Against the Kitchen Sink: On Vendor-Gravity Languages, Cognitive Load, and What Spherepop Proves

Flyxion

July 2026

There is a particular flavor of exhaustion that comes from opening a C++ header file and realizing that before you write a single line of logic you must first make peace with templates, with the preprocessor, with three overlapping systems of polymorphism, with a memory model that assumes you are simultaneously a systems engineer and a masochist, and with forty years of accumulated syntax that nobody had the courage to deprecate. Swift offers a gentler version of the same exhaustion: a language whose elegance is real but whose gravity well is Apple's own, whose idioms bend toward one platform's frameworks, and whose long-term legibility outside that platform is perpetually uncertain. Neither language is proprietary in the narrow legal sense — C++ is an ISO standard and Swift is open source — but both exhibit what might be called vendor gravity: a center of mass so heavily determined by one corporate ecosystem, one toolchain, one set of institutional priorities, that the language's evolution is never quite free of its patron. Microsoft's tooling shapes how C++ is actually written in practice across enormous swaths of industry; Apple's frameworks shape what Swift is actually for. The result in both cases is the same complaint dressed in different clothes: a language that increases cognitive load not because complexity is intrinsically necessary to the problems being solved, but because complexity has been allowed to accrete as a side effect of institutional history, backward compatibility obligations, and the gravitational pull of a single vendor's roadmap.

This is not a small aesthetic quibble. Cognitive load has real costs. Every hour spent memorizing the difference between `std::unique_ptr` and `std::shared_ptr`, or reasoning about ARC retain cycles in Swift closures, is an hour not spent thinking about the actual problem the software is meant to solve. Worse, these languages actively resist interoperability and diversity of thought. A codebase steeped in C++ template metaprogramming becomes a fortress that only initiates can enter; a Swift codebase wired tightly into UIKit or SwiftUI becomes portable in name only. The languages do not merely fail to reduce cognitive load — they actively convert cognitive load into a moat, a barrier to entry that happens to correlate suspiciously well with vendor lock-in. Learning C++ deeply is learning to think the way three decades of committee compromise wanted you to think. Learning Swift deeply is learning to think the way Apple's frameworks want you to think. Neither is learning to think about computation itself.

1 Three Kinds of Gravity

It is worth being more precise about what vendor gravity actually names, because the phenomenon is not uniform across languages and collapsing it into a single complaint invites an easy rebuttal. C++ suffers from what is better called committee gravity: no single corporation owns it, but the ISO standardization process guarantees that every proposed feature must survive contact with

decades of existing code that nobody is willing to break, so the language accretes rather than revises. Swift suffers from vendor gravity in the narrower sense: a single company's frameworks, release cadence, and platform priorities determine which idioms are rewarded and which quietly atrophy, so that writing Swift divorced from Apple's ecosystem is possible in principle but almost never observed in practice. Python occupies a third category worth naming honestly rather than exempting by omission: ecosystem gravity, in which no single actor dominates, but the accumulated weight of millions of existing libraries, tutorials, and expectations constrains future design almost as effectively as a corporate roadmap would. Guido van Rossum's own departure from the language's steering did not free Python from gravity; it simply confirmed that the gravity had migrated from a person to a community. The honest claim is not that Python is free of historical accretion while C++ and Swift are shackled by it. The honest claim is that Python's gravity is distributed and permissive, rewarding many idioms rather than enforcing one, while committee gravity and vendor gravity concentrate their weight into fewer sanctioned paths and therefore produce a sharper, more exclusionary form of cognitive load. All three are instances of the same underlying phenomenon, which deserves a name of its own before Spherepop can be introduced as a counterexample to it.

2 Essential, Accidental, and Historical Complexity

Fred Brooks gave the field its classic distinction between essential complexity, the complexity inherent in the problem domain itself, and accidental complexity, the complexity introduced by the tools and representations used to solve it [1]. That distinction is necessary here but not sufficient. C++ and Swift are not merely guilty of accidental complexity in Brooks's sense, where a poor representation makes an essentially simple problem look hard. They are guilty of a third category that deserves its own name: historical complexity, meaning complexity that is neither demanded by the problem nor introduced by a single bad design choice, but inherited wholesale from the obligation to remain compatible with every previous design choice the language has ever made. Raw pointers were not replaced by smart pointers in C++; they were supplemented by them, and then supplemented again by move semantics, and again by shared and unique ownership models, until a single problem — who owns this memory and when may it be freed — accumulated several partially overlapping answers that all remain valid simultaneously because none could ever be retired [14]. This is not accidental complexity in Brooks's sense, because no individual mechanism among them is poorly designed in isolation. It is historical complexity: the sedimentary residue of a language that has never been permitted to forget anything it once committed to, which is ironically the inverse of the discipline Spherepop takes as its founding virtue. A language that cannot forget its own past mistakes is not the same as a language that deliberately preserves history as a first-class value; the former is a fossil record accumulated by necessity, the latter is an ontological commitment chosen on purpose, and the difference between those two is most of what this essay is actually about.

3 Language Archaeology

Treating C++ as a fossil record rather than as a designed artifact clarifies why the complaint about cognitive load is not really about difficulty. Difficulty alone would not distinguish C++ from Haskell, which is also difficult but difficult in a different register: Haskell asks the programmer to learn a small number of deep ideas — laziness, purity, the type class hierarchy — and once those ideas are internalized, the rest of the language falls out coherently from them. C++ asks the programmer to learn a large number of shallow idioms, most of which exist only because an earlier idiom in the

same language proved insufficient and could not be removed. Excavating a C++ codebase written across multiple decades is genuinely archaeological: raw pointers from the 1980s coexist with `unique_ptr` from C++11, which coexists with move semantics refined in C++14, which coexists with ranges introduced in C++20, each stratum legible to a specialist trained in that particular decade's idioms and increasingly opaque to newcomers who must learn all strata simultaneously just to read code that predates them. These strata do not merely sit side by side; they actively distort one another. The interaction between lvalue and rvalue references, perfect-forwarding templates, and smart-pointer ownership models does not produce a linear list of idioms to memorize one at a time; it produces a combinatorial explosion of edge cases at every point where the strata touch, since a template written to forward arguments generically must now reason correctly about ownership semantics that did not exist when the forwarding idiom was designed, and a smart pointer must now interact correctly with move semantics that were bolted on after the ownership model was already in wide use. The cost of historical complexity, in other words, is not merely additive. It compounds at the seams between strata, which is precisely where most of the genuinely obscure C++ bugs live. Swift's archaeology is younger but structurally identical in miniature: completion handlers gave way to `async/await`, but completion handlers did not disappear from the ecosystem, and a working Swift programmer today must still be able to read both. Naming this archaeology explicitly reframes the complaint about bloat. The problem is not that these languages have many features. The problem is that they have many features solving the same problem, each surviving because deprecation would break someone's existing build, which means the language's size grows monotonically while its coherence does not.

