43.10 (Kolmogorov Complexity of a History)**.** The *Kolmogorov complexity* of a history H is

$$K(H) = \min\{|p| \mid U(p) = H\}$$

where U is a universal Turing machine and $|p|$ is the length of the shortest program p that outputs H.

Proposition 43.11 (Incompressible Histories Exist)**.** *For each length n, there exist histories H of length n with $K(H) \geq n$. These are* random histories *whose event sequences contain no exploitable regularity.*

Proof. There are $|E|^n$ histories of length n but at most 2^{n-1} programs of length less than n bits. By counting, most histories cannot be compressed. □

Theorem 43.12 (Generative Notes Reduce Kolmogorov Complexity)**.** *A generative note N that enables a new inferential framework reduces the Kolmogorov complexity of the histories generated within that framework: $K(H \mid N) < K(H)$ for histories H expressible concisely within N.*

Proof. Given N as a reference, the shortest program for H can invoke N as a subroutine. If N provides a compression for the structure of H (for example, H is a theorem in a formal system captured by N), then $|p| < K(H)$ where p invokes N and provides only the theorem-specific details. Hence $K(H \mid N) \leq |p| < K(H)$. □

Remark 43.13. This theorem gives a Kolmogorov complexity account of why generative notes have high reachability leverage. A programming language, a mathematical theory, or a grammar compresses the Kolmogorov complexity of all programs, theorems, or strings expressible within it. The note is a reference object that reduces the minimum description length of the histories it enables.

Part XVIII

Applications and Connections to Existing Frameworks

Chapter 44

Event Sourcing and Database Theory

Event Sourcing as Spherepop

Event sourcing [15] is an architectural pattern where application state is derived from an append-only log of events rather than a mutable state store. Spherepop is the formal theory underlying event sourcing.

Theorem 44.1 (Event Sourcing Is Spherepop with LastWrite Collapse)**.** *An event-sourced application using an append-only event log is precisely a Spherepop world* (H, Ω) *with the LastWrite collapse rule* c_{LW} *applied to derive the current application state.*

Proof. The event log is the history H: each event appended to the log is a Pop event in the Spherepop world. The current application state is $c_{LW}(H)$: the most recent value for each entity, derived by folding the history through the LastWrite rule. The event sourcing invariant—that the current state can always be reconstructed from the log—is Spherepop's Replay Equivalence (Theorem 10.3): $I(H) = V(C(H))$, meaning the interpreter and VM produce identical histories and hence identical LastWrite states.

Event sourcing's "projection" (computing a read model from the event log) is a specific collapse rule applied to the history. A CQRS read model is a $c_{\text{projection}}$ collapse that extracts the data relevant to specific query patterns. □

Example 44.2 (Bank Account as Spherepop World)**.** A bank account event log in event sourcing corresponds to the following Spherepop world:

Listing 44.1: Bank account as Spherepop world

```
// Each transaction is a Pop event
let history = vec![
    Event::Pop(Symbol::new("deposit:100")),
    Event::Pop(Symbol::new("withdraw:30")),
    Event::Pop(Symbol::new("deposit:50")),
```

```
    Event::Refuse(Symbol::new("withdraw:200"),
                  RefusalReason::ConstraintViolation(
                    "InsufficientFunds")),
];

// Balance is the Accumulate collapse
// deposits positive, withdrawals negative
let balance = apply_collapse(&history, &acc_rule);
// balance = 120
```

The refusal event for the insufficient-funds withdrawal is preserved in the history with its reason, making the audit trail complete. A state-centric database would only show the final balance; Spherepop shows every decision.

Provenance Semirings

Provenance semirings [32] are algebraic structures that track the derivation of database query results from input tuples. They generalize boolean (does this tuple appear?), counting (how many times?), and probability (with what likelihood?) provenance.

Theorem 44.3 (Collapse Rules Are Provenance Semirings)**.** *Each Spherepop collapse rule $c : \mathcal{H} \to O_c$ induces a provenance semiring $(O_c, \oplus, \otimes)$ where:*

$$o_1 \oplus o_2 = c(H_1 \sqcup H_2) \quad \textit{(union provenance)}$$
$$o_1 \otimes o_2 = c(H_1 +\!+ H_2) \quad \textit{(join provenance)}$$

Proof. The Accumulate rule c_{acc} gives the counting semiring $(\mathbb{N}, +, \times)$: the count of how many derivations produced each fact is the sum of counts from alternative derivations (union) and the product of counts from joined derivations. The LastWrite rule gives the boolean semiring $(\{0, 1\}, \vee, \wedge)$: a fact holds if any derivation produces it (union is OR) and if all required sub-derivations succeed (join is AND). Different collapse rules give different provenance semantics, all arising from the same Spherepop history. □

The Differential Privacy Connection

Differential privacy requires that no single individual's data significantly affects the observable output of a computation. In Spherepop terms, this is a constraint

on the collapse rule's sensitivity to individual Pop events.

Definition 44.4 (ε-Differential Privacy in Spherepop)**.** A collapse rule c satisfies *ε-differential privacy* if for any two histories H and H' differing in exactly one Pop event, and for any observable set $O \subseteq O_c$:

$$P(c(H) \in O) \leq e^{\varepsilon} \cdot P(c(H') \in O).$$

Proposition 44.5 (Noisy Accumulate Is Differentially Private)**.** *The Accumulate rule with Laplace noise* $\mathsf{Lap}(1/\varepsilon)$ *added to each count satisfies ε-differential privacy.*

Proof. The sensitivity of c_{acc} to a single Pop event is 1 (one event changes exactly one count by exactly 1). Adding $\mathsf{Lap}(1/\varepsilon)$ noise calibrated to this sensitivity produces a differentially private mechanism by the standard Laplace mechanism analysis. □

Chapter 45

Programming Language Theory Connections

Denotational Semantics as Collapse

Denotational semantics assigns mathematical meanings (denotations) to programs. In Spherepop, denotational semantics corresponds to applying a collapse rule to the operational history.

Theorem 45.1 (Denotational Semantics Is a Collapse Functor)**.** *Every compositional denotational semantics* $[\![\cdot]\!] : \mathsf{Terms} \to \mathsf{Meanings}$ *is a collapse functor* $F_c : \mathbf{Hist} \to \mathbf{Obs}_c$ *for an appropriate collapse rule* c*.*

Proof. Compositionality requires that $[\![P \parallel Q]\!] = [\![P]\!] \oplus [\![Q]\!]$ for some meaning composition operator $\oplus$. This is precisely the functor composition law $F_c(H_1 +\!+ H_2) = F_c(H_1) \circ F_c(H_2)$ of the collapse functor. The denotation is the observable state produced by applying c to the operational history. Different denotational semantics correspond to different collapse rules: the trace semantics uses the Identity rule; the collecting semantics uses the Accumulate rule; the big-step semantics uses the LastWrite rule. □

Hoare Logic as Admissibility Typing

Hoare logic [19] provides pre- and postconditions for program statements: $\{P\}\, C\, \{Q\}$ asserts that if P holds before executing C, then Q holds after. In Spherepop, Hoare triples are admissibility typing judgements.

Theorem 45.2 (Hoare Triple as Spherepop Typing)**.** *The Hoare triple* $\{P\}\, C\, \{Q\}$ *is equivalent to the Spherepop typing judgement*

$$P \vdash C : \mathsf{Admissible}(Q)$$

where P *is a predicate on admissibility sets and* Q *is the postcondition certified by the*

admissibility type.

Proof. The precondition P specifies the admissibility set before C executes: $\mathcal{A}(H) \supseteq \text{Required}(P)$. The program C corresponds to a sequence of Spherepop operations that, when typed in context P, produces a result of type $\mathsf{Admissible}(Q)$. The admissibility certificate $\mathsf{Admissible}(Q)$ certifies that the postcondition Q holds after execution. The standard Hoare logic rules (consequence, composition, conditional, loop) correspond to the standard Spherepop typing rules under this identification. □

Separation Logic and Bind

Separation logic extends Hoare logic with a spatial connective: $P * Q$ asserts that P and Q hold over disjoint portions of memory. In Spherepop, this corresponds to independent Bind structures over disjoint option spaces.

Theorem 45.3 (Separating Conjunction as Disjoint Bind)**.** *The separating conjunction $P * Q$ in separation logic corresponds to the Spherepop assertion that P and Q hold over disjoint option spaces $\Omega_P \cap \Omega_Q = \varnothing$, with no Bind events coupling Ω_P and Ω_Q.*

Proof. The frame rule of separation logic—$\{P\}\, C\, \{Q\}$ implies $\{P * R\}\, C\, \{Q * R\}$ for any R not modified by C—corresponds to the Spherepop property that operations on Ω_C do not affect the admissibility certificates over Ω_R when $\Omega_C \cap \Omega_R = \varnothing$ and no Bind events cross the boundary. The spatial separation in memory corresponds to option-space separation in Spherepop. □

Chapter 46

Connections to Cognitive Science and MEM|8

The Spherepop Model of Memory Retrieval

The MEM|8 architecture [7] treats memory as wave-based pattern activation rather than symbol lookup. This section develops the formal connection between Spherepop histories and MEM|8 memory dynamics.

Definition 46.1 (Memory as History Distribution)**.** In the MEM|8 framework, a memory trace corresponds to a history distribution μ_M over the history space $\mathcal{H}$: the distribution represents the set of histories compatible with the stored memory and their relative activation strengths.

Theorem 46.2 (Memory Retrieval as Conditional Collapse)**.** *Memory retrieval in MEM|8 corresponds to conditioning the history distribution μ_M on an observational cue o:*

$$\mu_{\text{retrieved}} = \mu_M(\cdot \mid c^{-1}(o))$$

where c is the collapse rule determining what aspects of the memory are visible to the cue.

Proof. MEM|8 retrieval works by pattern completion: a partial cue activates the wave patterns associated with the complete memory. In Spherepop terms, the cue o specifies an observable state, and retrieval reconstructs the most probable history in the fibre $c^{-1}(o)$. This is Bayesian inference over the history distribution conditioned on the cue observation—precisely the conditional collapse $\mu_M(\cdot \mid c^{-1}(o))$. □

Example 46.3 (Episodic Memory as Identity-Rule Collapse)**.** Episodic memory (memory for specific events, including their temporal and contextual details) corresponds to near-Identity collapse: the memory trace preserves the full event sequence H, and retrieval under Identity rule c_I returns the original history. The wave-based pattern in MEM|8 stores the temporal ordering and contextual bind-

ings of events—equivalent to preserving H up to small perturbations.

Example 46.4 (Semantic Memory as Accumulate-Rule Collapse)**.** Semantic memory (knowledge about concepts and categories, abstracted from specific episodes) corresponds to Accumulate collapse: the memory trace stores the count of how many times each concept was encountered, losing the specific episodic context. MEM|8's categorical wave patterns represent this statistical summary across episodes.

Forgetting as Controlled Collapse

Definition 46.5 (Forgetting Rule)**.** A *forgetting rule* $c_{\text{forget}}(t)$ is a time-parameterized collapse rule where the coarseness of the collapse increases with time t: for $t_1 < t_2$, $c_{\text{forget}}(t_1)$ is finer than $c_{\text{forget}}(t_2)$.

Theorem 46.6 (Ebbinghaus Forgetting as Entropy Growth)**.** *Under a forgetting rule $c_{\text{forget}}(t)$, the observational entropy $S_{c(t)}(o)$ of any fixed observation o is non-decreasing in t:*

$$t_1 < t_2 \implies S_{c(t_1)}(o) \leq S_{c(t_2)}(o).$$

Proof. By Entropy Monotonicity Under Rule Refinement (Theorem of Part IX): since $c_{\text{forget}}(t_2)$ is coarser than $c_{\text{forget}}(t_1)$, the fibre of $c_{\text{forget}}(t_2)$ is larger, hence its log is larger. Forgetting coarsens the collapse rule, increasing the number of histories consistent with the retained observation, which is precisely what the Ebbinghaus forgetting curve describes: more historical detail is lost, more alternative histories become consistent with the remaining memory trace. □

Priming as Pre-emptive Admissibility Adjustment

Definition 46.7 (Priming Event)**.** A *priming event* is a Refuse event that marks certain continuations as temporarily less admissible and a Bind event that increases the admissibility curvature of related continuations.

Theorem 46.8 (Semantic Priming as Admissibility Field Reshaping)**.** *Semantic priming (the facilitated retrieval of concepts related to a recently activated concept) corresponds to a temporary reshaping of the admissibility field κ_A in the direction of the primed concept: symbols related to the prime have increased κ_A (higher admissibility) while unrelated symbols are temporarily Refused.*

Proof. Priming in cognitive science increases the activation of related concepts by lowering their retrieval threshold. In Spherepop, this corresponds to tem-

porarily increasing $\kappa_A(x)$ for symbols x related to the prime (making them more admissible, i.e., lower-cost to Pop), and decreasing $\kappa_A(y)$ for unrelated symbols (making them relatively less admissible). The decay of the priming effect over time corresponds to κ_A returning to its baseline, modeled by the forgetting rule. □

Chapter 47

Connections to Physics and Complex Systems

Thermodynamic Interpretation of Collapse

The relationship between information-theoretic entropy and thermodynamic entropy runs deep. The Spherepop collapse framework provides a computational analogue of thermodynamic processes.

Theorem 47.1 (Second Law as Entropy Non-Decrease Under Coarsening)**.** *Under any physical measurement* (*coarsening of observational framework from c_1 to c_2 with c_2 coarser*)*, the observational entropy of any history does not decrease:*

$$S_{c_2}(o) \geq S_{c_1}(o').$$

This is the computational Second Law: every measurement loses information.

Proof. By Entropy Monotonicity Under Rule Refinement (Part IX Chapter 22): coarsening increases fibre size and hence entropy. Physical measurement is a collapse (projecting the quantum state onto a classical observable), and each measurement can only coarsen the observational framework. □

Remark 47.2 (Maxwell's Demon as Refusal Without Cost). Maxwell's Demon in thermodynamics is an entity that observes individual molecules and opens or closes a gate based on their velocities, apparently violating the Second Law. In Spherepop terms, the Demon performs Refuse operations: marking slow molecules as inadmissible for passage through the gate. The resolution (Landauer's principle) is that the Demon's memory erasure has thermodynamic cost. In Spherepop, this corresponds to GC: erasing the Demon's history of observations (applying Certificate-Only GC to the Refuse records) has a minimum computational cost of $k_B T \ln 2$ per bit erased. The Refuse events cannot be erased for free.