4 Initiation Rites

Every mature language ecosystem accumulates a body of knowledge that functions less like a computational principle and more like a survival skill specific to that ecosystem: do not capture `self` strongly inside a Swift closure, remember the rule of five in C++, never let an exception cross this particular boundary, watch for iterator invalidation after a container resize. None of these are truths about computation. They are truths about the specific historical accidents of one language's implementation, and they must be memorized and transmitted socially — in code review comments, in style guides, in the folklore passed from senior engineers to junior ones — precisely because nothing about the language's surface grammar would lead a newcomer to discover them independently. This is the initiation-rite structure of vendor-gravity languages: competence is measured not by understanding computation more deeply, but by having survived enough encounters with the language's specific historical accidents to have internalized its unwritten rules. Spherepop's ambition, whatever its practical limitations, is that the rules should not need to be unwritten. If capturing a value strongly creates a retain cycle, that should be visible as a structural fact about Bind narrowing the option space in a particular way, not as a piece of folklore that must be separately taught. The test of whether a language demands initiation rites is simple: can a newcomer derive the correct behavior from the primitives alone, or must the primitives be supplemented by a body of community-transmitted lore that is, in effect, a second and unwritten language layered on top of the first?

5 Where the Elegant Languages Fall Short of Adoption

Rust, Haskell, and Lisp deserve to be named honestly in this rant, because their elegance is not in dispute. Rust's ownership model actually solves the memory safety problem that C++ merely

gestures at solving with a library of smart pointers and prayer, and it does so as a structural property of the type system rather than as folklore to be memorized [13]. Haskell's purity and its type system make entire categories of bugs structurally impossible rather than merely discouraged by convention. Lisp's homoiconicity — code as data, data as code — remains one of the cleanest ideas in the history of computing, and every macro system built since is a pale tribute to it [3]. And yet none of these languages carries the library depth or the documentation density of Python, or the market saturation of C++ and Swift. This is the tragedy of elegance in programming languages: correctness and clarity of design do not automatically produce adoption, and adoption is what produces the enormous secondary literature — the Stack Overflow answers, the tutorials, the battle-tested libraries — that make a language usable at scale by people who are not language theorists. Rust is closing this gap faster than Haskell or Lisp ever did, precisely because it married elegance to a pragmatic systems-programming use case that industry actually needed. But the general principle holds: a beautiful language with three libraries and a devoted cult following is not yet a practical alternative to a bloated language with three hundred thousand libraries and a single vendor's institutional weight behind it. Python earns its praise in this same comparison by contrast, and the contrast is instructive precisely because Python is not the most elegant language on the list. It is not the fastest, and its type system is an afterthought bolted onto a dynamically typed core. But Python's genius is that it never demanded initiation rites in the sharp, exclusionary sense that C++ and Swift do. It can be written functionally, with map, filter, comprehensions, and generators doing real work; it can be written in a plain imperative style that reads like pseudocode; and crucially, it has accumulated an enormous ecosystem of prebuilt libraries through distributed ecosystem gravity rather than a single vendor's product strategy. NumPy did not need Microsoft's blessing. Pandas did not need Apple's frameworks.

6 A Lineage, Not an Invention

None of this is new in kind, only in emphasis, and Spherepop should be situated honestly within a long tradition of attempts to discover minimal computational foundations rather than presented as though it arrived from nowhere. Alonzo Church showed that the entirety of computation could be derived from function abstraction and application alone [2]. John McCarthy showed, with Lisp, that a handful of primitives — quote, atom, eq, cons, car, cdr, cond — sufficed to build an entire self-describing language, code and data sharing one representation [3]. Christopher Strachey's work on denotational semantics insisted that a language's meaning should be derivable compositionally from a small set of semantic primitives rather than defined operationally feature by feature [4]. Dijkstra spent much of his career arguing, often abrasively, that most of what passes for necessary complexity in programming is in fact a failure of discipline, and that guarded commands and structured reasoning could replace whole taxonomies of ad hoc control flow [5]. Alan Kay's original vision of object orientation was itself a minimalism — everything is a message send — that the mainstream languages claiming his lineage today have mostly abandoned in favor of exactly the kind of feature accretion this essay is complaining about [6]. Spherepop's contribution to this lineage is narrow but specific: where Church, McCarthy, Strachey, Dijkstra, and Kay each found minimal foundations for computing values, states, or messages, Spherepop asks what happens if the primitive is neither value nor state nor message but event, and if identity itself is redefined as the history of events rather than as a snapshot at any single moment. That reframing is what makes the four primitives — Pop, Refuse, Bind, Collapse — sufficient to derive conditionals, loops, assignment, error handling, and object identity without borrowing a separate syntactic apparatus for each.

7 The Primitives

Spherepop's ontology is deliberately small. There is Pop, the strict elimination of one or more options from a space of possibilities, with the entropy of the system strictly decreasing as a result. There is Refuse, geometrically identical to Pop but semantically marked as rejection rather than selection — a primitive most languages do not even recognize as distinct from ordinary branching, even though refusal and selection are experientially and causally different acts. There is Bind, an endomorphism that narrows the admissible family of continuations without collapsing anything, the mechanism by which dependency enters a system without yet forcing commitment. And there is Collapse, a quotient morphism that projects the full causal history onto an observable state, irreversibly destroying the provenance of how that state was reached. Everything else in the calculus, including its proven Turing completeness via equivalence to the lambda calculus, is built from compositions of these four operators inside a strict monoidal category. No preprocessor. No fifteen kinds of initialization syntax. No ARC. Four verbs and a commitment to never allowing history to be silently erased.

It is worth being precise about what kind of claim the idiom translations that follow are actually making, because a formalist would rightly ask whether "conditional equals Pop" is an exact equivalence or merely a suggestive analogy. The honest answer is that within the calculus the mappings are exact — Pop, Refuse, Bind, and Collapse are formally defined morphisms in a strict monoidal category, and the reduction of conditionals, loops, and assignment to compositions of them is a derivation, not a metaphor. What remains interpretive is the claim that this derivation is the natural or canonical one, rather than one derivation among several possible ones. A rigorous defense of that stronger claim belongs in an appendix with a literal step-by-step reduction of a concrete program, not merely in the prose analogy offered here; the sketch that follows should be read as showing that the reduction exists and is coherent, not as proving it is unique.