Phase Transitions in Note Infrastructure

Definition 47.3 (Note Density)**.** The *note density* of a civilization at time t is

$$\rho_N(t) = \frac{|R(Z(t))|}{|\mathcal{C}_{\text{total}}|}$$

the fraction of all possible continuations that are reachable given the civilization's note infrastructure $\mathcal{N}(t)$.

Theorem 47.4 (Percolation Threshold in Note Networks)**.** *Note infrastructure undergoes a percolation phase transition: for note density ρ_N below a critical threshold ρ_c, the reachability graph of the civilization's knowledge is fragmented into isolated components. Above ρ_c, a giant connected component emerges, and most knowledge becomes mutually reachable.*

Proof sketch. The reachability graph has nodes for continuations and edges for notes that enable traversal between them. As note density increases from 0 to 1, this random graph undergoes the Erdős-Rényi percolation transition: for $\rho_N < \rho_c$, the largest connected component has size $O(\log n)$; for $\rho_N > \rho_c$, it has size $\Theta(n)$. The critical density ρ_c depends on the structure of the continuation space. □

Remark 47.5. The invention of writing corresponds to a transition through ρ_c in human civilizational history: before writing, knowledge was stored in individual memories with high loss rates (note density below threshold). Writing created persistent notes with much lower loss rates, enabling the civilization's reachability graph to transition into its giant-component phase. The emergence of the internet represents another such transition, this time increasing not just persistence but connectivity between note systems.

Part XIX

Philosophical Foundations

Chapter 48

Ontology of Process Versus Substance

The Substance Tradition

Western metaphysics from Aristotle through Kant has predominantly held that the primary entities of reality are *substances*: things that persist through time and bear properties. On this view, a rock, a person, a number, a program are all substances—entities that exist, that have attributes, and whose identity persists across change.

The substance tradition generates characteristic puzzles. What makes a ship that has had all its planks replaced the same ship (the Ship of Theseus)? What makes a person who has changed all their beliefs the same person? What makes a program that has been refactored the same program? These puzzles arise because the substance view treats identity as primitive and change as something that happens to an otherwise stable entity.

The Spherepop ontology inverts this. Histories are primary; substances are derived. Change is not something that happens to an entity; entities are persistent patterns in a history of events. The puzzles of identity through change dissolve: the ship is identified by its history, not by its material composition at any moment. The person is identified by their developmental history, not by their current beliefs. The program is identified by its development history (the Git repository), not by its current source text.

Process Philosophy Formalized

Process philosophy, developed by Whitehead and later by Rescher, holds that processes rather than substances are the fundamental entities of reality. Spherepop provides a formal framework for this tradition.

Definition 48.1 (Process)**.** In the Spherepop ontology, a *process* is a history $H \in$

$\mathcal{H}$: a finite sequence of events that constitutes a trajectory through possibility space. Processes are the fundamental entities. Substances are derived.

Definition 48.2 (Substance as Persistent Process)**.** A *substance* is an equivalence class $[H]_c$ under a stable collapse rule c: a family of histories whose observable state $c(H)$ remains constant over admissible extensions. Substances are persistent processes—processes whose observable character does not change as they evolve.

Theorem 48.3 (Process-Substance Duality)**.** *There is a categorical duality between the category* **Hist** (*processes as morphisms*) *and the category* $\mathbf{Obs}_c$ (*substances as objects*)*: the collapse functor* $F_c : \mathbf{Hist} \to \mathbf{Obs}_c$ *exhibits* $\mathbf{Obs}_c$ *as the category of stable orbits of* **Hist** *under the equivalence* $\sim_c$*.*

Proof. Objects of $\mathbf{Obs}_c$ are equivalence classes $[H]_c$—precisely the persistent processes (substances). Morphisms of $\mathbf{Obs}_c$ are observable transitions between these classes. The functor F_c maps each process (morphism of **Hist**) to its observable substance (object of $\mathbf{Obs}_c$). This is the formal statement of the Noun Fallacy: objects of $\mathbf{Obs}_c$ appear to be the primary entities, but they are images of morphisms of **Hist** under F_c. □

Identity Through Change

Theorem 48.4 (Ship of Theseus Resolution)**.** *In the Spherepop ontology, the Ship of Theseus has a determinate identity at every point in its history. Two configurations of the ship are the same ship if and only if they appear in the same continuous history.*

Proof. The ship is identified by its history H, not by its material composition at any moment. Two configurations H_t and $H_{t'}$ are configurations of the same ship if and only if H_t is a prefix of $H_{t'}$ (or vice versa) in the continuous developmental history. The gradual replacement of planks is a sequence of events in the history; the ship's identity is the continuous history, not any particular state $c(H_t)$ at a moment. □

Chapter 49

Language, Meaning, and Semantic Collapse

Meaning as Reachability

The semantics of natural language has been understood primarily in terms of truth conditions (what makes a sentence true) or use conditions (what role a sentence plays in a practice). The Spherepop framework suggests a third account: meaning as reachability.

Definition 49.1 (Meaning as Continuation Set)**.** The *meaning* of a linguistic expression E is the set of continuations $\mathcal{C}(E)$ that E makes reachable for a competent interpreter:

$$[\![E]\!] = \mathcal{C}(E) = \{c \in \mathcal{C} \mid P_A(c, t \mid E) > P_A(c, t)\}.$$

Theorem 49.2 (Synonymy as Continuation Equivalence)**.** *Two expressions E_1 and E_2 are synonymous if and only if $\mathcal{C}(E_1) = \mathcal{C}(E_2)$: they make the same set of continuations reachable.*

Proof. If $\mathcal{C}(E_1) = \mathcal{C}(E_2)$, then for any agent A and continuation c, $P_A(c, t \mid E_1) = P_A(c, t \mid E_2)$. The two expressions have identical effects on the reachability of all future actions, which is the strongest sense in which two expressions can mean the same thing. □

Theorem 49.3 (Semantic Drift as Collapse Rule Change)**.** *Semantic change over time—a word acquiring a new meaning or losing an old one—corresponds to a change in the collapse rule applied to the word's continuation set.*

Proof. A word w at time t_1 has meaning $[\![w]\!]_{c_1} = F_{c_1}(H_w)$ where H_w is the history of uses of w and c_1 is the community's collapse rule for interpreting uses. At time $t_2 > t_1$, the community may apply a different rule c_2 to the same history H_w, producing $[\![w]\!]_{c_2} = F_{c_2}(H_w)$. The word's meaning has changed not because the

word itself changed but because the observational framework for interpreting it changed. □

Translation as Approximate Adjoint

Theorem 49.4 (Translation as Partial Adjoint)**.** *Translation between languages L_1 and L_2 is a partial adjoint between the semantic categories $\mathsf{Sem}(L_1)$ and $\mathsf{Sem}(L_2)$: it preserves continuation sets as closely as possible given the structural differences between the languages.*

Proof. A translation $T_{12} : L_1 \to L_2$ maps expressions in L_1 to expressions in L_2. It is an approximate right adjoint to the meaning functor if $\mathcal{C}(T_{12}(E)) \approx \mathcal{C}(E)$ for all $E \in L_1$. Perfect translation would require $\mathcal{C}(T_{12}(E)) = \mathcal{C}(E)$ exactly, which fails when L_2 lacks the grammatical resources to express the continuation set of some $E \in L_1$ (a translation gap). The quality of a translation is measured by the closeness of the induced adjunction. □

Chapter 50

Ethics, Institutions, and Civilizational Responsibility

Moral Obligations as Coordination Notes

Theorem 50.1 (Moral Norms as Coordination Notes)**.** *Moral norms, insofar as they enable or constrain coordination between agents, are coordination notes in the Spherepop sense: they bind the continuation spaces of multiple agents, creating coupled trajectories from what would otherwise be independent evolutions.*

Proof. A moral norm N that prohibits action a (such as theft or deception) is a Refuse event applied to all agents: it marks action a as inadmissible with reason $\mathsf{MoralProhibition}(N)$. A moral norm that requires action a (such as keeping promises) is a Bind event: it couples the agent's current commitment to the future execution of a. Both types of norm are coordination notes: they synchronize agent trajectories, enabling cooperation and joint reachability that would be impossible without the shared normative framework. □

Remark 50.2. The formalization of moral norms as Spherepop operations has implications for moral epistemology. The claim that a norm is valid is the claim that the corresponding coordination note is genuine: that it actually increases collective reachability for the community that adopts it. Evaluating moral norms by their reachability consequences—rather than by their intrinsic properties or their conformity to ideal principles—is a version of moral consequentialism expressed in the Spherepop framework.

Institutional Failure as Note Collapse

Definition 50.3 (Institutional Integrity)**.** An institution has *integrity* with respect to its mandate M if its history H_{inst} preserves the admissibility certificates needed to continue executing M: $\mathcal{A}(H_{\text{inst}}) \supseteq \text{Required}(M)$.

Theorem 50.4 (Institutional Failure as Admissibility Collapse)**.** *An institution fails when its history accumulates refusal events that make its mandate inadmissible:* $\mathcal{A}(H_{\text{inst}}) \cap \text{Required}(M) = \varnothing$*. Institutional reform is a repair morphism that restores* $\mathcal{A}$ *to include* $\text{Required}(M)$*.*

Proof. By Refusal Monotonicity, the admissibility region $\mathcal{A}(H)$ can only shrink as the institution's history grows. If the history accumulates refusals of actions required by the mandate M, then eventually $\mathcal{A}(H) \cap \text{Required}(M) = \varnothing$: the institution can no longer execute its mandate. This is institutional failure. Reform corresponds to a repair morphism that replaces the inadmissible history with an alternative history H' in which the refused actions are no longer refused—restoring the admissibility of the mandate. □

Example 50.5 (Democratic Institutions)**.** A democratic institution's mandate includes the requirement that elections be held, results certified, and power transferred peacefully. If the institution's history accumulates refusals of these requirements (e.g., failure to certify elections, refusal to transfer power), then $\mathcal{A}(H) \cap \{\mathsf{Election}, \mathsf{Certification}, \mathsf{Transfer}\} = \varnothing$, and the institution has failed its democratic mandate. Restoration requires a repair morphism—new elections, constitutional reform, external oversight—that restores these requirements to the admissibility set.

Part XX

Implementation Architecture

Chapter 51

The Spherepop Interpreter in Full

Architecture Overview

The Spherepop interpreter is organized around the principle that the World structure—containing history and option space—is the primary runtime object. Observable state is never stored; it is always derived on demand. This section documents the full module architecture and its relationship to the theoretical results developed in preceding parts.

Listing 51.1: Complete World structure

```
pub struct World {
    pub history: History,
    pub options: OptionSpace,
    // Observable state is intentionally absent.
    // It is a method, not a field.
}

impl World {
    pub fn new(options: OptionSpace) -> Self {
        World { history: History::new(), options }
    }

    // Pop: commits to symbol, reduces option space
    pub fn pop(&mut self, sym: Symbol)
        -> Result<(), AdmissibilityError>
    {
        if !self.options.contains(&sym) {
            return Err(AdmissibilityError::NotInOptionSpace(
                sym));
        }
```

```
        if !self.admissible(&sym) {
            return Err(AdmissibilityError::PreviouslyRefused(
                sym));
        }
        self.history.append(Event::Pop(sym.clone()));
        self.options.remove(&sym);
        Ok(())
    }

    // Refuse: documents inadmissibility with reason
    pub fn refuse(&mut self, sym: Symbol, reason:
        RefusalReason) {
        self.history.append(Event::Refuse(sym, reason));
        // Option space unchanged: Refuse != Pop
    }

    // Bind: records dependency between symbols
    pub fn bind(&mut self, a: Symbol, b: Symbol) {
        self.history.append(Event::Bind(a, b));
    }

    // Collapse: derives observable state under rule
    pub fn observe(&self, rule: &CollapseRule) ->
        ObservableState {
        ObservableState::derive(&self.history, rule)
    }

    // Admissibility check
    fn admissible(&self, sym: &Symbol) -> bool {
        !self.history.events.iter().any(|e| matches!(
            e, Event::Refuse(s, _) if s == sym
        ))
    }

    // Conservation law invariant
    pub fn check_conservation(&self, initial_size: usize) ->
        bool {
        let history_weight: usize = self.history.events.iter
            ()
```

```
            .filter(|e| matches!(e, Event::Pop(_)))
            .count();
        self.options.len() + history_weight == initial_size
    }
}
```

The Evaluation Engine

Listing 51.2: Core evaluation loop

```
pub struct EvalResult {
    pub value: Value,
    pub world: World,
    pub admissible: bool,
    pub steps: usize,
}

pub fn eval(term: &Term, env: &Env, world: &mut World)
    -> Result<EvalResult, EvalError>
{
    let mut steps = 0;
    let result = eval_inner(term, env, world, &mut steps)?;
    Ok(EvalResult {
        value: result,
        world: world.clone(),
        admissible: world.history.is_admissible(),
        steps,
    })
}

fn eval_inner(
    term: &Term,
    env: &Env,
    world: &mut World,
    steps: &mut usize,
) -> Result<Value, EvalError> {
    *steps += 1;
    match term {
```

```
        Term::Var(x) => env.lookup(x).ok_or(EvalError::
            UnboundVar),
        Term::Lam(x, ty, b) => Ok(Value::Closure {
            param: x.clone(),
            param_ty: *ty.clone(),
            body: b.clone(),
            env: env.clone(),
        }),
        Term::App(f, a) => {
            let fv = eval_inner(f, env, world, steps)?;
            let av = eval_inner(a, env, world, steps)?;
            match fv {
                Value::Closure { param, body, env: cenv, .. }
                    => {
                    let env2 = cenv.extend(&param, av);
                    eval_inner(&body, &env2, world, steps)
                }
                _ => Err(EvalError::NotAFunction),
            }
        }
        Term::Pop(t) => {
            let v = eval_inner(t, env, world, steps)?;
            let sym = v.to_symbol()?;
            world.pop(sym.clone())?;
            Ok(Value::Admitted(Box::new(Value::Symbol(sym))))
        }
        Term::Refuse(t, r) => {
            let v = eval_inner(t, env, world, steps)?;
            let sym = v.to_symbol()?;
            world.refuse(sym.clone(), r.clone());
            Ok(Value::Refused(Box::new(Value::Symbol(sym)), r
                .clone()))
        }
        Term::Bind(a, b) => {
            let av = eval_inner(a, env, world, steps)?;
            let bv = eval_inner(b, env, world, steps)?;
            let asym = av.to_symbol()?;
            let bsym = bv.to_symbol()?;
            world.bind(asym.clone(), bsym.clone());
```

```
            Ok(Value::Bound(Box::new(av), Box::new(bv)))
        }
        Term::Collapse(t, rule) => {
            let v = eval_inner(t, env, world, steps)?;
            match v {
                Value::Admitted(inner) => {
                    let obs = world.observe(rule);
                    Ok(Value::Observed(inner, rule.clone(),
                       obs))
                }
                _ => Err(EvalError::
                   CollapseRequiresAdmissible),
            }
        }
        Term::Seq(terms) => {
            let mut last = Value::Unit;
            for t in terms {
                last = eval_inner(t, env, world, steps)?;
            }
            Ok(last)
        }
    }
}
```