8 Idiom Translations

Translating familiar programming idioms into this vocabulary is where the contrast with C++ and Swift becomes sharp. A conditional branch, in most languages a syntactic special form with its own keyword, its own scoping rules, and its own edge cases around dangling `else` clauses, is in Spherepop nothing more than a Pop over the option space $\{\text{then-branch}, \text{else-branch}\}$: one branch is selected, the other is eliminated, and the elimination itself is recorded as part of the system's permanent history rather than discarded. A loop, which in C++ requires choosing among `for`, `while`, `do-while`, range-based iteration, and iterator invalidation hazards, is in Spherepop a repeated Bind narrowing an option space toward a terminal condition, followed by a Pop that exits the loop by eliminating the option of continuing — iteration is not a separate control construct bolted onto the language but simply repeated application of the same two primitives already used for everything else. Variable assignment, which in Swift carries the entire freight of value versus reference semantics, `let` versus `var`, and property wrappers, is in Spherepop a Bind: a narrowing of the system's future admissible continuations to those consistent with the newly introduced dependency, with no separate concept of storage location required because identity was never state-based to begin with.

Error handling is where the idiom translation is most pointed, because it is where C++ and Swift are most visibly incoherent with each other and with themselves. C++ offers exceptions, error codes, `std::optional`, and `std::expected`, four incompatible idioms coexisting uneasily in the same codebase depending on which decade the code was written in. Swift offers `try/catch`, optionals,

and `Result`, three idioms that Swift's own standard library cannot always agree on. Spherepop collapses all of this into Refuse: an error is not a special control-flow escape hatch requiring its own syntax, but simply the semantically marked rejection of an option, recorded in history exactly the way a successful Pop would be, differing only in its tag. There is no separate exceptional path through the program, because Spherepop never had an ordinary path that exceptions could deviate from in the first place — every step is already a selection or rejection among options, and failure is just one more instance of that same primitive. One clarification is worth making explicit here rather than leaving implicit until the appendix: Bind and Refuse are not competing mechanisms for the same job. Bind narrows which options are admissible before any commitment is made; Refuse is the commitment operator that records, after the fact, that a particular option was eliminated because it failed to be admissible. An admissibility filter applied by Bind can rule an option out of the residual space, but it is Refuse that turns that exclusion into a permanent entry in the history — the filtering and the recording are two different operators because a system may need to know not only what is currently permitted, but specifically what was tried and rejected, which a silently narrowed option space alone would not preserve.

Object mutation and the murky question of object identity, which in C++ produces the entire apparatus of copy constructors, move semantics, and rule-of-five boilerplate, and which in Swift produces the value-type-versus-reference-type distinction that trips up nearly every newcomer, is in Spherepop simply Collapse: the moment at which an accumulated event history is projected onto an observable state. Two objects are "the same" in Spherepop not because they occupy the same memory address or satisfy some equality operator, but because they are collapses of the same underlying history — identity is provably a sheaf of histories rather than a pointer value, which sidesteps the entire aliasing-versus-copying debate that consumes so much of a C++ or Swift programmer's cognitive budget.

9 A Worked Example

The claim is easiest to evaluate against a single small program rather than against abstractions. Consider summing the positive numbers in a list, a task trivial enough in every language that the comparison isolates syntax and conceptual overhead rather than genuine difficulty. In Python the task requires nothing beyond a running total and a comprehension, written as a single sum over a generator expression that filters as it accumulates, and the entire cognitive burden is that one built-in construct reused for filtering and summing alike. In C++ the same task, written idiomatically rather than minimally, requires a choice among a raw loop, `std::accumulate` with a lambda, or a ranges-based pipeline depending on which decade of the language the author learned, and each choice carries its own idioms for how the lambda captures its environment, whether the container is iterated by reference or by value, and whether the accumulator's type must be specified explicitly to avoid narrowing conversions. In Swift the task requires deciding between a `for` loop with an `if` guard, a `filter` followed by a `reduce`, and reasoning about whether the array is a value type being copied or a reference type being shared across the closure boundary. In Spherepop the task is a single Bind narrowing the option space of the accumulator across the sequence, composed with a Pop at each element that either includes it in the running history or refuses it, and a final Collapse that projects the accumulated history onto the observable sum; no separate syntax exists for loop, filter, or accumulate, because all three are already the same composition of the same two primitives viewed from different angles. The point of the comparison is not that Spherepop is shorter — Python is already about as short as this task can be written — but that Spherepop requires zero additional primitives beyond the four it already has to express filtering, iteration,

and accumulation together, while C++ and Swift each require the programmer to choose among several genuinely different mechanisms for what is conceptually one operation.

10 What Computation Forgets

Garbage collection, the perennial tradeoff between manual memory management and runtime overhead that both languages handle with elaborate and mutually incompatible machinery — reference counting with retain cycles in Swift, RAII and smart pointers in C++ — is usually treated as an unavoidable tax that any serious language-design argument must eventually concede to. A C++ programmer confronted with the claim that Spherepop retains history monotonically would object, correctly, that physical machines have finite storage, and that history never being erased cannot literally mean that nothing is ever reclaimed. The objection is right, but it targets a version of the claim Spherepop does not actually make. Spherepop does not eliminate the need to reclaim resources; it relocates the question from an operational one to a semantic one. Conventional languages ask when an object may be deleted, a question answered by tracking reference counts, ownership graphs, or reachability from a root set — mechanisms that are themselves a significant fraction of the historical complexity catalogued earlier in this essay. Spherepop instead asks which portions of causal history may be forgotten without violating the invariants of the computation, which reframes memory management as a special case of history compression rather than as a separate subsystem bolted onto the language. Collapse already performs exactly this operation at the level of a single value, projecting accumulated history onto an observable state and discarding provenance; nothing prevents the same operation from being applied at a coarser grain to entire regions of the event graph that are causally sealed off from all future continuations, which is the Spherepop analogue of garbage collection, except that what is discarded is chosen because it is causally irrelevant rather than because a reference count happened to reach zero. This distinction matters because reference counting and mark-and-sweep are both answers to the operational question of when memory happens to become unreachable, which is a fact about the implementation, not a fact about the computation's meaning; asking which history may be forgotten without violating invariants is a question about meaning first, with the implementation's storage reclamation following as a consequence rather than preceding as a mechanism the programmer must reason about directly.

The deeper claim beneath this reframing is that most languages, including Python, are optimized for state manipulation: variables overwrite previous values, objects mutate in place, and any record of how the current state was reached is optional, discarded by default unless the programmer deliberately logs it. Spherepop inverts that default. History is primary; state is a projection obtained by Collapse when and only when observation is required. This is not merely a memory-management convenience but the essay's actual philosophical center, and it connects outward to the rest of the surrounding research program rather than standing alone: to an event-log conception of memory in which recall is reconstruction from history rather than retrieval from a fixed store, to a broader insistence that a computation is a trajectory rather than a transition between snapshots, and to a continuation-and-repair framework in which preserving admissible histories matters more than preserving any single instantaneous state. Seen from that vantage, Spherepop is not really competing with Python, Rust, or C++ as a tool for getting work done faster. It is proposing that the entire category of programming language, as commonly understood, has quietly assumed that forgetting is cheap and remembering is expensive, and demonstrating what a language looks like when that assumption is reversed.