The Observable State Derivation

Listing 51.3: Collapse rules as observable state derivation

```
#[derive(Clone, Debug)]
pub enum CollapseRule {
    Identity,
    LastWrite,
    Accumulate,
    Projection(String),
}

#[derive(Clone, Debug, PartialEq)]
pub struct ObservableState {
    pub entries: HashMap<String, ObsValue>,
```

```
    pub rule: CollapseRule,
}

impl ObservableState {
    pub fn derive(history: &History, rule: &CollapseRule)
        -> ObservableState
    {
        let mut entries = HashMap::new();
        match rule {
            CollapseRule::Identity => {
                // Full history: every event is visible
                for (i, event) in history.events.iter().
                   enumerate() {
                    entries.insert(
                        format!("event_{}", i),
                        ObsValue::Event(event.clone())
                    );
                }
            }
            CollapseRule::LastWrite => {
                // Most recent pop for each symbol
                for event in &history.events {
                    if let Event::Pop(sym) = event {
                        entries.insert(
                            sym.name.clone(),
                            ObsValue::Present
                        );
                    }
                    // Refusals after last pop remove symbol
                    if let Event::Refuse(sym, _) = event {
                        entries.remove(&sym.name);
                    }
                }
            }
            CollapseRule::Accumulate => {
                // Count of pops per symbol
                for event in &history.events {
                    if let Event::Pop(sym) = event {
                        let count = entries
```

```
                            .entry(sym.name.clone())
                            .or_insert(ObsValue::Count(0));
                        if let ObsValue::Count(n) = count {
                            *n += 1;
                        }
                    }
                }
            }
            CollapseRule::Projection(prefix) => {
                // Only events matching prefix
                for event in &history.events {
                    let name = event.symbol_name();
                    if name.starts_with(prefix.as_str()) {
                        entries.insert(
                            name.to_string(),
                            ObsValue::Event(event.clone())
                        );
                    }
                }
            }
        }
        ObservableState { entries, rule: rule.clone() }
    }

    pub fn equivalent(h1: &History, h2: &History,
                      rule: &CollapseRule) -> bool {
        Self::derive(h1, rule) == Self::derive(h2, rule)
    }
}
```

Chapter 52

The Compiler and Virtual Machine

The Event IR

Listing 52.1: Complete Event IR definition

```
#[derive(Clone, Debug, PartialEq)]
pub enum IrEvent {
    Pop,
    Refuse { reason: RefusalReason },
    Collapse { rule: CollapseRule },
    Bind,
    LoadSym(String),     // push symbol onto stack
    LoadVar(String),     // push variable binding
    Apply,               // function application
    LamIntro(String, Box<Type>),  // lambda introduction
    Seq(usize),          // sequence of n events
}

pub struct IrBlock {
    pub events: Vec<IrEvent>,
    pub source_map: Vec<usize>,  // event index -> source
        line
}
```

The Compiler

Listing 52.2: The Spherepop compiler

```
pub struct Compiler {
    pub output: IrBlock,
```

```
}

impl Compiler {
    pub fn compile(&mut self, term: &Term) {
        match term {
            Term::Var(x) =>
                self.output.emit(IrEvent::LoadVar(x.clone()))
                    ,
            Term::Lam(x, ty, body) => {
                self.output.emit(IrEvent::LamIntro(
                    x.clone(), ty.clone()
                ));
                self.compile(body);
            }
            Term::App(f, a) => {
                self.compile(f);
                self.compile(a);
                self.output.emit(IrEvent::Apply);
            }
            Term::Pop(t) => {
                self.compile(t);
                self.output.emit(IrEvent::Pop);
            }
            Term::Refuse(t, r) => {
                self.compile(t);
                self.output.emit(IrEvent::Refuse {
                    reason: r.clone()
                });
            }
            Term::Bind(a, b) => {
                self.compile(a);
                self.compile(b);
                self.output.emit(IrEvent::Bind);
            }
            Term::Collapse(t, rule) => {
                self.compile(t);
                self.output.emit(IrEvent::Collapse {
                    rule: rule.clone()
                });
```

```
            }
            Term::Seq(terms) => {
                let n = terms.len();
                for t in terms {
                    self.compile(t);
                }
                self.output.emit(IrEvent::Seq(n));
            }
        }
    }
}
```

The Virtual Machine

Listing 52.3: The Spherepop VM

```
pub struct Machine {
    pub world: World,
    pub stack: Vec<Value>,
    pub call_stack: Vec<(String, Box<Term>, Env)>,
}

impl Machine {
    pub fn run(&mut self, block: &IrBlock)
        -> Result<Value, MachineError>
    {
        for event in &block.events {
            match event {
                IrEvent::LoadSym(s) =>
                    self.stack.push(Value::Symbol(Symbol::new
                        (s))),
                IrEvent::LoadVar(x) => {
                    let v = self.current_env().lookup(x)
                        .ok_or(MachineError::UnboundVar)?;
                    self.stack.push(v);
                }
                IrEvent::Pop => {
                    let sym = self.stack.pop()
                        .ok_or(MachineError::StackUnderflow)?
```

```
                .to_symbol()?;
            self.world.pop(sym.clone())?;
            self.stack.push(
                Value::Admitted(Box::new(Value::
                    Symbol(sym)))
            );
        }
        IrEvent::Refuse { reason } => {
            let sym = self.stack.pop()
                .ok_or(MachineError::StackUnderflow)?
                .to_symbol()?;
            self.world.refuse(sym.clone(), reason.
                clone());
            self.stack.push(Value::Refused(
                Box::new(Value::Symbol(sym)), reason.
                    clone()
            ));
        }
        IrEvent::Bind => {
            let b = self.stack.pop()
                .ok_or(MachineError::StackUnderflow)
                    ?;
            let a = self.stack.pop()
                .ok_or(MachineError::StackUnderflow)
                    ?;
            let asym = a.to_symbol()?;
            let bsym = b.to_symbol()?;
            self.world.bind(asym.clone(), bsym.clone
                ());
            self.stack.push(Value::Bound(
                Box::new(a), Box::new(b)
            ));
        }
        IrEvent::Collapse { rule } => {
            let top = self.stack.pop()
                .ok_or(MachineError::StackUnderflow)
                    ?;
            match top {
                Value::Admitted(inner) => {
```

```
                        let obs = self.world.observe(rule
                            );
                        self.stack.push(Value::Observed(
                            inner, rule.clone(), obs
                        ));
                    }
                    _ => return Err(
                        MachineError::
                            CollapseRequiresAdmissible
                    ),
                }
            }
            IrEvent::Apply => {
                let arg = self.stack.pop()
                    .ok_or(MachineError::StackUnderflow)
                        ?;
                let func = self.stack.pop()
                    .ok_or(MachineError::StackUnderflow)
                        ?;
                match func {
                    Value::Closure { param, body, env, ..
                        } => {
                        let env2 = env.extend(&param, arg
                            );
                        let result = eval_inner(
                            &body, &env2, &mut self.world
                                ,
                            &mut 0
                        )?;
                        self.stack.push(result);
                    }
                    _ => return Err(MachineError::
                        NotAFunction),
                }
            }
            _ => {} // LamIntro, Seq handled by context
        }
    }
    self.stack.pop().ok_or(MachineError::EmptyStack)
```

```
    }
}
```

Replay Equivalence Test Suite

Listing 52.4: Replay equivalence verification

```
#[cfg(test)]
mod replay_tests {
    use super::*;

    fn assert_replay_equivalent(term: &Term) {
        let mut world_i = World::new(test_option_space());
        let result_i = eval(term, &Env::empty(), &mut world_i
            )
            .expect("interpretation␣failed");

        let mut compiler = Compiler::new();
        compiler.compile(term);
        let block = compiler.output;

        let mut machine = Machine::new(test_option_space());
        machine.run(&block).expect("VM␣execution␣failed");

        // THE CORRECTNESS CRITERION: history must be
            identical
        assert_eq!(
            world_i.history, machine.world.history,
            "Replay␣Equivalence␣violated:␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣I(p).history␣!=␣V(C(p)).history"
        );
    }

    #[test]
    fn replay_pop() {
        assert_replay_equivalent(&Term::Pop(
            Box::new(Term::Var("x".into()))
        ));
    }
```

```
#[test]
fn replay_refuse_with_reason() {
    assert_replay_equivalent(&Term::Refuse(
        Box::new(Term::Var("x".into())),
        RefusalReason::ConstraintViolation("c".into())
    ));
}

#[test]
fn replay_seq_bind_pop() {
    assert_replay_equivalent(&Term::Seq(vec![
        Term::Bind(
            Box::new(Term::Var("a".into())),
            Box::new(Term::Var("b".into()))
        ),
        Term::Pop(Box::new(Term::Var("a".into()))),
    ]));
}

#[test]
fn replay_collapse_after_pop() {
    assert_replay_equivalent(&Term::Collapse(
        Box::new(Term::Pop(
            Box::new(Term::Var("x".into()))
        )),
        CollapseRule::LastWrite,
    ));
}

#[test]
fn conservation_invariant_maintained() {
    let initial_size = 5;
    let mut world = World::new(OptionSpace::of_size(
       initial_size));
    world.pop(Symbol::new("a")).unwrap();
    world.pop(Symbol::new("b")).unwrap();
    world.refuse(Symbol::new("c"),
                 RefusalReason::NotAvailable);
```

```
        // Conservation: |H_pop| + |Omega| = initial_size
        assert!(world.check_conservation(initial_size));
    }
}
```

Part XXI

Extended Examples and Case Studies

Chapter 53

The Spherepop Type Checker as Its Own Subject

Self-Application

The most striking demonstration of a type system's coherence is its ability to type-check itself. The Spherepop type checker is written in a language that Spherepop can reason about, and the type-checking process is itself a Spherepop computation.

Example 53.1 (Type Checking as Collapse)**.** The type checker applies a specific collapse rule c_{type} to the history of term formation: each introduction rule ([T-POP], [T-REFUSE], etc.) is a Pop event in the type-derivation history, and the resulting type is the observable state under c_{type}.

Type errors are Refuse events: the type checker refuses a term with a reason ("collapse requires Admissible type, found Refused") that becomes part of the derivation history. The history of a type-checking run is itself a Spherepop history, auditable by applying different collapse rules to extract different aspects of the derivation.

Proof Checking as Admissibility Verification

Example 53.2 (Mathematical Proof as Spherepop Computation)**.** A formal mathematical proof in a system like Lean 4 can be expressed as a Spherepop computation. Each lemma application is a Pop event (committing to a specific inference step). Each assumption is a Bind event (recording the dependency on an axiom or hypothesis). Each application of modus ponens is a Collapse event (collapsing the conditional and its antecedent to the consequent under the modus-ponens rule).

The correctness of the proof corresponds to the Spherepop type: $\vdash \textit{proof} : \mathsf{Admissible}(T)$ where T is the statement being proved. A proof error is $\mathsf{Refused}(T, r)$

where r names the specific inference error (wrong lemma applied, type mismatch in substitution, etc.).

The history of the proof-checking computation is the complete audit trail of every inference step, every assumption used, and every definition invoked. This history is more informative than the proof term alone: it records not just *what* was proved but *how* the proof was discovered.

Chapter 54

Historical Case Studies

The Library of Alexandria as Note Collapse

Example 54.1 (The Library of Alexandria)**.** The Library of Alexandria (founded approximately 3rd century BCE, burned in successive incidents between 48 BCE and 642 CE) was the ancient world's largest note repository. At its peak, it held an estimated 400,000–700,000 scrolls, representing a significant fraction of the written knowledge of the Mediterranean world.

In Spherepop terms, the Library was a civilizational note infrastructure $\mathcal{N}_{\text{Alex}}$ whose destruction was a catastrophic note collapse. By the Civilizational Collapse Theorem:

$$R(Z_{\text{after}}) < R(Z_{\text{before}}).$$

The lost trajectories include works of Aristotle's that are known only through references in surviving texts (Capture Notes lost), mathematical texts by Euclid's contemporaries whose results are inaccessible (Refusal Notes lost—we cannot verify the admissibility of contemporary results against these benchmarks), and technical treatises on engineering and medicine that would have enabled trajectories in applied science that instead had to be rediscovered centuries later (Generative Notes lost).

The repair morphisms that partially addressed the loss—the preservation of Arabic translations of Greek texts, the transmission of some works through Byzantine copies, the reconstruction of some mathematical results by independent derivation—each restore specific fibers of the destroyed history, but none restores the full reachability of the original infrastructure.

Git as a Note Infrastructure

Example 54.2 (Git as Spherepop Implementation)**.** Git, the distributed version control system, is the most widely deployed instance of Spherepop principles in existing software infrastructure.

The Git commit graph is a history H_{git}: each commit is a Pop event that records a specific state of the codebase. Branches are parallel worlds (H_1, H_2) that diverge from a common ancestor. Merges are history merges $H_1 \sqcup H_2$. Tags are Collapse events that seal a specific state under a named rule.

Git's admissibility structure is enforced by the branch protection rules: a Refuse event is a rejected pull request, with a reason (failing CI, code review rejection, merge conflict). The reason is preserved in the Git history as a closed pull request with comments.

The collapse rules in Git correspond to:

- c_I (Identity): `git log --all` showing the full history.
- c_{LW} (LastWrite): the current working tree state.
- c_{acc} (Accumulate): `git shortlog` counting commits per author.
- π_L (Projection): `git log -- path/to/file` showing only events affecting a specific path.

The Replay Equivalence theorem (Theorem 10.3) is implemented in Git's architecture: checking out any commit and applying all subsequent commits must produce the same working tree state as checking out the latest commit directly. Git's content-addressable storage ensures this invariant by making commit hashes depend on the full history.

Chapter 55

Spherepop Programs as Executable Specifications

The Dining Philosophers in Spherepop

Example 55.1 (Dining Philosophers)**.** The classic Dining Philosophers problem (five philosophers, five forks, each philosopher needs two adjacent forks to eat) is a deadlock scenario expressible in the Spherepop process calculus.

Listing 55.1: Dining philosophers as Spherepop

```
// Five philosophers and five forks
let fork_options = OptionSpace::new([
    Symbol::new("fork0"), Symbol::new("fork1"),
    Symbol::new("fork2"), Symbol::new("fork3"),
    Symbol::new("fork4"),
]);

// Philosopher i picks up left fork (i) then right fork (i+1
   mod 5)
fn philosopher(i: usize, world: &mut World)
    -> Result<(), AdmissibilityError>
{
    let left = Symbol::new(&format!("fork{}", i));
    let right = Symbol::new(&format!("fork{}", (i + 1) % 5));
    // Try to acquire left fork
    world.pop(left.clone())?;
    // Try to acquire right fork
    match world.pop(right.clone()) {
        Ok(_) => {
            // Eat, then release
            world.refuse(left, RefusalReason::Explicit(
                "released".into()
```

```
            ));
            world.refuse(right, RefusalReason::Explicit(
                "released".into()
            ));
            Ok(())
        }
        Err(e) => {
            // Deadlock: right fork unavailable
            // Must release left and retry
            Err(e)
        }
    }
}
```

Deadlock is detected by the cycle detection algorithm (Theorem 34.2): when all five philosophers hold their left fork and wait for their right fork, the dependency graph G_H has the cycle $0 \to 1 \to 2 \to 3 \to 4 \to 0$. The Spherepop history preserves which philosopher acquired which fork in which order, making the deadlock state fully auditable.

A Blockchain as Spherepop History

Example 55.2 (Blockchain as Distributed Spherepop)**.** A blockchain is a distributed Spherepop world system where each block is a Collapse event that seals a batch of transactions (Pop events) under the Accumulate collapse rule, producing a cryptographic hash (the block hash) as the observable state.

The blockchain's consensus mechanism corresponds to global consistency (Definition 33.2): all nodes in the network must agree on the same history H and hence the same block hashes. A blockchain fork is a network partition that allows two sets of nodes to produce divergent histories $H_1 \neq H_2$. The longest-chain rule (in Bitcoin) is a specific merge strategy: when the partition heals, adopt the history with more accumulated proof-of-work, corresponding to the history with higher total action $\mathcal{S}(H)$ in the variational sense.

Smart contracts are Spherepop programs executed within the blockchain's history. They commit to outcomes (Pop), document failures (Refuse), record dependencies between contracts (Bind), and produce verifiable states (Collapse). The immutability of the blockchain history corresponds to the monotonicity of Spherepop history growth: events can be added but never removed.

Part XXII

Possibility Dynamics and Entropy

Chapter 56

Possibility as a Computational Resource

Possibility Entropy

The conservation law counts possibility discretely. A more informative invariant is obtained by passing from cardinality to entropy.

Definition 56.1 (Possibility Entropy)**.** The *possibility entropy* of a history H is

$$S_\Omega(H) = \log |\Omega(H)|.$$

When $\Omega(H) = \varnothing$, we set $S_\Omega(H) = -\infty$.

Theorem 56.2 (Pop Entropy Reduction)**.** *If* $H' = H +\!+ [\mathsf{Pop}(x)]$ *and* $x \in \Omega(H)$ *with* $|\Omega(H)| > 1$*, then*

$$S_\Omega(H') < S_\Omega(H).$$

Proof. Since Pop removes exactly one available symbol, $|\Omega(H')| = |\Omega(H)| - 1$. If $|\Omega(H)| > 1$, then $0 < |\Omega(H)| - 1 < |\Omega(H)|$. The logarithm is strictly increasing, so $\log(|\Omega(H)| - 1) < \log |\Omega(H)|$. Therefore $S_\Omega(H') < S_\Omega(H)$. □

Corollary 56.3 (Refuse Does Not Reduce Structural Entropy)**.** *If* $H' = H +\!+ [\mathsf{Refuse}(x, r)]$*, then* $S_\Omega(H') = S_\Omega(H)$*.*

Proof. Refuse appends an event but does not remove x from Ω. Thus $\Omega(H') = \Omega(H)$, so $S_\Omega(H') = \log |\Omega(H')| = \log |\Omega(H)| = S_\Omega(H)$. □

Remark 56.4. This separates two kinds of possibility loss. Pop reduces the option space: it commits to a specific future by removing alternatives. Refuse does not reduce structural availability; it records that a structurally available path is inadmissible. The two entropies—structural (S_Ω) and admissibility ($S_\mathcal{A}$)—track these independent dimensions.

Example 56.5 (Choosing from a Menu)**.** A diner faces a menu with n options. Each choice is a Pop event. Before choosing: $S_\Omega = \log n$. After choosing the

main course: $S_\Omega = \log(n-1)$. The dinner's preference rules (vegetarian, allergies) are Refuse events: they reduce $S_\mathcal{A}$ without affecting S_Ω. The restaurant still offers the refused dishes; the diner simply will not order them. This models real decisions precisely: structural and admissibility constraints are independent.

Example 56.6 (Compiler Optimization as Possibility Reduction)**.** A compiler transforms high-level source (high S_Ω: many possible execution paths) into optimized machine code (low S_Ω: fewer execution branches). Each optimization pass is a sequence of Pop events—selecting specific implementations for abstract operations—reducing possibility entropy while preserving the observable behavior (the collapse rule c applied to source and output produces the same observable state).

Example 56.7 (Biological Differentiation)**.** A pluripotent stem cell has high possibility entropy: it can differentiate into many cell types. Differentiation is a sequence of Pop events, each committing to specific gene expression patterns. A fully differentiated neuron has $S_\Omega \approx 0$: it has committed to its cell type. Apoptosis signals correspond to Refuse events: they mark developmental paths as inadmissible without immediately committing the cell to an alternative.

Example 56.8 (Legal Precedent as Pop)**.** A legal precedent is a Pop event in a jurisdiction's possibility space. Before the ruling, multiple legal interpretations are structurally available (S_Ω high). After the precedent, the court has committed to one interpretation, reducing S_Ω. Dissenting opinions are Refuse events: they document that alternative interpretations were considered and found inadmissible (in the view of dissenting justices), without reducing the structural option space (which is still available for future courts to revisit).

Admissibility Entropy

Structural possibility counts what remains selectable. Admissibility counts what remains justifiable.

Definition 56.9 (Admissibility Entropy)**.** The *admissibility entropy* of H is

$$S_\mathcal{A}(H) = \log |\mathcal{A}(H)|.$$

Theorem 56.10 (Refuse Entropy Reduction)**.** *If* $H' = H \mathbin{+\!\!+} [\mathsf{Refuse}(x, r)]$ *and* $x \in \mathcal{A}(H)$ *with* $|\mathcal{A}(H)| > 1$*, then*

$$S_\mathcal{A}(H') < S_\mathcal{A}(H).$$

Proof. The refusal of x removes x from the admissibility set: $\mathcal{A}(H') = \mathcal{A}(H) \setminus \{x\}$, so $|\mathcal{A}(H')| = |\mathcal{A}(H)| - 1$. Since the logarithm is strictly increasing, $\log |\mathcal{A}(H')| < \log |\mathcal{A}(H)|$. □

Corollary 56.11 (Pop and Refuse Act on Different Entropies)**.** *Pop strictly reduces S_Ω but need not reduce $S_\mathcal{A}$. Refuse strictly reduces $S_\mathcal{A}$ but does not reduce S_Ω.*

Proof. Pop removes a symbol from Ω by definition. Refuse removes a symbol from $\mathcal{A}(H)$ by definition but leaves Ω unchanged. □

Remark 56.12 (The Entropy Dual)*.* The two entropies form a dual pair:

$$S_\Omega(H) \geq S_\mathcal{A}(H) \quad \text{for all } H,$$

because $\mathcal{A}(H) \subseteq \Omega(H)$ always holds (admissible futures are a subset of structurally available futures). The gap $S_\Omega(H) - S_\mathcal{A}(H) = \log |\Omega(H)/\mathcal{A}(H)|$ measures the admissibility burden: the proportion of structurally available futures that are inadmissible given the current refusal history.

Possibility Debt

A computation that accumulates unresolved possibilities incurs a burden: every undecided symbol must eventually be committed or refused, and the cost of that decision is deferred but not eliminated.

Definition 56.13 (Decision Cost Function)**.** Let $C_s(x, H)$ denote the future cost of resolving whether symbol $x \in \Omega(H)$ should be committed, refused, bound, or collapsed.

Definition 56.14 (Possibility Debt)**.** The *possibility debt* of history H is

$$D_\Omega(H) = \sum_{x \in \Omega(H)} C_s(x, H).$$

Proposition 56.15 (Pop Reduces Local Possibility Debt)**.** *If $H' = H +\!+ [\mathsf{Pop}(x)]$, then*

$$D_\Omega(H') = D_\Omega(H) - C_s(x, H) + \Delta,$$

where $\Delta = \sum_{y \in \Omega(H')} (C_s(y, H') - C_s(y, H))$ is the change in decision cost for the remaining options.

Proof. Before Pop, the debt sum includes $C_s(x, H)$ and all remaining terms. After Pop, x is removed from Ω. Let Δ be as defined. Then $D_\Omega(H') = D_\Omega(H) -$

$C_s(x, H) + \Delta$. □

Remark 56.16. A Pop operation is *locally simplifying* when $\Delta < C_s(x, H)$: the remaining options become cheaper after the commitment. It is *globally complicating* when $\Delta > C_s(x, H)$: committing to x increases the downstream complexity of the remaining decisions. This models the phenomenon of premature commitment: early choices can constrain later choices in costly ways, even when they simplify the immediate decision.

Example 56.17 (Technical Debt in Software)**.** Technical debt is possibility debt in software development. A codebase with many unresolved design decisions (high D_Ω) imposes costs on every future change: the developer must navigate around or resolve these decisions. Refactoring is a sequence of Pop and Refuse events that resolves deferred decisions, reducing D_Ω at the cost of immediate effort. The observation that technical debt compounds corresponds to $\Delta > C_s(x, H)$ in many software contexts.

Reversible and Irreversible Computation

Definition 56.18 (Reversible Computation)**.** A computation is *reversible* if every operation has an inverse that restores the prior history. In Spherepop, an operation e is reversible if there exists an event e^{-1} such that $H +\!\!+ [e] +\!\!+ [e^{-1}] \sim_c H$ for all collapse rules c.

Theorem 56.19 (Pop Is Irreversible)**.** *No event e^{-1} can reverse a Pop operation* $\mathsf{Pop}(x)$ *while preserving history monotonicity.*

Proof. By the Conservation Law, $|\Omega(H +\!\!+ [\mathsf{Pop}(x)])| = |\Omega(H)| - 1$. Any subsequent event can only decrease or maintain $|\Omega|$; no event increases it (since Ω only shrinks). Therefore the option space after $\mathsf{Pop}(x)$ can never recover to $|\Omega(H)|$, and the Pop is irreversible. □

Remark 56.20 (Connection to Thermodynamics). The irreversibility of Pop corresponds to the thermodynamic irreversibility of measurement and commitment. In reversible computing (Bennett's erasure principle), reversible operations have zero thermodynamic cost; irreversible operations (like Pop) dissipate energy equal to $k_B T \ln 2$ per bit erased. The Spherepop Conservation Law is the computational analogue of the thermodynamic entropy production law: commitment converts possibility into history at a fixed rate, with no way to convert history back into possibility without external work.

Chapter 57

Collapse Curvature and Fiber Geometry

Collapse Curvature

Definition 57.1 (Collapse Fiber)**.** For collapse rule $c : \mathcal{H} \to O_c$ and observable state $o \in O_c$, the *collapse fiber* over o is $F_o = c^{-1}(o)$.

Definition 57.2 (Collapse Curvature)**.** The *collapse curvature* of history H under rule c is

$$\kappa_c(H) = \log |F_{c(H)}| = \log |c^{-1}(c(H))|.$$

Theorem 57.3 (Identity Collapse Has Zero Curvature)**.** *For the identity rule, $\kappa_{c_I}(H) = 0$ for all H.*

Proof. Under c_I, the fiber over H is $\{H\}$, so $|F_H| = 1$ and $\kappa_{c_I}(H) = \log 1 = 0$. □

Theorem 57.4 (Coarser Collapse Increases Curvature)**.** *If c_2 is coarser than c_1 (i.e. $H \sim_{c_1} H' \Rightarrow H \sim_{c_2} H'$), then $\kappa_{c_2}(H) \geq \kappa_{c_1}(H)$ for all H.*

Proof. Coarsening means every history identified with H by c_1 is also identified by c_2, so $F^{c_1}_{c_1(H)} \subseteq F^{c_2}_{c_2(H)}$. Therefore $|F^{c_1}_{c_1(H)}| \leq |F^{c_2}_{c_2(H)}|$, and since log is monotone, $\kappa_{c_1}(H) \leq \kappa_{c_2}(H)$. □

Corollary 57.5 (Maximum Collapse Theorem)**.** *State-only representations maximize collapse curvature among all collapse rules with finite observable space. That is, the LastWrite rule c_{LW} achieves the maximum curvature over the canonical rules.*

Proof. c_{LW} identifies histories that differ only in intermediate states, retaining only the final committed value per symbol. This is coarser than the Identity rule (which retains all history) and coarser than the Accumulate rule (which retains counts). By Coarser Collapse Increases Curvature, c_{LW} has higher curvature than these alternatives. □

Remark 57.6. The collapse curvature $\kappa_c(H) = \log |c^{-1}(c(H))|$ is precisely the observational entropy $S_c(o)$ from Part IX, now interpreted geometrically as the

curvature of the quotient space. A high-curvature collapse rule flattens history space into a few large equivalence classes; a low-curvature rule preserves most historical distinctions. The connection to CLIO is direct: CLIO's representational entropy is the collapse curvature of the projection functor.