11 History as a Computational Primitive

The argument so far has treated history as a virtue Spherepop happens to preserve. It is worth stating more sharply what kind of claim that actually is, because "history matters" is a much weaker thesis than the one this essay has been building toward. Imperative programming, from Fortran through C++ and Swift, treats state as primitive: a program is a sequence of mutations to a store, and everything else — functions, objects, control flow — exists to organize when and how the store changes. Functional programming, from Lisp through Haskell, treats transformation as primitive instead: a program is a composition of pure functions, and state, where it appears at all, is either threaded explicitly through arguments and return values or quarantined behind a monad boundary. Object-oriented programming, in Kay's original and largely abandoned sense, treats message exchange as primitive: a program is a society of objects whose identity is constituted by what messages they can receive and how they respond, state being a private implementation detail of each object rather than a global resource [6]. Spherepop's claim belongs in this same list, not beside it: it treats event history as primitive, and state, transformation, and message exchange all become derived notions — a state is a Collapse of a history, a transformation is a Bind narrowing a history's future continuations, and a message exchange is simply two histories becoming causally entangled through shared Pops. This is the taxonomy the earlier sections have been implicitly building toward. C++ and Swift are not merely badly designed; they are commitments to a particular computational ontology, the state-primitive one, and the cognitive load this essay has been cataloguing is in large part the cost of maintaining that ontology's bookkeeping — tracking who owns a piece of state, who may mutate it, and when a mutation becomes visible to whom. A language built on a different ontology does not inherit that bookkeeping, because it was never asking the question that made the bookkeeping necessary in the first place.

12 Observability Is Already Collapse

Of the four Spherepop primitives, Pop, Refuse, and Bind tend to strike readers as intuitive almost immediately, because they resemble selection, rejection, and constraint, three ideas every programmer already has informal names for. Collapse is the operator that feels genuinely unfamiliar, and it is worth spending some time showing that it is not actually new so much as newly named. Every mainstream programming language performs Collapse constantly; it simply does not call the operation anything, and does not treat it as a first-class term in the language's own grammar. Rendering a user interface is a Collapse: an enormous, continuously accumulating history of user input, network responses, and internal state changes gets projected onto a single visible frame, and everything about how that frame was reached is discarded the instant the next frame is drawn. Serializing an object to JSON or to a database row is a Collapse: the object's full history of construction, mutation, and intermediate computation is thrown away, leaving only a snapshot that asserts its own timelessness. Logging a value is a partial Collapse, one that happens to retain a fragment of provenance the rest of the system has already discarded, which is exactly why logs feel indispensable — they are the one place a state-primitive system admits, almost apologetically, that history was worth keeping after all. Printing program output is a Collapse. A database query, at the moment it returns a result set, is a Collapse of an enormous and continuously mutating table history onto a single answer valid only at the instant the query ran, which is precisely why "the data changed between when I queried it and when I acted on it" is one of the oldest bug categories in software engineering. The claim, then, is not that Spherepop invents Collapse. The claim is that every mainstream language already performs Collapse everywhere, constantly, while treating it as an invisible implementation detail

rather than a term the programmer can name, reason about, and compose deliberately. Making Collapse explicit does not add a new operation to programming. It gives a name to an operation every program was already doing in secret.

13 The Hidden Cost of State

State-primitive languages market themselves as cheap, and in a narrow sense they are: a mutable variable costs one machine word and one assignment instruction. What that accounting leaves out is everything state requires in order to remain trustworthy once a program grows past toy size. Aliasing — two names referring to the same mutable location — has to be tracked, or reasoned about defensively, or ruled out entirely by a borrow checker, which is exactly the problem Rust's ownership system exists to solve and exactly the problem C++'s six coexisting pointer idioms exist to paper over [13]. Synchronization has to be introduced the moment more than one thread might mutate the same state, an entire subfield of systems programming built on the premise that shared mutable state is dangerous by default. Mutation tracking, rollback, and undo all have to be built as separate infrastructure precisely because the state-primitive default throws away the information they need: once a variable is overwritten, the language itself no longer knows what it used to be, so any system that wants to know must reconstruct that knowledge by other means. Debugging provenance — the question of how a program arrived at some incorrect state — is answered not by the language, which has already discarded the answer, but by stepping through a debugger and manually reconstructing a history the language chose not to keep. What makes this cost easy to overlook is that software engineers have spent decades quietly reinventing history-first structures inside state-first systems, as though repeatedly rediscovering the same solution to the same unacknowledged problem. Git is a history-first structure bolted onto languages whose values it commits are otherwise mutable and forgetful [11]. Database transaction logs and write-ahead logs exist because the database's own row-level state cannot be trusted to explain itself after a crash. Event sourcing, an entire architectural pattern in enterprise software, exists to replace "the current balance is X" with "here is the sequence of deposits and withdrawals that produced X," precisely because state alone turned out to be an insufficient record for auditing, debugging, and recovery [12]. Undo trees in editors, audit trails in compliance systems, and the logs underlying distributed consensus protocols such as Raft and Paxos are all the same pattern recurring: a state-primitive foundation, discovered under real engineering pressure to be inadequate, gets a history-primitive structure grafted onto it after the fact [7, 10]. Spherepop's proposal is not to introduce this pattern. It is to stop treating it as an afterthought bolted onto every sufficiently large system, and to make it the foundation instead.

14 Debugging Is Archaeology

Every debugging session begins with a question that sounds like a state question but is not: why did the program reach this state? A programmer staring at a stack trace, a core dump, or an unexpected value in a variable is not actually asking what the state is — the debugger already shows that plainly. They are asking how the state came to be that way, which is a question about a causal path through time that the state-primitive language has already discarded by the time the question is asked. The entire apparatus of practical debugging exists to answer a history question using a system that was never designed to retain history. Stack traces are a partial, automatically retained fragment of the call history leading to a crash. Execution traces and logging statements are the programmer manually reintroducing the history the language declined to keep by default. Git bisect is a search procedure

for locating the historical commit that introduced a regression, which only works because Git, unlike the language it version-controls, actually kept every state along the way. Time-travel debuggers, which checkpoint or record enough information to step a program backward as well as forward, are explicit admissions that forward-only, history-discarding execution is an obstacle to understanding rather than a neutral default. Database audit logs and distributed tracing systems exist for the identical reason at the systems level: a request's behavior cannot be explained by its final state alone, so an entire secondary infrastructure is built to reconstruct the request's causal path across services that, individually, remember almost nothing about how they got where they are. What all of these tools have in common is that they are archaeology: careful, effortful reconstruction of a past that the primary system was not built to preserve. In a Spherepop program this reconstruction work does not need to happen, because there is nothing to reconstruct. The history a debugger labors to recover from stack traces, logs, and audit tables is, in Spherepop, simply H_t — already present, already complete, already the thing the program has been building the entire time rather than a side channel bolted on to compensate for the language's amnesia. Debugging in a history-primitive language is not archaeology. It is reading.