Fiber Structure of Histories

Definition 57.7 (History Fiber)**.** Given a state projection $\sigma : \mathcal{H} \to S$, the *fiber* over state $s \in S$ is $F_s = \sigma^{-1}(s) = \{H \in \mathcal{H} \mid \sigma(H) = s\}$.

Theorem 57.8 (Average State Degeneracy Growth)**.** *Let S be a finite state space with $|S|$ elements and let $\sigma_n : \mathcal{H}_n \to S$ be any state projection from length-n histories. Then the average fiber size is*

$$\bar{D}_n = \frac{|E|^n}{|S|},$$

which grows exponentially in n when $|E| > 1$.

Proof. The number of length-n histories over alphabet E is $|\mathcal{H}_n| = |E|^n$. The projection σ_n partitions $\mathcal{H}_n$ into $|S|$ fibers. The average fiber size is $\bar{D}_n = |E|^n/|S|$. Since $|S|$ is constant and $|E| > 1$, this grows exponentially in n. □

Corollary 57.9 (Long Computations Are More Collapsed)**.** *For fixed finite observable state space S, longer histories are increasingly underdetermined by their final state. The fraction of historical information retained by the final state decreases as $|S|/|E|^n \to 0$.*

Remark 57.10. This is the formal content of the state illusion: as a computation proceeds, each additional step multiplies the number of distinct histories consistent with the current observable state. A debugger examining a running program faces exponentially many possible execution paths that produced the current state.

Fiber Dimension and Geometry

Definition 57.11 (Fiber Dimension)**.** The *fiber dimension* of observation o under rule c is $\dim(F_o) = \log_2 |c^{-1}(o)|$—the number of binary digits needed to distinguish histories within the fiber.

Definition 57.12 (Projection Distortion)**.** The *projection distortion* of collapse rule c at history H is

$$\delta_c(H) = \kappa_c(H) = \log |c^{-1}(c(H))|.$$

Theorem 57.13 (Projection Distortion Theorem)**.** *Larger fibers imply larger projection distortion, and distortion is bounded below by zero* (*Identity rule*) *and above by* $\log |\mathcal{H}_n|$ (*trivial rule collapsing all histories to one point*).

Proof. $\delta_c(H) = \kappa_c(H) \geq 0$ with equality iff the fiber is a singleton (injective collapse). The upper bound $\log |\mathcal{H}_n|$ is achieved by the trivial rule $c_\top$ that maps all histories to a single observable, producing one fiber of size $|E|^n$. □

Chapter 58

The Prefix Topology and History Metrics

Prefix Topology on Histories

Histories can be topologized by their common prefix structure, giving a natural notion of "closeness" for computational trajectories.

Definition 58.1 (Prefix Cylinder)**.** For a finite history P, define the *prefix cylinder*

$$U_P = \{H \in \mathcal{H} \mid P \text{ is a prefix of } H\}.$$

Theorem 58.2 (Prefix Cylinders Form a Basis)**.** *The collection $\mathcal{B} = \{U_P \mid P \in \mathcal{H}\}$ forms a basis for a topology on $\mathcal{H}$.*

Proof. Every history H belongs to U_H, so $\mathcal{B}$ covers $\mathcal{H}$.

For the intersection property: suppose $H \in U_P \cap U_Q$. Then both P and Q are prefixes of H. For finite sequences, if two sequences are both prefixes of H, then one is a prefix of the other (they form a total order under the prefix relation). Assume P is a prefix of Q. Then $U_Q \subseteq U_P$, so $H \in U_Q \subseteq U_P \cap U_Q$. For every point in the intersection, the basis element U_Q contains that point and is contained in the intersection. Therefore $\mathcal{B}$ is a basis. □

Remark 58.3. In the prefix topology, two histories are topologically close when they share a long initial segment. Computations with common early development occupy the same local region of history space, even if they later diverge. This captures the intuition that two program executions that run the same initialization code are "nearby" even if their subsequent paths differ.

Proposition 58.4 (Continuation Neighborhoods)**.** *For history H, the set U_H is the basic open neighborhood of H in the prefix topology. Every continuation of H lies in U_H. Diverging computations that share prefix P but not H lie in $U_P \setminus U_H$.*

Proof. A continuation of H is any $H' = H +\!\!+ H''$ for some non-empty H''. Such H' has H as a prefix, so $H' \in U_H$. A computation in $U_P \setminus U_H$ shares prefix P but

diverges after P, before reaching the state corresponding to H. □

The Prefix Ultrametric

Definition 58.5 (Longest Common Prefix Length)**.** For histories H_1, H_2, let $\mathsf{lcp}(H_1, H_2)$ denote the length of their longest common prefix.

Definition 58.6 (Prefix Distance)**.** Define

$$d_p(H_1, H_2) = 2^{-\mathsf{lcp}(H_1, H_2)}.$$

For identical finite histories, $d_p(H, H) = 0$.

Theorem 58.7 (Prefix Distance Is an Ultrametric)**.** *The function d_p satisfies the ultrametric inequality:*

$$d_p(H_1, H_3) \leq \max\{d_p(H_1, H_2),\ d_p(H_2, H_3)\}.$$

Proof. Let $a = \mathsf{lcp}(H_1, H_2)$ and $b = \mathsf{lcp}(H_2, H_3)$. Then H_1 and H_3 must share at least the first $\min(a, b)$ events, since both agree with H_2 on that prefix. Hence $\mathsf{lcp}(H_1, H_3) \geq \min(a, b)$. Therefore $d_p(H_1, H_3) = 2^{-\mathsf{lcp}(H_1, H_3)} \leq 2^{-\min(a,b)} = \max\{2^{-a}, 2^{-b}\} = \max\{d_p(H_1, H_2), d_p(H_2, H_3)\}$. □

Remark 58.8. The ultrametric inequality is stronger than the ordinary triangle inequality: $d(x, z) \leq \max\{d(x, y), d(y, z)\}$ implies $d(x, z) \leq d(x, y) + d(y, z)$ but not vice versa. Ultrametrics arise naturally in p-adic number theory and in the study of hierarchical structures, both of which are relevant to computation: program execution traces form a tree whose natural metric is ultrametric.

Geodesics in History Space

Definition 58.9 (History Geodesic)**.** A *geodesic* between histories H_0 and H_n in the prefix metric is a sequence $H_0, H_1, \ldots, H_n$ where each H_{i+1} is obtained from H_i by appending exactly one event, minimizing the total path length under the action functional $\mathcal{S}$.

Theorem 58.10 (Geodesic Repair Theorem)**.** *The shortest repair path from a failed history H to a target observable state o is the history geodesic from H to the nearest element of $c^{-1}(o)$ in the prefix metric.*

Proof. A repair morphism $\rho : H \to H'$ with $c(H') = o$ requires constructing a new history H' that maps to o under c. The repair cost is $d_{\text{edit}}(H, H')$—the mini-

mum number of event substitutions needed. In the prefix metric, the nearest history in $c^{-1}(o)$ is the one with the longest common prefix with H, minimizing the number of events that must differ. This is precisely the minimum-edit-distance repair. $\square$

Chapter 59

The Full Semantics of Bind

Motivation: Bind Has Syntax but No Ontology

The critique identified what is perhaps the most important gap in the theory. $\mathsf{Bind}(a, b)$ appears throughout the monograph as a dependency declaration, but its operational semantics—what it actually *does* to computation—has not been fully specified.

Three interpretations are possible, and distinguishing them clarifies the design:

1. *Bind as annotation*: $\mathsf{Bind}(a, b)$ is a pure record in the history, with no immediate operational effect. Future agents reading the history can see the dependency, but the runtime does not enforce it.
2. *Bind as constraint*: $\mathsf{Bind}(a, b)$ establishes that if a is refused, b becomes inadmissible. The runtime enforces dependency propagation.
3. *Bind as synchronization*: $\mathsf{Bind}(a, b)$ requires that a and b be committed simultaneously in parallel contexts—a synchronization barrier.

All three are coherent; they correspond to different use cases. This chapter develops interpretation (2) as the primary operational semantics, with the others as derived modes.

Dependency Graphs

Definition 59.1 (Dependency Graph)**.** The *dependency graph* of history H is the directed graph $G_H = (V, E)$ where $V = \Omega_0$ and $(a, b) \in E$ iff $\mathsf{Bind}(a, b) \in H$.

Definition 59.2 (Dependency Closure)**.** The *dependency closure* of a set $X \subseteq \Omega_0$ under G_H is

$$\overline{X}^{G_H} = \{y \in \Omega_0 \mid \text{there exists a directed path from some } x \in X \text{ to } y \text{ in } G_H\}.$$

Refusal Propagation

Definition 59.3 (Propagation Rule)**.** Given $\mathsf{Bind}(a, b) \in H$ and a refusal $\mathsf{Refuse}(a, r) \in H$, the *propagated refusal* for b is $\mathsf{Refuse}(b, \rho(r))$ where $\rho : \mathsf{RefusalReason} \to \mathsf{RefusalReason}$ is a reason transformation (e.g. $\rho(r) = \mathsf{DependencyFailed}(a, r)$).

Definition 59.4 (Refusal Closure)**.** The *refusal closure* of history H under dependency propagation is the smallest extension $H^* \supseteq H$ such that for every $\mathsf{Bind}(a, b) \in H$ and $\mathsf{Refuse}(a, r) \in H^*$, we have $\mathsf{Refuse}(b, \rho(r)) \in H^*$.

Theorem 59.5 (Dependency Propagation Theorem)**.** *Refusals propagate along dependency edges: if* $\mathsf{Bind}(a, b) \in H$ *and* $\mathsf{Refuse}(a, r) \in H$*, then* $b \notin \mathcal{A}(H^*)$*.*

Proof. By definition of refusal closure, $\mathsf{Refuse}(b, \rho(r)) \in H^*$. By definition of $\mathcal{A}$, $b \notin \mathcal{A}(H^*)$. □

Theorem 59.6 (Dependency Closure Theorem)**.** *In an acyclic dependency graph* G_H*, every refusal generates a unique maximal refusal closure.*

Proof. Acyclicity ensures that dependency propagation terminates: since every path in G_H has finite length (no cycles), the propagation reaches every dependent node exactly once. Uniqueness follows from the determinism of ρ: each refusal generates exactly one propagated reason at each dependent node. Maximality holds because we take the closure under all dependency edges. □

Corollary 59.7 (Deadlock Detection via Dependency Closure)**.** *A dependency cycle* $a_1 \to a_2 \to \cdots \to a_k \to a_1$ *in* G_H *with any* a_i *refused implies that the refusal propagates around the cycle indefinitely, making all* a_i *inadmissible.*

Proof. Starting from $\mathsf{Refuse}(a_i, r)$, propagation reaches $a_{i+1}, \ldots, a_k, a_1, a_2, \ldots$ continuing around the cycle. All elements of the cycle become inadmissible under the refusal closure. □

Bind Symmetry and Asymmetry

Proposition 59.8 (Bind Is Not Symmetric)**.** *In general,* $\mathsf{Bind}(a, b)$ *and* $\mathsf{Bind}(b, a)$ *have different operational effects: the former propagates refusals from* a *to* b*, the latter from* b *to* a*.*

Proof. Under the propagation rule, $\mathsf{Bind}(a, b)$ causes $\mathsf{Refuse}(a, r) \Rightarrow \mathsf{Refuse}(b, \rho(r))$, while $\mathsf{Bind}(b, a)$ causes $\mathsf{Refuse}(b, r) \Rightarrow \mathsf{Refuse}(a, \rho(r))$. These are distinct effects unless $\mathcal{A}$ treats a and b symmetrically. □

Definition 59.9 (Symmetric Bind)**.** $\mathsf{Bind}(a, b)$ is *symmetric* if both $\mathsf{Bind}(a, b)$ and $\mathsf{Bind}(b, a)$ are in the history, creating bidirectional dependency.

Example 59.10. A symmetric bind models mutual dependency: two modules that each require the other to be non-refused. An asymmetric bind models unidirectional dependency: a database connection that fails if the database server is refused, but the server is not affected by failures in the client connection.

Bind as Coordination: the Multi-Agent Case

Theorem 59.11 (Bind Creates Coupled Trajectory Spaces)**.** *Let agents A_1 and A_2 have independent continuation spaces $\mathcal{C}_1$ and $\mathcal{C}_2$. A bind $\mathsf{Bind}(a, b)$ with $a \in \Omega_1$ and $b \in \Omega_2$ creates a coupled trajectory space $\mathcal{C}_{12}$ where*

$$|\mathcal{C}_{12}| > |\mathcal{C}_1| + |\mathcal{C}_2|$$

whenever new joint continuations are enabled by the coupling.

Proof. Without the bind, agents evolve independently. A joint continuation requiring both a to be committed and b to be committed at related times is only reachable when the agents are coupled. The coupling via Bind enables this joint continuation, adding to the total reachability beyond the independent sum. □

Chapter 60

Reconstruction Geometry and Cost Models

Defining Reconstruction Cost Formally

The reconstruction cost $C_r(c)$ has been used throughout the monograph as a primitive. This chapter gives it a formal definition.

Definition 60.1 (Reconstruction Problem)**.** Given an observable state $o \in O_c$ and collapse rule c, the *reconstruction problem* is: find the history $H \in c^{-1}(o)$ that produced o.

Definition 60.2 (Reconstruction Cost)**.** The reconstruction cost $C_r(c, o)$ is the minimum number of oracle queries to $c^{-1}(o)$ (each query reveals one event of the unknown history) needed to identify H with probability at least $1 - \varepsilon$, for fixed error tolerance $\varepsilon > 0$.

Theorem 60.3 (Information-Theoretic Lower Bound)**.**

$$C_r(c, o) \geq \frac{S_c(o)}{\log |\Sigma|}$$

where Σ is the event alphabet and $S_c(o) = \log |c^{-1}(o)|$ is the observational entropy.

Proof. The fiber $c^{-1}(o)$ contains $|c^{-1}(o)| = 2^{S_c(o)}$ candidate histories. Each oracle query reveals one event (chosen from Σ), providing $\log |\Sigma|$ bits of information. To distinguish $2^{S_c(o)}$ candidates requires at least $S_c(o)/\log |\Sigma|$ queries. □

Theorem 60.4 (Note Leverage as Entropy Reduction Rate)**.** *The reachability leverage of a note N satisfies*

$$\Lambda(N) = \frac{k}{C_{\text{create}}(N)} \cdot (S_c(o) - S_c(o \mid N)),$$

where $k = 1/\log |\Sigma|$ is the information density constant and $S_c(o \mid N)$ is the entropy of observation o when N is available.