15 Concurrency as Composition of Histories

State-based languages carry an enormous apparatus for concurrency — locks, mutexes, atomics, memory barriers, transactional memory, increasingly elaborate ownership systems — and all of it exists to answer one question: who may mutate this shared state, and when. The question is posed that way because state-primitive languages treat a piece of mutable memory as a single, unowned resource that multiple agents are racing to modify, and the entire machinery of concurrency control is the accumulated cost of preventing those races from corrupting the resource. A history-primitive foundation does not eliminate concurrency, but it does change the question being asked. Instead of who may mutate this state, the question becomes how do multiple histories compose — and that is a question with a much older and better-developed mathematical answer than lock-based synchronization ever provided. Lamport's observation that distributed events form a partial order under a happens-before relation, rather than a single global sequence, was already an argument that concurrent computation is naturally a composition of causal histories rather than a scramble over shared state [8]. Spherepop's formalization as a strict monoidal category makes this composition explicit and structural rather than incidental: two histories produced independently by different agents compose via the category's tensor product, and the interesting question is not whether one agent's Pop happened before or after another's in wall-clock time, but whether their two histories can be coherently combined into a single further history at all. A genuine conflict — two agents attempting to Collapse the same underlying option space onto incompatible observable states — surfaces as a structural fact about whether the composition is well-typed, checkable at the moment of composition, rather than as a runtime race condition that only manifests probabilistically under load and that a lock must be purchased in advance to prevent. This does not make concurrent Spherepop programs trivial to reason about; composing causal histories correctly is its own discipline. But it replaces a defensive architecture built around the fear of simultaneous mutation with a compositional architecture built around the question of when two histories are allowed to merge, which is a question the category-theoretic structure was already equipped to answer before concurrency was ever posed as a separate problem.

16 Irreversibility and Information Loss

Every mainstream language treats assignment as though it were a perfectly ordinary, repeatable operation. The statement `x = 5;` looks identical in kind to the statement `x = 9;` that might follow it, and nothing in the syntax marks the second statement as having done something the first one did not: destroyed a value that can no longer be recovered by any means internal to the language. The operation was irreversible the entire time. The language simply declined to say so. Spherepop's insistence that Pop and Refuse are explicitly irreversible operations, each one strictly decreasing the entropy of the residual option space, is not an added restriction so much as an admission of something that was already true of ordinary mutation and that mainstream languages have simply never required the programmer to notice. This connects to a genuine physical fact rather than a merely stylistic one: Landauer's principle establishes that erasing a bit of information has an unavoidable thermodynamic cost, precisely because irreversible logical operations correspond to real entropy increase in the physical substrate performing them [9]. A language that hides its irreversible operations behind syntax indistinguishable from its reversible ones is not lying about physics, but it is declining to represent a distinction that physics itself insists is real. There is a precise symmetry worth stating plainly: conventional languages treat erasure as free at the level of syntax, and the cost that erasure does not pay in thermodynamic terms at the language level, it pays instead in cognitive load and synchronization friction at the engineering level — the aliasing bugs, the debugging archaeology, and the ad hoc undo systems catalogued in earlier sections are the accounting entries for exactly the erasure that Landauer's principle says was never actually free. Spherepop does not eliminate that cost. It relocates it from a hidden systemic tax, paid unpredictably wherever an overwritten value turns out to have mattered later, to an explicit entry in the history, paid once, at the moment the elimination occurs and is named. The practical cost of that concealment shows up everywhere serious software already treats certain operations with more caution than others: a database commit, a financial charge, and the act of launching a rocket are all irreversible in exactly the sense a loop counter's increment is not, and yet in most languages all four are written with the same assignment syntax, distinguished only by convention, code comments, and the discipline of the engineer, rather than by anything the language itself enforces. Scientific measurement exhibits the identical structure outside of computing entirely: an experiment collapses a space of live hypotheses in a way that cannot be undone by disbelieving the result, which is why replication rather than retraction is the field's actual remedy for a wrong measurement. Decision theory treats commitment the same way, modeling a choice as the elimination of forgone alternatives rather than as a reversible selection among them. Spherepop does not invent irreversibility. It refuses to let the language's syntax pretend the operation was ever anything else.

17 State as Compressed History

It would be easy to read the preceding sections as arguing that state is a mistake and history is the truth state-primitive languages have been hiding from. That reading is stronger than the essay actually needs and weaker than the truth. The more defensible claim is a duality rather than a verdict: a state is a compressed history, and a history is an expanded state trajectory, and neither description is more real than the other so much as suited to a different purpose. Collapse already behaves exactly like a compression operator in the technical sense — it is lossy, it is chosen deliberately from among several possible reduction rules for a given type of computation, and it discards information that cannot be recovered from its output alone, which is precisely what a compression function does to the data it compresses. On this reading the complaint this essay has

been making about C++ and Swift is not that they use state, which would be an absurd complaint against any practical language, but that they use state as though the compression were free and lossless, treating the compressed projection as though it simply were the computation rather than one deliberately chosen view of a richer underlying process. A state-primitive language is not wrong to compute with states; it is incomplete when it forgets that a state is a projection and starts treating the projection as though nothing was lost in producing it, which is exactly the moment aliasing bugs, debugging archaeology, and ad hoc undo systems become necessary rather than optional. The stronger and less ideological version of this essay's thesis, then, is not history good, state bad. It is state is a lossy projection of history, and a language that makes the projection explicit, nameable, and reversible in principle — even where the underlying history genuinely cannot be reversed — gives the programmer a choice that a language treating the projection as the only reality never offers: the choice of when to compress, and the ability to know, later, that a compression already happened.

18 Languages That Remember

Placed on a spectrum by how much of their own execution history they preserve as a matter of course, mainstream languages separate cleanly, and the separation tracks the cost-of-state argument just made rather than contradicting it. C preserves almost none: once a value is overwritten, nothing in the language retains any trace of what it used to be, and any provenance a C program has must be logged by hand. C++ and Swift are mostly state-oriented in the same sense, their added complexity spent almost entirely on managing who may mutate what and when, not on remembering what mutation occurred. Python sits slightly further along the spectrum than either, since its exception tracebacks, its `__dict__` introspection, and its dynamic typing retain more incidental historical information than a compiled state-oriented language typically bothers to, even though this is a side effect of Python's dynamism rather than a designed commitment to history. Git-like version control systems are explicitly historical, treating the commit graph itself as the primary object and any single checked-out working tree as a derived, disposable view onto it — which is exactly a Collapse in the sense defined above, arrived at independently by version-control designers who never used Spherepop's vocabulary but rediscovered its central move regardless [11]. It is worth dwelling on Git specifically, because it is the strongest existence proof available for the argument this essay is making, stronger even than the formal derivation in the appendix. Git is what a history-native system looks like when programmers build one accidentally inside a state-native ecosystem. Nobody disputes that the commit history is primary in Git, and nobody disputes that a working tree is a derived, disposable projection of that history — these are not contested design opinions but the uncontroversial, load-bearing facts every Git user relies on daily, from 'git checkout' to 'git bisect' to 'git reflog' recovering a commit the working tree itself has already forgotten. That Git is universally accepted while a history-primitive programming language sounds exotic is itself evidence for the essay's central claim rather than against it: the same architecture that seems radical when proposed as a foundation for a general-purpose language is already the unquestioned foundation of the single tool most programmers trust most with their own work. Seen this way, Git, event sourcing, database write-ahead logs, Raft's replicated log, and undo trees in editors are not five separate engineering patterns that happen to resemble each other. They are five independent, uncoordinated rediscoveries of the same underlying architecture, arrived at by engineers solving five different problems who were never trying to agree with each other and who mostly do not use the same vocabulary to describe what they built. That convergence, achieved without coordination and under real engineering pressure rather than theoretical preference, is a

far more persuasive argument for history-primacy than any single worked example of arithmetic could be. Event-sourced architectures go further still, making the event log the only source of truth and every materialized view, including the "current state" a client actually queries, an explicitly acknowledged Collapse recomputed on demand [12]. Spherepop sits at the end of this spectrum not because it invented the idea of a history-native system, but because it is the first system on this list that makes history-primacy a property of the language's own core grammar rather than an architecture that has to be built, by hand, on top of a state-primitive language that was never designed to support it.

19 Beyond Arithmetic

Summing positive numbers demonstrated that Spherepop's four primitives can express ordinary arithmetic accumulation, but arithmetic is not where the case for Spherepop is strongest, because every language can do arithmetic and the comparison there is mostly about notation. The stronger cases are the ones where mainstream languages have historically had to invent an entirely new subsystem — a new keyword, a new library, a new architectural pattern — to handle a problem that Spherepop's existing four primitives already cover without modification.

Backtracking search is the clearest case. In Python, C++, or Swift, searching a maze for a path typically requires recursion with explicit state snapshots, an explicit stack simulating the call stack, or a generator maintaining paused execution state, because the language's state-primitive foundation has no native concept of "the path not taken" once a wrong turn has been made and abandoned. In Spherepop, the option space at each junction is simply the set of available directions; choosing one is a Pop, discovering that the chosen direction is blocked is a Refuse, and a successful traversal is not a state reached at the end of the search but a surviving history that was never forced to refuse its way back out. Backtracking is not a separate mechanism requiring its own machinery; it is what reading the recorded history of Pops and Refuses already gives you for free.

Permission and role management, which in most enterprise systems accumulates thousands of lines of special-purpose code, reduces the same way: granting a capability narrows the admissible option space and is a Bind, revoking one is a Refuse, committing a user to a specific role is a Pop, and the permission set currently in effect for an access check is a Collapse of the accumulated grant-and-revoke history. Workflow engines, which invent their own vocabulary of states, transitions, approval chains, and exception routes for every new domain they are applied to, reduce identically: a manuscript moving from draft to review is a Pop, constraints a reviewer imposes are Binds, a rejected submission is a Refuse, and the published document is a Collapse of the entire editorial history that produced it. Version control, the domain already noted above as intuitively history-native, maps almost too cleanly: a branch is a Bind narrowing the admissible future of a line of development, a merge is a Collapse reconciling two divergent histories into one observable tree, a rejected pull request is a Refuse, and an accepted commit is a Pop — which is less a translation exercise than an observation that Git already speaks a dialect of Spherepop's grammar without knowing it. Undo systems follow from the same observation taken to its logical end: where conventional software must build a dedicated undo stack alongside its ordinary state transitions, undo in Spherepop is simply traversing a shorter prefix of a history that was never going to be thrown away regardless, since no separate undo architecture is needed when the architecture you already have is a history. Database transactions, which force relational systems to introduce commits, rollbacks, savepoints, and audit logs as machinery layered on top of the row-level state model, reduce to the same four operators: a transaction is a Bind restricting the admissible future of a set of rows, a commit is a Pop, a rollback is a Refuse, and the row state a subsequent query observes is a Collapse.

Two further examples push the argument outside programming entirely, which is itself evidence that the calculus is doing something more general than translating syntax. Training a machine learning model begins with a large space of admissible parameter configurations; gradient updates progressively Bind that space tighter around configurations consistent with the training signal, a choice of hyperparameters or architecture is a Pop, discarded architectures during a search are Refuses, and the evaluation metric reported at the end is a Collapse of an optimization history that the trained model, as a static set of weights, does not on its own reveal. And the scientific method itself, stripped to its structure, is the same pattern again: a field begins with a space of admissible hypotheses, each failed experiment is a Refuse that eliminates one without confirming the others, surviving hypotheses become progressively Bound by accumulating evidence, acceptance of a theory into the working consensus is a Pop, and the published literature at any given moment is a Collapse of an experimental history that individual papers only ever sample. None of this is offered as proof that Spherepop should replace peer review or gradient descent. It is offered as evidence that conditionals, loops, error handling, backtracking, permissions, workflows, version control, undo, database transactions, and even the epistemics of scientific inquiry are not naturally seven or eight or nine unrelated problems requiring seven or eight or nine unrelated pieces of machinery. Under a history-primary ontology they are one problem, viewed nine times, and the four operators that solve it once solve it every time.

20 What Spherepop Does Not Prove

Intellectual honesty requires stating the limits of this argument as plainly as the argument itself has been stated. Spherepop demonstrates that a wide range of familiar programming and non-programming constructs can be derived from a small history-based calculus, and the worked derivations in the appendix show that this reduction is mechanical rather than merely rhetorical, at least for the cases exhibited. It does not demonstrate that every practical concern of software engineering is best expressed this way. It does not demonstrate that a four-primitive language automatically produces readable code, since readability is a property of notation and tooling as much as of underlying semantics, and Spherepop's notation is at present far less refined than Python's decades of iterative design. It does not demonstrate high performance, since a history that is never discarded is a history that must be stored somewhere, and the coarse-grained history compression sketched earlier remains a design sketch rather than a benchmarked implementation. And it certainly does not demonstrate industrial adoption, which this essay has already conceded is a separate, sociological question that elegance alone has never been sufficient to answer, for Haskell and Lisp any more than for Spherepop. What the argument does establish, and what remains true regardless of whether Spherepop or any of its cousins is ever used to ship production software, is the more modest and more durable claim that state and history are both viable foundations for a theory of computation, that mainstream languages picked the state-primitive foundation for reasons of hardware history rather than conceptual necessity, and that a substantial fraction of the complexity those languages now carry is the accumulated cost of maintaining a foundation that was never obviously the right one to begin with.