Proof. Note N reduces the effective fiber size from $|c^{-1}(o)|$ to $|c^{-1}(o) \cap \mathsf{Compatible}(N)|$. The reconstruction cost reduction is $\Delta C_r = C_r(c, o) - C_r(c, o \mid N) \geq k(S_c(o) - S_c(o \mid N))$. Dividing by $C_{\text{create}}(N)$ gives $\Lambda(N) \geq k(S_c(o) - S_c(o \mid N))/C_{\text{create}}(N)$. □

Corollary 60.5 (Optimal Note Theorem)**.** *The optimal note N^* for minimizing reconstruction cost of observations under c is the note that maximizes $(S_c(o) - S_c(o \mid N))/C_{\text{create}}(N)$: maximum entropy reduction per unit creation cost.*

Reconstruction Distance

Definition 60.6 (Reconstruction Distance)**.** The *reconstruction distance* between two histories $H_1, H_2 \in c^{-1}(o)$ is the minimum number of events that must be queried to distinguish H_1 from H_2, given that both are in the same fiber.

Proposition 60.7 (Reconstruction Distance Equals Edit Distance in Fiber)**.** *The reconstruction distance between H_1 and H_2 in the same fiber $c^{-1}(o)$ equals the Hamming distance $d_H(H_1, H_2)$ when the fiber consists of equal-length histories.*

Proof. Two histories in the same fiber are observationally equivalent under c: they produce the same observable state. To distinguish them, an oracle must identify the positions where they differ. The minimum number of queries needed is the Hamming distance—the number of positions where they actually differ. □

The Reachability Leverage Spectrum

Different note classes achieve different positions on the leverage spectrum.

Note Class	Entropy Reduction	Creation Cost	Typical Λ
Grocery list	Low	Very low	≈ 100
Commit message	Medium	Low	$\approx 10{,}000$
One-line equation	Very high	Low	$\approx 10^6$
Mathematical proof	Extremely high	Medium	$\approx 10^9$
Programming language	Vast	High	$\approx 10^{12}$
Scientific theory	Unbounded	Very high	$\to \infty$

The divergence of Λ for scientific theories reflects the fact that a theory like Newtonian mechanics reduces the reconstruction cost of infinitely many observations (planetary positions, projectile trajectories, tidal cycles) to near zero for any agent fluent in the theory.

Chapter 61

Refusal Logic and the Algebra of Inadmissibility

Refusal Sets as a Monotone Structure

Definition 61.1 (Refusal Set)**.** For a history H, define the refusal set $\mathcal{R}(H) = \{(x, r) \mid \mathsf{Refuse}(x, r) \in H\}$.
Definition 61.2 (Refusal Order)**.** Define $H_1 \preceq_R H_2$ iff $\mathcal{R}(H_1) \subseteq \mathcal{R}(H_2)$.
Theorem 61.3 (Refusal Monotonicity)**.** *If $H' = H +\!\!+ [e]$ for any event e, then $H \preceq_R H'$.*

Proof. If $e = \mathsf{Refuse}(x, r)$, then $\mathcal{R}(H') = \mathcal{R}(H) \cup \{(x, r)\} \supseteq \mathcal{R}(H)$. If e is any other event, $\mathcal{R}(H') = \mathcal{R}(H)$. In both cases $\mathcal{R}(H) \subseteq \mathcal{R}(H')$, so $H \preceq_R H'$. □

Corollary 61.4 (No Silent Unrefusal)**.** *No legal Spherepop event removes a refusal from the history. Once $(x, r) \in \mathcal{R}(H)$, we have $(x, r) \in \mathcal{R}(H')$ for all legal extensions H' of H.*

Refusal Algebra

Definition 61.5 (Refusal Conjunction)**.** Histories H and H' with refusal sets $\mathcal{R}(H)$ and $\mathcal{R}(H')$ have *refusal conjunction* $\mathcal{R}(H) \wedge \mathcal{R}(H') = \mathcal{R}(H) \cup \mathcal{R}(H')$—the set of symbols refused in either history.
Definition 61.6 (Refusal Disjunction)**.** The *refusal disjunction* is $\mathcal{R}(H) \vee \mathcal{R}(H') = \mathcal{R}(H) \cap \mathcal{R}(H')$—the set of symbols refused in both histories.
Proposition 61.7 (Refusal Sets Form a Lattice)**.** *Under conjunction ($\cup$) and disjunction ($\cap$), the refusal sets of all histories over Ω_0 form a distributive lattice with bottom $\varnothing$ (no refusals) and top $\Omega_0 \times \mathsf{Reasons}$ (everything refused).*

Proof. $(\wp(\Omega_0 \times \mathsf{Reasons}), \cup, \cap)$ is the powerset lattice, which is always distributive. □

Definition 61.8 (Refusal Implication)**.** $r_1 \Rightarrow_R r_2$ (refusal r_1 implies refusal r_2) if every history containing r_1 necessarily contains r_2 due to dependency propagation: $\mathsf{Bind}(x_1, x_2) \in H$ and $r_1 = (x_1, r)$ imply $r_2 = (x_2, \rho(r))$.

Theorem 61.9 (Refusal Logic is Monotone)**.** *The refusal implication relation $\Rightarrow_R$ is monotone: if $H \preceq_R H'$ and $\mathcal{R}(H) \vdash_R (x, r)$ (i.e. the refusal closure of $\mathcal{R}(H)$ contains (x, r)), then $\mathcal{R}(H') \vdash_R (x, r)$.*

Proof. Since $\mathcal{R}(H) \subseteq \mathcal{R}(H')$ and the refusal closure is monotone in the refusal set (adding more refusals only adds more propagated refusals), the conclusion (x, r) follows from $\mathcal{R}(H')$ whenever it follows from $\mathcal{R}(H)$. □

Repair as Certified Transition

Repair is not the erasure of refusal. It is the construction of a new admissible path that acknowledges and resolves the refusal.

Definition 61.10 (Repair Operator)**.** A *repair operator* is a partial map

$$\mathsf{Repair}_\rho : \mathsf{Refused}(T, r) \to \mathsf{Admissible}(T)$$

defined only when ρ is a proof that the condition producing refusal reason r has been resolved.

Theorem 61.11 (Repair Soundness)**.** *If $\Gamma \vdash t : \mathsf{Refused}(T, r)$ and ρ is a valid repair certificate for r, then $\Gamma \vdash \mathsf{Repair}_\rho(t) : \mathsf{Admissible}(T)$.*

Proof. The typing rule for repair is:

$$\frac{\Gamma \vdash t : \mathsf{Refused}(T, r) \quad \Gamma \vdash \rho : \mathsf{Resolved}(r)}{\Gamma \vdash \mathsf{Repair}_\rho(t) : \mathsf{Admissible}(T)} \quad \text{[T-REPAIR]}$$

The proof $\rho : \mathsf{Resolved}(r)$ certifies that the condition causing refusal r has been remedied. Given this certificate, the repaired term satisfies the admissibility requirement for its type. □

Theorem 61.12 (Repair Does Not Violate Refusal Monotonicity)**.** *If a refused term is repaired, the original refusal remains in the history.*

Proof. Repair appends a new event $\mathsf{Repair}(x, \rho)$ to the history. It does not remove the earlier $\mathsf{Refuse}(x, r)$ event. Thus $\mathsf{Refuse}(x, r) \in H$ implies $\mathsf{Refuse}(x, r) \in H \mathbin{+\!\!+} [\mathsf{Repair}(x, \rho)]$. □

Remark 61.13. This is the difference between repair and denial. Repair preserves the evidence that something failed while adding the evidence that it was fixed. Denial removes or suppresses the failure record. Spherepop permits repair but forbids denial as a history operation—the history is immutable, and past refusals cannot be erased.

Chapter 62

Partial-Order Histories and True Concurrency

The Limitation of Total-Order Histories

The histories developed throughout this monograph are sequences: total orders on events. This is adequate for sequential computation but misrepresents true concurrency. In a distributed system where events at different nodes occur simultaneously, there is no canonical total order on them—only a partial order of causal precedence.

Definition 62.1 (Partial-Order History)**.** A *partial-order history* (or *event structure*) is a pair $\mathcal{E} = (E, \prec)$ where E is a finite set of events and $\prec$ is a strict partial order (irreflexive, asymmetric, transitive) representing causal precedence: $e_1 \prec e_2$ means e_1 causally precedes e_2.

Definition 62.2 (Concurrent Events)**.** Events e_1 and e_2 are *concurrent* (written $e_1 \parallel e_2$) if neither $e_1 \prec e_2$ nor $e_2 \prec e_1$.

Linearizations and the Linearization Theorem

Definition 62.3 (Linearization)**.** A *linearization* of $\mathcal{E} = (E, \prec)$ is a total order H on E that extends $\prec$: if $e_1 \prec e_2$ then e_1 appears before e_2 in H.

Theorem 62.4 (Linearization Theorem)**.** *Every partial-order history $\mathcal{E} = (E, \prec)$ has at least one linearization. If $\mathcal{E}$ has k pairs of concurrent events, then the number of linearizations is at least 2^k.*

Proof. Existence follows from Szpilrajn's theorem: every partial order can be extended to a total order. For the count: each pair of concurrent events $e_i \parallel e_j$ can be ordered as e_i before e_j or e_j before e_i, independently. With k independent concurrent pairs, there are at least 2^k distinct linearizations. □

Remark 62.5. This is the formal foundation of the interleaving explosion noted in Part XV. The number of linearizations of a partial-order history with k con-

current pairs is exponential in k—which is why concurrent systems are hard to test by examining linearizations one at a time.

Partial-Order Collapse Rules

Definition 62.6 (Partial-Order Collapse Rule)**.** A *partial-order collapse rule* $c : \mathsf{POHist} \to O_c$ maps partial-order histories to observable states. The rule is *linearization-invariant* if $c(\mathcal{E}) = c(H)$ for every linearization H of $\mathcal{E}$.

Theorem 62.7 (Collapse Invariance Theorem)**.** *The Accumulate rule c_{acc} is linearization-invariant: the count of Pop events for each symbol is the same in every linearization of any partial-order history.*

Proof. The Accumulate rule counts events by type, ignoring their order. Since every linearization of $\mathcal{E}$ contains exactly the same events (just in different orders), the count for each symbol is identical across all linearizations. □

Theorem 62.8 (LastWrite Is Not Linearization-Invariant)**.** *There exist partial-order histories $\mathcal{E}$ and linearizations H_1, H_2 such that $c_{LW}(H_1) \neq c_{LW}(H_2)$.*

Proof. Let $e_1 = \mathsf{Pop}(x)$ from process P_1 and $e_2 = \mathsf{Pop}(x)$ from process P_2, with $e_1 \parallel e_2$ (concurrent). In linearization $H_1 = [e_1, e_2]$, c_{LW} observes the last write to x as e_2. In linearization $H_2 = [e_2, e_1]$, the last write is e_1. Hence $c_{LW}(H_1) \neq c_{LW}(H_2)$. This is the formal content of the race condition: the LastWrite collapse is sensitive to linearization order. □

Corollary 62.9. *Only linearization-invariant collapse rules can be used in distributed Spherepop systems without requiring total ordering of events. Systems requiring LastWrite semantics (such as most database systems) must impose a total order (via timestamps, Lamport clocks, or consensus protocols).*

Lamport Clocks as Spherepop Events

Definition 62.10 (Lamport Clock Event)**.** A *Lamport clock event* is a Pop event of the form $\mathsf{Pop}(\mathsf{ts}(e, t))$ where e is a computational event and t is a logical timestamp satisfying: $t > \max\{t' \mid \mathsf{Pop}(\mathsf{ts}(e', t')) \in H \text{ and } e' \prec e\}$.

Theorem 62.11 (Lamport Clocks Produce Total-Order History)**.** *Adding Lamport clock events to a partial-order history $\mathcal{E}$ produces a total-order history H_{LC} consistent with the causal ordering: $e_1 \prec e_2 \Rightarrow \mathsf{ts}(e_1) < \mathsf{ts}(e_2)$ in H_{LC}.*

Proof. By the Lamport clock construction, a Lamport timestamp is always greater than the timestamps of causally preceding events. The resulting total order on timestamps induces a total order on events that extends the causal partial order. □

Part XXIII

Operator Theory and Algebraic Structure

Chapter 63

The Operator Representation Theorem

Primitive Transformations of Possibility Space

The four Spherepop operators were derived in Part II from engineering requirements. This chapter provides the deeper derivation: the operators emerge as structural necessities from the requirement that possibility space be managed in a way that preserves historical accountability.

Definition 63.1 (Possibility Space Transformation)**.** A *possibility space transformation* is a function $f : (\mathcal{H} \times \wp(\Omega_0)) \to (\mathcal{H} \times \wp(\Omega_0))$ that maps world states to world states.

Definition 63.2 (Primitive Transformation Types)**.** The four primitive transformation types are:

1. *Selection* (Sel): removes one element from Ω and records its removal.
2. *Exclusion* (Exc): records that an element is inadmissible without removing it from Ω.
3. *Coupling* (Cpl): records a dependency between two elements without consuming either.
4. *Projection* (Proj): applies a function to the history, producing an observable.

Theorem 63.3 (Operator Representation Theorem)**.** *Every legal transformation of a Spherepop world state (H, Ω) that (a) appends to history monotonically, (b) does not increase $|\Omega|$, (c) preserves the admissibility record, and (d) can produce a new observable decomposes as a composition of the four primitive transformations:*

$$f = \mathsf{Proj}^{n_4} \circ \mathsf{Cpl}^{n_3} \circ \mathsf{Exc}^{n_2} \circ \mathsf{Sel}^{n_1}$$

for some $n_1, n_2, n_3, n_4 \geq 0$, which correspond to sequences of Pop, Refuse, Bind, *and* Collapse *operations.*

Proof. We show each primitive transformation is realized by exactly one operator.

Selection (Pop): The only transformation that decreases $|\Omega|$ is P_x for some $x \in \Omega$. By requirement (b), no transformation increases $|\Omega|$, and by requirement (a), every transformation appends to history. A transformation that decreases $|\Omega|$ by exactly one and appends one event is precisely Pop.

Exclusion (Refuse): A transformation that leaves $|\Omega|$ unchanged but modifies $\mathcal{A}(H)$ must append a Refuse event (the only event that changes admissibility without changing Ω). By requirement (c), the reason must be recorded. This is precisely $\mathsf{Refuse}(x, r)$.

Coupling (Bind): A transformation that leaves $|\Omega|$ and $\mathcal{A}(H)$ unchanged but appends a dependency record must be a Bind event. No other event type records binary relations between symbols.

Projection (Collapse): A transformation that produces an observable output by applying a function to the history is precisely $\mathsf{Collapse}(x, c)$. Requirements (a)–(c) do not force this transformation; requirement (d) does.

Any compound transformation decomposes into a sequence of these four types by structural induction on the sequence of events appended to the history. □

Remark 63.4. The Operator Representation Theorem elevates the four operators from language design choices to structural necessities. Any system that (a) maintains a monotonically growing history, (b) manages a shrinking option space, (c) preserves admissibility records, and (d) can produce observations must contain operations isomorphic to Pop, Refuse, Bind, and Collapse.

Algebraic Properties of the Operators

Proposition 63.5 (Refuse Idempotence)**.** *Applying* $\mathsf{Refuse}(x, r)$ *twice has the same effect as applying it once:*

$$\mathcal{A}(H +\!\!+ [\mathsf{Refuse}(x, r)]) = \mathcal{A}(H +\!\!+ [\mathsf{Refuse}(x, r)] +\!\!+ [\mathsf{Refuse}(x, r)]).$$

Proof. Both histories have x removed from the admissibility set after the first Refuse. The second Refuse has no additional effect since x is already inadmissible. □

Proposition 63.6 (Pop-Refuse Non-Commutativity)**.** *In general,* $\mathsf{Pop}(x)$ *followed*

by $\mathsf{Refuse}(x, r)$ *is not the same as* $\mathsf{Refuse}(x, r)$ *followed by* $\mathsf{Pop}(x)$.

Proof. After $\mathsf{Pop}(x)$: $x \notin \Omega$ and $x \in \mathcal{A}$ (Pop removes from Ω). Then $\mathsf{Refuse}(x, r)$: $x \notin \mathcal{A}$ (inadmissible and not in Ω).

After $\mathsf{Refuse}(x, r)$: $x \in \Omega$ but $x \notin \mathcal{A}$ (Refuse removes from $\mathcal{A}$ only). Then $\mathsf{Pop}(x)$ would fail the admissibility check: $x \in \Omega$ but $x \notin \mathcal{A}$, so the Pop is rejected.

The two sequences have different outcomes. □

Theorem 63.7 (Collapse Non-Commutativity)**.** *For collapse rules* c_1 *and* c_2*, in general* $c_1(c_2(H)) \neq c_2(c_1(H))$ *when treating* c_1 *and* c_2 *as functions on the history.*

Proof. Let $c_1 = c_{LW}$ and $c_2 = c_{\text{acc}}$. For history $H = [\mathsf{Pop}(x), \mathsf{Pop}(x)]$: $c_{LW}(H) = \{x \mapsto 1\}$ (last write wins, x is present once); $c_{\text{acc}}(H) = \{x \mapsto 2\}$ (two pops counted). Applying the other rule to these outputs gives different results depending on the order, since the outputs have different structures. □

Theorem 63.8 (Bind Commutativity Under Symmetric Dependencies)**.** *If* $\mathsf{Bind}(a, b)$ *and* $\mathsf{Bind}(b, a)$ *are both in* H*, then the dependency between* a *and* b *is symmetric:* $a \notin \mathcal{A}(H^*) \Leftrightarrow b \notin \mathcal{A}(H^*)$ *under the refusal closure.*

Proof. By the Dependency Propagation Theorem, $\mathsf{Refuse}(a, r) \in H$ propagates to $\mathsf{Refuse}(b, \rho(r)) \in H^*$ via $\mathsf{Bind}(a, b)$, and $\mathsf{Refuse}(b, r') \in H$ propagates to $\mathsf{Refuse}(a, \rho(r')) \in H^*$ via $\mathsf{Bind}(b, a)$. Hence inadmissibility propagates in both directions, creating symmetric mutual inadmissibility. □

Chapter 64

Counterfactuals and Causal Reasoning

Continuations and Counterfactuals

Causal reasoning asks: what would have happened if we had done otherwise? In Spherepop, this is a question about alternative histories—histories that agree with the actual up to some point and then diverge.

Definition 64.1 (Counterfactual History)**.** A *counterfactual history* relative to H at step k is any history H' such that H' agrees with H on the first $k-1$ events and differs on event k: $H[1:k-1] = H'[1:k-1]$ but $H[k] \neq H'[k]$.

Definition 64.2 (Counterfactual Continuation Difference)**.** The *counterfactual difference* between histories H_1 and H_2 is

$$\mathcal{C}(H_1) \triangle \mathcal{C}(H_2) = (\mathcal{C}(H_1) \setminus \mathcal{C}(H_2)) \cup (\mathcal{C}(H_2) \setminus \mathcal{C}(H_1)),$$

the symmetric difference of their future continuation sets.

Theorem 64.3 (Counterfactual Distinguishability Theorem)**.** *If $\mathcal{C}(H_1) \neq \mathcal{C}(H_2)$, then H_1 and H_2 are computationally non-equivalent: there exists a future computation that is possible from one but not the other.*

Proof. $\mathcal{C}(H_1) \neq \mathcal{C}(H_2)$ implies $\mathcal{C}(H_1) \triangle \mathcal{C}(H_2) \neq \varnothing$. There exists c in the symmetric difference: $c \in \mathcal{C}(H_1) \setminus \mathcal{C}(H_2)$ (or vice versa). This means c is reachable from H_1 but not from H_2. The two histories therefore differ in their computational potential. □

Definition 64.4 (Causal Influence)**.** Event e_i in history H *causally influences* continuation c if removing e_i from H (and adjusting subsequent events consistently) changes whether c is reachable:

$$c \in \mathcal{C}(H) \text{ but } c \notin \mathcal{C}(H \setminus \{e_i\}).$$

Theorem 64.5 (Pop Events Are Causally Primary)**.** *Every continuation $c \in \mathcal{C}(H)$ is causally influenced by at least one Pop event in H.*

Proof. A continuation c consists of further events extending H. The specific form of c depends on what has been committed (the Pop events in H), since commitments remove alternatives and shape which future events are admissible. Removing all Pop events from H leaves only Refuse, Bind, and Collapse events, which preserve Ω and do not commit to specific paths. The set $\mathcal{C}(H_{\text{no}-\text{pop}})$ is a superset of $\mathcal{C}(H)$, so specific continuations reachable from H are reachable only because specific Pop events have narrowed the possibility space in the right direction. □

Causal Models and Spherepop

Definition 64.6 (Spherepop Causal Model)**.** A *Spherepop causal model* is a dependency graph G_H together with a labeling of each node x with its Pop or Refuse status in history H. An intervention on x is a counterfactual history H' that changes the event at position k (where x first appears) from $\mathsf{Pop}(x)$ to $\mathsf{Refuse}(x, \mathsf{Intervened})$ or vice versa.

Theorem 64.7 (Do-Calculus Correspondence)**.** *The Spherepop intervention operation corresponds to Pearl's do-operator: $\mathsf{do}(\mathsf{Pop}(x))$ sets x to committed status regardless of its upstream causes, which in Spherepop corresponds to adding $\mathsf{Pop}(x)$ to the history while severing the incoming dependency edges to x in G_H.*

Proof. Pearl's $\mathsf{do}(X = x)$ removes the incoming arrows to X in the causal graph and sets X to x. In Spherepop: severing incoming dependency edges corresponds to removing Bind events (y, x) from G_H; setting X to committed corresponds to adding $\mathsf{Pop}(x)$. The resulting modified history H' is the Spherepop representation of the interventional distribution $P(\cdot \mid \mathsf{do}(X = x))$. □

Chapter 65

Notes as a Category

Note Morphisms

Definition 65.1 (Note Morphism)**.** A *note morphism* from note N_1 to note N_2 is a function $f : \mathcal{C}(N_1) \to \mathcal{C}(N_2)$ that maps continuations preserved by N_1 to continuations preserved by N_2, such that $P_A(f(c), t \mid N_2) \geq P_A(c, t \mid N_1)$: the target note preserves the continuation at least as well as the source.

Theorem 65.2 (Notes Form a Category)**.** *Notes with note morphisms form a category* Notes*:*

- *Objects: notes N (artifacts increasing reachability of some continuation).*
- *Morphisms: note morphisms $f : N_1 \to N_2$.*
- *Composition: $(g \circ f)(c) = g(f(c))$, which is a valid note morphism since reachability is preserved by composition.*
- *Identity:* $\mathsf{id}_N(c) = c$ *is the identity note morphism.*

Proof. Composition: if $f : N_1 \to N_2$ and $g : N_2 \to N_3$ are note morphisms, then for any $c \in \mathcal{C}(N_1)$: $P_A(g(f(c)), t \mid N_3) \geq P_A(f(c), t \mid N_2) \geq P_A(c, t \mid N_1)$. So $g \circ f$ is a valid note morphism. Identities and associativity are immediate. □

Note Functors

Theorem 65.3 (The Continuation Functor)**.** *There is a functor $\mathcal{C} : \mathsf{Notes} \to \mathsf{Set}$ sending each note to its continuation set and each morphism to the corresponding function on continuation sets.*

Proof. $\mathcal{C}$ sends note N to $\mathcal{C}(N) \subseteq \mathcal{C}_{\text{total}}$ and morphism $f : N_1 \to N_2$ to $f : \mathcal{C}(N_1) \to \mathcal{C}(N_2)$. This preserves composition and identities by definition of note morphism. □

Theorem 65.4 (Note Composition Increases Reachability)**.** *The tensor product*

$N_1 \otimes N_2$ of two notes satisfies

$$\mathcal{C}(N_1 \otimes N_2) \supseteq \mathcal{C}(N_1) \cup \mathcal{C}(N_2)$$

with strict inclusion when the notes enable joint continuations not reachable from either alone.

Proof. A combined note system has access to the continuations of each component plus any joint continuations enabled by having both. Since $\mathcal{C}(N_1) \cup \mathcal{C}(N_2) \subseteq \mathcal{C}(N_1 \otimes N_2)$ by definition and joint continuations strictly add to the union, the inclusion is proper when $\mathcal{C}_{\text{bind}}(N_1, N_2) \neq \varnothing$. □

The Reachability Value of Knowledge

Definition 65.5 (Knowledge Value Functional)**.** The *knowledge value* of a note N for agent A at time t is

$$V_K(N, A, t) = \sum_{c \in \mathcal{C}} (P_A(c, t \mid N) - P_A(c, t)) = G_A(N, t),$$

the total reachability gain.

Theorem 65.6 (Knowledge Leverage Theorem)**.** *The knowledge value of a note equals the total increase in reachability it provides:*

$$V_K(N, A, t) = |\mathcal{C}(N)_{\text{new}}| \cdot \bar{g},$$

where $|\mathcal{C}(N)_{\text{new}}|$ is the number of new continuations enabled and $\bar{g}$ is the average gain per continuation.

Proof. Decompose the sum: $V_K(N, A, t) = \sum_{c \in \mathcal{C}(N)} G_A(N, c, t) + \sum_{c \notin \mathcal{C}(N)} G_A(N, c, t)$. For $c \notin \mathcal{C}(N)$, $G_A(N, c, t) = 0$ by definition of the continuation set. For $c \in \mathcal{C}(N)$, the average gain is $\bar{g}$. The result follows. □

Part XXIV

Civilization as Continuation Reservoir

Chapter 66

Libraries, Archives, and Scientific Infrastructure

Libraries as Continuation Reservoirs

Definition 66.1 (Continuation Reservoir)**.** A collection of notes $\mathcal{N}$ is a *continuation reservoir* for agent population A if it strictly increases total reachability:

$$\sum_{c \in \mathcal{C}} P_A(c, t \mid \mathcal{N}) > \sum_{c \in \mathcal{C}} P_A(c, t).$$

Theorem 66.2 (Library Reachability Theorem)**.** *If each note $N_i \in \mathcal{N}$ has non-negative reachability leverage and at least one note has positive leverage, then $\mathcal{N}$ is a continuation reservoir.*

Proof. Let $\Delta_i(c) = P_A(c, t \mid N_i) - P_A(c, t)$. Non-negative leverage means $\sum_c \Delta_i(c) \geq 0$ for all i. Positive leverage for some N_j means $\sum_c \Delta_j(c) > 0$. Summing over all notes: $\sum_{c,i} \Delta_i(c) > 0$, which implies the total reachability increases. □

Remark 66.3. Civilization is not primarily the accumulation of facts. It is the accumulation of preserved re-entry points into abandoned, deferred, repaired, and generative continuations. A library's value is not the mass of paper it contains but the volume of continuations it makes accessible.

Science as Civilizational Refusal

Theorem 66.4 (Scientific Progress as Directed Refusal)**.** *Scientific progress is a sequence of Refuse events applied to the space of possible theories: each failed experiment refuses a class of theories with reason* ExperimentalRefutation*.*

Proof. A scientific experiment tests a prediction P derived from theory T. If the experiment fails ($\neg P$ is observed), the falsification principle (Popper) requires adding $\mathsf{Refuse}(T, \mathsf{Refuted}(P))$ to the scientific community's history. This mono-

tonically shrinks the admissibility set of viable theories. Scientific progress is therefore the directed shrinkage of the theory-admissibility set, guided by experiment. □

Corollary 66.5 (Science Increases Admissibility Entropy Compression)**.** *As science progresses, the admissibility entropy of the theory space $S_{\mathcal{A}}(\text{theories})$ decreases: fewer theories are admissible given the accumulated experimental record.*

Remark 66.6. This formalizes the intuition that science constrains rather than merely accumulates. Each confirmed refutation reduces the set of viable theories, making the remaining theories more precisely constrained. The admissibility entropy of scientific theories in 2025 is dramatically lower than in 1600, not because we have proved more but because we have refused more.