21 Objections Worth Taking Seriously

The preceding sections have made the case for Spherepop as forcefully as the argument allows, and intellectual honesty requires giving equal space to the strongest objections that case has drawn,

rather than only the ones easily answered. Five deserve a direct response, and not all of them can be answered in Spherepop's favor.

The first concerns resource cleanup. Mainstream languages handle the pattern of acquiring a resource and guaranteeing its release — a file handle, a lock, a network connection — through mechanisms like `finally`, `defer`, or C++'s RAII, all of which rely on a well-defined moment at which a scope ends and cleanup runs regardless of how that scope was exited. It is a fair objection that the four primitives as presented do not obviously provide this. History in Spherepop is monotonic — Pop, Refuse, and Bind all add to it, never remove from it — which means that if a resource acquisition is recorded and a subsequent Refuse aborts the operation using that resource, the acquisition remains in the history exactly as before, and nothing in the four primitives as stated automatically triggers the corresponding release. A rigorous account would need a further convention, something like a paired acquire-and-release pattern recognized by Collapse when it seals off a region of history, releasing whatever resources that region acquired and did not already release. That convention has not been given a careful treatment here, and until it is, resource cleanup is a genuine gap in the calculus as this essay has presented it, not a problem the four primitives already solve by virtue of being four rather than fifteen. It is worth noting that this gap has the same shape as the problem RAII itself was invented to solve in C++, layered on top of raw pointers that did not handle cleanup either — which suggests the fix, if there is one, looks like a similar disciplined layer built on top of Collapse rather than a revision to the four primitives themselves, but that fix remains future work rather than a settled result.

The second concerns performance, and here the honest answer is closer to a concession than a defense. A naive implementation of Spherepop that genuinely never discards history pays a storage cost proportional to the total number of operations a program has ever performed, where a conventional imperative loop over a large collection uses working memory proportional only to the loop's own state. Iterating over a billion elements and recording a Pop or Refuse for each one is not a minor implementation detail to be waved at with a phrase like coarse-grained history compression; it is the central open engineering question the calculus faces, and no benchmark, prototype measurement, or even a fully specified compression algorithm has been offered in this essay to answer it. The claim this essay can actually support is narrower than a practical performance claim: that the four primitives are conceptually sufficient to express the constructs surveyed, not that any existing implementation of them is competitive with a mainstream compiler's output. Where earlier sections may have let that distinction blur, it should be stated without hedging here: performance is unproven, and until history compression is specified precisely enough to analyze its complexity, no claim about Spherepop's practicality at scale is warranted.

The third concerns readability, and the objection that Spherepop's own primitives are simply a different kitchen sink deserves to be taken more seriously than a quick rebuttal allows. The worked derivation in the appendix is, by any ordinary measure, less readable than the one-line Python equivalent it is compared against, and minimality of primitive count was never, on reflection, the part of the claim doing real work — the lambda calculus has three primitives and nobody argues that this makes it more ergonomic to write in than Python. The claim worth defending is narrower and was under-stated earlier: not that four is fewer than fifteen, but that the same four primitives recur unmodified across loops, conditionals, error handling, permissions, workflows, version control, and backtracking, where mainstream languages introduce a structurally new mechanism for nearly every one of those in turn. Readability of any single example, including the arithmetic one worked through in the appendix, was never the right test of that claim, and presenting it as though it might be invited exactly the objection it drew.

The fourth concerns the tension between this essay's praise of Rust, Haskell, and Lisp for conceptual elegance and its own observation that elegance alone has not produced adoption for

any of them. If conceptual clarity does not reliably translate into practical use, then advocacy for Spherepop cannot honestly be framed as a practical recommendation, and any passage in this essay that reads that way overstates its own case. The more defensible position, consistent with the limitations already stated above, is that clarifying which assumptions are built into mainstream language design has a kind of value that does not depend on anyone adopting an alternative built on different assumptions — formal semantics and category theory shaped how languages are taught and understood long before, and in most cases instead of, shaping which languages got used in production. That is a real contribution, but it is a more modest one than a pitch for adoption, and the essay should be read as making the former rather than implying the latter.

The fifth is the most serious, and it is a genuine open question rather than a rhetorical one: if history-primitive architecture keeps getting rediscovered at the systems level — Git, event sourcing, write-ahead logs, Raft's replicated log — why has it never been adopted as the default foundation of a general-purpose language, and is the earlier explanation offered in this essay, that mainstream languages inherited state-primacy from hardware history rather than conceptual necessity, actually sufficient to answer that? A harder and more honest possibility is that history-primacy is a genuinely higher-level architectural choice that a general-purpose language cannot cheaply provide as a universal default, because most computation within a typical program is not append-heavy the way a commit log or an audit trail is, and a language that pays history-retention cost on every operation regardless of whether that particular operation needs it would be paying, everywhere, a tax that Git and its relatives only pay selectively, at the specific boundary where auditability or recoverability is worth the storage cost. Those systems layer history-primacy on top of an otherwise state-primitive substrate precisely because most of their own internals do not need it either. If that diagnosis is right, Spherepop's four primitives may be answering a real question at the wrong layer of the system: not because history-primacy is false as an ontology, but because making every operation historical by default forces a cost that most operations in most programs would rather not pay, where a selectively adopted history-native subsystem lets a program pay that cost only where it is worth paying. Resolving this would require Spherepop to demonstrate a workable notion of scoped or selective history retention — distinguishing the regions of a program that need to remember from the regions that are safe to forget by default — and to show that this distinction can be drawn without reintroducing the same bookkeeping burden this essay spent its first sections cataloguing in C++. That demonstration does not exist yet. Until it does, this objection stands as the strongest open problem the argument faces, not a solved one, and the essay is more honest for saying so plainly than for treating its own thesis as though the hardware-history explanation had already closed the question.

None of this is an argument that Spherepop should replace Python for data science or Rust for systems programming; it has a handful of prototype implementations and none of Python's library depth, which is exactly the adoption problem already conceded for Haskell and Lisp. Nor is the strongest version of the claim that four primitives are sufficient for practical software engineering at scale — that would be an industrial claim, and industrial claims are settled by adoption, libraries, and decades of production hardening that Spherepop simply does not have. The claim that survives scrutiny is narrower and, precisely because it does not depend on anyone adopting Spherepop, harder to dismiss: many of the constructs that C++ and Swift treat as fundamental and irreducible — the separate syntax for loops versus conditionals versus error handling versus assignment, the several coexisting answers to memory ownership, the folklore of initiation rites required to use any of it safely — are not fundamental at all. They are derived, and they are derivable more economically than either language chooses to derive them, once identity is allowed to be a matter of history rather than of present state. The bloat in C++ and Swift is not a tax levied by the underlying problems of computation; it is committee gravity and vendor gravity doing what gravity does, accumulating