Writing Systems as Compression Infrastructure

Theorem 66.7 (Writing Reduces Reconstruction Cost Across Time)**.** *A writing system reduces the reconstruction cost $C_r(c, o)$ for observations o made before its invention by making the associated histories accessible across the temporal gap between creation and reading.*

Proof. Without writing, an observation at time t_1 produces a fiber $c^{-1}(o)$ at time $t_2 > t_1$ of size exponential in $(t_2 - t_1)$ (all histories consistent with the biological memory trace are candidates). With writing, the fiber is reduced to histories consistent with the written record, which is exponentially smaller. Hence $C_r(c, o)$ decreases by an amount proportional to the entropy reduction. □

Example 66.8 (The Rosetta Stone as Repair Note)**.** The Rosetta Stone (196 BCE) is a repair note for the ancient Egyptian writing system. When Egyptian hieroglyphics became unreadable (the writing system was refused by history—practitioners died and the tradition collapsed), the Rosetta Stone provided a Bind event: it coupled the Greek text (whose continuation space was still accessible) to the Egyptian text (whose continuation space had become unreachable). The stone did not restore the original writing tradition, but it created a repair morphism from the collapsed tradition to a reconstructed one.

Programming Languages as Generative Notes

Theorem 66.9 (Programming Language as Maximum-Leverage Note)**.** *A general-purpose programming language L is a generative note with unbounded reachability*

leverage: $\Lambda(L) \to \infty$ *as the number of programs expressible in* L *grows.*

Proof. The continuation set $\mathcal{C}(L)$ includes all computations expressible in L. For a Turing-complete language, $|\mathcal{C}(L)|$ is infinite (countably many programs). The creation cost $C_{\text{create}}(L)$ is finite (the language was designed and implemented once). Therefore $\Lambda(L) = \sum_c \Delta P(c)/C_{\text{create}}(L)$ involves dividing a diverging numerator by a finite denominator: $\Lambda(L) \to \infty$. □

Legal Systems as Constraint Ledgers

Theorem 66.10 (Legal Code as Coordinated Refusal Note)**.** *A legal code is a Coordination Note that documents, for a community of agents, which actions are admitted* (*legal*)*, refused* (*illegal*)*, and under what conditions repair* (*amnesty, appeal, rehabilitation*) *is available.*

Proof. A law L that prohibits action a is a Refuse event applied to all agents in the jurisdiction: $\mathsf{Refuse}(a, \mathsf{IllegalUnder}(L))$. A law that requires action b is a Bind event: $\mathsf{Bind}(\text{citizenship}, b)$, coupling the agent's status to the obligation. Enforcement is the execution of the refusal: making refusal-violations actually inadmissible (with sanctions). The legal system's history is the sequence of legislative acts (Pop events on the space of possible legal rules) and judicial decisions (Refuse events on specific behavioral patterns). □

Example 66.11 (Precedent as Collapse Note)**.** A legal precedent (stare decisis) is a Collapse Note: it collapses the family of future cases similar to the precedent case into a single observable state (the holding). The precedent makes the collapse rule explicit: future courts must observe similar cases through the lens of the established holding, and may only refine or distinguish the rule (not ignore it).

Markets as Distributed Prospective Notes

Theorem 66.12 (Prices as Distributed Prospective Notes)**.** *A market price* p *for good* x *is a Prospective Note: it preserves access to the continuation of exchanging* x *at price* p *for all agents who observe the price signal.*

Proof. A price p is a public Bind event: $\mathsf{Bind}(\text{seller of } x, \text{buyer of } x \text{ at } p)$. It creates a prospective continuation—the exchange—that was not reachable before

the price was established. Without the price signal, buyers and sellers must individually search for counterparties (high reconstruction cost). With the price signal, the continuation of exchange is directly accessible (low reconstruction cost). Market prices therefore reduce the reconstruction cost of economic coordination. □

Chapter 67

History–Reachability Correspondence

Formal Statement

Throughout this monograph, histories and reachability have been treated as two aspects of the same underlying structure. This chapter makes the correspondence explicit.

Definition 67.1 (Reachability Volume of a History)**.** Let $\mathsf{Fut}(H)$ be the set of admissible continuations extending H (finite sequences of events that are admissible given H). The *reachability volume* of H is $V_R(H) = |\mathsf{Fut}(H)|$.

Theorem 67.2 (Pop Narrows Reachability)**.** *If $H' = H{+}{+}[\mathsf{Pop}(x)]$, then $V_R(H') \leq V_R(H)$.*

Proof. Every continuation extending H' is also a continuation extending H, since H' itself extends H. Thus $\mathsf{Fut}(H') \subseteq \mathsf{Fut}(H)$ and $V_R(H') \leq V_R(H)$. □

Theorem 67.3 (Refuse Narrows Admissible Reachability)**.** *If $H' = H{+}{+}[\mathsf{Refuse}(x, r)]$, then $V_R(H') \leq V_R(H)$.*

Proof. A continuation extending H' must respect the refusal of x: any continuation requiring an unrepaired Pop of x is excluded from $\mathsf{Fut}(H')$. Since all continuations extending H' are continuations extending H with an additional admissibility constraint, $\mathsf{Fut}(H') \subseteq \mathsf{Fut}(H)$. □

Corollary 67.4 (Computation as Reachability Shaping)**.** *A Spherepop computation is not merely a path from input to output. It is a monotone transformation of a reachability volume: commitment (Pop) and refusal (Refuse) both narrow the reachability volume, while Bind and Collapse reshape its structure without necessarily shrinking it.*

The History-Reachability Duality

Theorem 67.5 (History-Reachability Duality)**.** *There is a duality between the lattice of histories ordered by prefix extension and the lattice of reachability volumes ordered by subset inclusion:*

$$H_1 \leq H_2 \text{ (prefix)} \implies V_R(H_2) \leq V_R(H_1) \text{ (subset)}.$$

Proof. If H_1 is a prefix of H_2, then $H_2 = H_1 +\!\!+ H'$ for some non-empty H'. By repeated application of Pop Narrows Reachability and Refuse Narrows Admissible Reachability (one application per event in H'), $V_R(H_2) \leq V_R(H_1)$. □

Remark 67.6. The History-Reachability Duality is the formal content of the monograph's central thesis: computation is the progressive restriction of possibility. As the history grows (computation proceeds), the reachability volume shrinks. The history and the reachability volume are dual representations of the same computational trajectory.

The History-Reachability Correspondence Theorem

Theorem 67.7 (History-Reachability Correspondence)**.** *Let $\Phi : \mathcal{H} \to \wp(\mathcal{C})$ be the map sending history H to its reachability volume $\mathsf{Fut}(H)$. Then:*

1. *Φ is an order-reversing map between the prefix lattice on $\mathcal{H}$ and the inclusion lattice on $\wp(\mathcal{C})$.*
2. *Φ preserves the Conservation Law: $|\Omega(H)| + |\mathsf{consumed}(H)| = |\Omega_0|$.*
3. *$\Phi(\varepsilon) = \mathcal{C}_{\text{total}}$ (the empty history has maximal reachability).*
4. *$\Phi(H_{\text{terminal}}) = \varnothing$ when $\Omega(H_{\text{terminal}}) = \varnothing$.*

Proof. (1) is the History-Reachability Duality theorem. (2) follows from the Conservation of Possibility: committed events ($|\mathsf{consumed}(H)|$) plus remaining options ($|\Omega(H)|$) equal the initial option count. (3) holds because the empty history has imposed no commitments, so all continuations are reachable. (4) holds because if $\Omega = \varnothing$, no further Pop events are possible, so no continuation can extend H by selection. □

Remark 67.8. Theorem 67.7 is the unification theorem promised from the beginning of this monograph. It shows that Spherepop's two primitive concepts—the history that computation produces and the reachability volume that computation consumes—are formally dual. Each determines the other. A complete his-

tory determines a terminal reachability volume. A specified reachability volume constrains the possible histories that could have produced it.

This duality is the same duality that appears throughout the monograph in other guises: state versus history, observation versus trajectory, collapse versus reconstruction, notes as anti-collapse artifacts.

All of these are the same duality, seen from different angles.

Theorem Status Table

The following table records the status of principal results in this monograph, implementing the reviewer's recommendation for epistemic transparency.

Result	Chapter	Status
Conservation of Possibility	3	Proved
Completeness of Four Operators	3	Sketch
Operator Representation Theorem	45	Proved
History Monoid (free monoid)	4	Proved
Hist as free strict monoidal category	4	Proved
Unique History Factorisation	4	Proved
History Monotonicity	5	Proved
Option Space Monotonicity	5	Proved
Collapse Soundness	7	Proved
Type Preservation (full)	40	Proved
Progress (full)	40	Proved
Type Soundness Corollary	40	Proved
Canonical Forms Lemma	40	Proved
Weakening Lemma	40	Proved
Exchange Lemma	40	Proved
Substitution Lemma	40	Proved
Admissibility Preservation	40	Proved
Pop Entropy Reduction	42	Proved
Refuse Admissibility Entropy Reduction	42	Proved
State Illusion (Proposition 2.2)	2	Proved
No Direct Observation Theorem	17	Proved
Ephemerality Inversion	13	Proved

Collapse Functor Functoriality	6	Proved
Collapse as Reflection (adjunction)	20	Proved
Universal Property of Collapse	6	Proved
History-Observation Adjunction	20	Sketch
Notes as Partial Right Adjoints	25	Sketch
State Illusion as Sheaf Non-gluing	20	Proved
Refusal Obstruction Prevents Gluing	47	Proved
Civilizational Collapse Theorem	21	Proved
Percolation Phase Transition	38	Sketch
Replay Equivalence	10	Proved
Compiler Full Faithfulness	22	Proved
GC Safety (Certificate-Only)	15	Proved
Error Correction via Certified Refusal	16	Proved
Eve's Law as Pythagorean Theorem	31	Proved
Tower Property as Nested Projection	31	Proved
Dependency Propagation Theorem	47	Proved
Dependency Closure Theorem	47	Proved
Deadlock as Graph Cycle	34	Proved
Linearization Theorem	48	Proved
Collapse Invariance (Accumulate)	48	Proved
History-Reachability Duality	51	Proved
History-Reachability Correspondence	51	Proved
Library Reachability Theorem	50	Proved
Scientific Progress as Directed Refusal	50	Proved
Repair Soundness	47	Proved
Notes Form a Category	46	Proved
Knowledge Leverage Theorem	46	Proved
Entropy Lower Bound on Reconstruction	23	Proved
Optimal Note Theorem	48	Proved
Bisimulation Is Congruence	34	Proved
CAP as Collapse Tradeoff	34	Sketch
CRDTs Achieve Eventual Consistency	34	Proved

Repair Cohomology (obstruction)	26	Conjecture
Dependent Process Types (full)	8	Programmatic
Distributed Spherepop Runtimes	34	Programmatic
CoC Strong Normalization	24	Sketch
Proof-Carrying Refusal (full system)	8	Programmatic
Admissibility Lattice Completeness	8	Programmatic

Status codes: *Proved*: formal proof given in this manuscript. *Sketch*: main argument given; details omitted or rely on standard references. *Conjecture*: proposed result, no proof given. *Programmatic*: research direction; result not yet formulated precisely enough to prove.

Bibliography

[1] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] Borgman, C. L. *Scholarship in the Digital Age*. MIT Press, 2007.

[3] Clark, A. and Chalmers, D. The extended mind. *Analysis* 58, no. 1 (1998): 7–19.

[4] Eisenstein, E. L. *The Printing Press as an Agent of Change*. Cambridge University Press, 1980.

[5] Feldman, J. On the absolute continuity of Gaussian measures. *Zeitschrift für Wahrscheinlichkeitstheorie* 1 (1958).

[6] Felleisen, M. and Friedman, D. P. A calculus for assignments in higher-order languages. *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987.

[7] Chenoweth, C., Chenoweth, A. MEM|8: A Wave-Based Cognitive Architecture for Multimodal Memory Integration and Consciousness Simulation. Zenodo, 2025. `https://doi.org/10.5281/zenodo.16436298`

[8] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[9] Borgman, C. L. *Scholarship in the Digital Age*. MIT Press, 2007.

[10] Coquand, T. and Huet, G. The calculus of constructions. *Information and Computation* 76 (1988): 95–120.

[11] Clark, A. and Chalmers, D. The extended mind. *Analysis* 58, no. 1 (1998): 7–19.

[12] Eisenstein, E. L. *The Printing Press as an Agent of Change*. Cambridge University Press, 1980.

[13] Feldman, J. On the absolute continuity of Gaussian measures. *Zeitschrift für Wahrscheinlichkeitstheorie* 1 (1958).

[14] Felleisen, M. and Friedman, D. P. A calculus for assignments in higher-order languages. *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987.

[15] Fowler, M. Event sourcing. `martinfowler.com`, 2005.

[16] Girard, J.-Y., Lafont, Y., and Taylor, P. *Proofs and Types*. Cambridge University Press, 1989.

[17] Goody, J. *The Domestication of the Savage Mind*. Cambridge University Press, 1977.

[18] Hájek, J. On a property of normal distributions of an arbitrary stochastic process. *Czechoslovak Mathematical Journal* 8 (1958).

[19] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, 1985.

[20] Hofmann, M. and Streicher, T. The groupoid interpretation of type theory. In *Twenty-five Years of Constructive Type Theory*. Oxford University Press, 1998.

[21] Hutchins, E. *Cognition in the Wild*. MIT Press, 1995.

[22] Knuth, D. E. Literate programming. *The Computer Journal* 27, no. 2 (1984): 97–111.

[23] Mac Lane, S. *Categories for the Working Mathematician*. Springer, 1971.

[24] Martin-Löf, P. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[25] Milner, R. *A Calculus of Communicating Systems*. Springer, 1980.

[26] Naur, P. (ed.). Report on the algorithmic language ALGOL 60. *Communications of the ACM* 3, no. 5 (1960): 299–314. (Original introduction of Backus–Naur Form.)

[27] Nordström, B., Petersson, K., and Smith, J. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[28] Ong, W. J. *Orality and Literacy: The Technologizing of the Word*. Methuen, 1982.

[29] Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.

[30] Reiter, R. On closed world data bases. In *Logic and Data Bases*. Plenum Press, 1978.

[31] Santoro, M., Waghmare, S., and Panaretos, V. M. Kernel embeddings of functional data and mutual singularity. Preprint, 2024.

[32] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. INRIA Technical Report, 2011.

[33] Strachey, C. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation* 13 (2000): 11–49. Originally delivered 1967.

[34] Tulving, E. Episodic and semantic memory. In *Organization of Memory*. Academic Press, 1972.

[35] The Univalent Foundations Program. *Homotopy Type Theory: Univalent*

Foundations of Mathematics. Institute for Advanced Study, 2013.

[36] Wright, A. K. and Felleisen, M. A syntactic approach to type soundness. *Information and Computation* 115, no. 1 (1994): 38–94.