mass around whatever already has mass, one backward-compatible decision at a time. Python earns its popularity by distributing that gravity widely enough that it rarely feels like a tax. Rust, Haskell, and Lisp earn their elegance by refusing the accretion outright, even at real cost to adoption. And Spherepop, whatever its practical limitations, earns its place in this argument by showing that the complexity programmers encounter daily should not be mistaken for evidence about the nature of computation itself. It is evidence, far more often than the field is willing to admit, about the nature of institutional history — and, underneath that, about a choice most languages never admit to having made at all: whether computation is fundamentally the evolution of state or the accumulation of history. C++ and Swift are not being singled out here because they are uniquely bad implementations of the state-primitive choice; in many respects they are highly refined ones. They are being singled out because they are the clearest available evidence of what that choice costs when it is never revisited, decade after decade, one backward-compatible layer at a time. That deeper choice, not the particular syntax of any one language, is what this essay has actually been arguing about, and it will remain worth arguing about long after C++, Swift, Python, and Spherepop alike have been superseded by whatever comes next. It will remain worth arguing about, in particular, whether that choice belongs at the level of a general-purpose language at all, or only at the level of the specific systems — Git, event sourcing, consensus logs — that keep rediscovering it under real pressure; the previous section left that question open rather than resolved, and it is the honest place to leave it.

A A Formal Derivation of the Worked Example

The comparison in the body of the essay between Python, C++, Swift, and Spherepop was offered as a sketch, and a formalist is right to want more than a sketch before accepting that "loop equals repeated Bind and Pop" is a derivation rather than an analogy. This appendix supplies the derivation, using the core Spherepop grammar and typing discipline in which a state is a pair of an option space and a history, $\mathsf{State} \cong (\mathrm{Opt}, \mathrm{Hist})$, and the four term-forming operators are typed as follows.

$$\frac{}{\Gamma \vdash (\mathrm{Opt}_t, H_t) : \mathsf{State}} \quad \text{(state introduction)}$$

$$\frac{\Gamma \vdash t : \mathsf{State}}{\Gamma \vdash \mathrm{Bind}_f(t) : \mathsf{State}} \quad \text{(bind)}$$

$$\frac{\Gamma \vdash t : \mathsf{State} \qquad a \in \mathrm{Opt}_t}{\Gamma \vdash \mathrm{Pop}_a(t) : \mathsf{State}} \quad \text{(pop)}$$

$$\frac{\Gamma \vdash t : \mathsf{State} \qquad a \in \mathrm{Opt}_t}{\Gamma \vdash \mathrm{Refuse}_a(t) : \mathsf{State}} \quad \text{(refuse)}$$

$$\frac{\Gamma \vdash t : \mathsf{State}}{\Gamma \vdash \mathrm{Collapse}(t) : \mathsf{Obs}} \quad \text{(collapse)}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{State} \qquad \Gamma \vdash t_2 : \mathsf{State}}{\Gamma \vdash t_1 ; t_2 : \mathsf{State}} \quad \text{(sequence)}$$

$\mathrm{Bind}_f(t)$ applies an admissibility filter f to the option-space component of the state denoted by t, replacing Opt_t with $\{\, o \in \mathrm{Opt}_t : f(o) \,\}$. $\mathrm{Pop}_a(t)$ removes the admissible option a from the residual option space and appends it to the history, tagged as *selected*. $\mathrm{Refuse}_a(t)$ performs the identical structural operation but tags a in the history as *rejected* rather than selected — geometrically the

same morphism as Pop, differing only in the label carried forward into the history. Collapse(t) is the reconstruction step: it reads the accumulated history H_t and reduces it to an observable value under a collapse rule fixed by the type of the computation being performed. The core grammar leaves that collapse rule uninstantiated in general, since different programs collapse their histories in different ways; for the arithmetic accumulation used in the worked example, the collapse rule is fixed as summation over every atom in the history tagged *selected*, ignoring atoms tagged *rejected*. This instantiation is stated explicitly here because it is a modeling choice at the application layer, not a fact already fixed by the four primitives themselves — the same four primitives could equally instantiate Collapse as a product, a concatenation, or a maximum, depending on what the surrounding program requires.

The program. Let *numbers* $= [3, -2, 5]$, and let the task be to compute the sum of the positive entries, so that the expected observable result is 8. Define the admissibility filter used at each step as $f_{\text{pos}}(o) = \big(\text{tag}(o) = \text{include} \Rightarrow \text{value}(o) > 0\big)$, read as: the option to include an element is admissible only when that element is positive; the option to exclude an element is always admissible regardless of sign.

Initial state. $S_0 = (\emptyset, \varepsilon)$, the empty option space and the empty history, typed $\vdash S_0 : \mathsf{State}$ by the state-introduction rule.

Step 1, element 3. The option space is opened to the binary choice for this element, Opt $= \{\text{include}_1, \text{exclude}_1\}$, and $\text{Bind}_{f_{\text{pos}}}$ is applied: since $3 > 0$, both options remain admissible after filtering, $\text{Opt}' = \{\text{include}_1, \text{exclude}_1\}$. The program then commits with $\text{Pop}_{\text{include}_1}$, which is admissible since $\text{include}_1 \in \text{Opt}'$, yielding $S_1 = (\{\text{exclude}_1\}, [\text{include}_1])$ — the residual option space still formally contains the unchosen alternative, and the history now records one selected atom carrying the value 3. Typing: $\vdash \text{Bind}_{f_{\text{pos}}}(S_0 \oplus \text{Opt}) : \mathsf{State}$ and $\vdash \text{Pop}_{\text{include}_1}(\text{Bind}_{f_{\text{pos}}}(\ldots)) : \mathsf{State}$, both by the bind and pop rules above.

Step 2, element -2. Opt $= \{\text{include}_2, \text{exclude}_2\}$. Applying $\text{Bind}_{f_{\text{pos}}}$: since $-2 > 0$ is false, include_2 is filtered out, leaving $\text{Opt}' = \{\text{exclude}_2\}$ — the admissibility filter has already done the work that an `if` statement would do in Python, C++, or Swift, before any commitment operator runs. Because include_2 is now inadmissible, the program cannot $\text{Pop}_{\text{include}_2}$; instead it commits with $\text{Refuse}_{\text{include}_2}$, which is well-typed under the refuse rule using the *original* unfiltered option space (refusal, unlike selection, is precisely the operator that records the elimination of an option that failed admissibility, so it is typed against Opt before filtering, not Opt′ after). This yields $S_2 = (\{\text{exclude}_2\}, [\text{include}_1, \text{refuse}_2])$, the history now carrying one selected atom and one rejected atom.

Step 3, element 5. As in step 1, $\text{Bind}_{f_{\text{pos}}}$ leaves both options admissible since $5 > 0$, and the program commits with $\text{Pop}_{\text{include}_3}$, yielding $S_3 = (\{\text{exclude}_3\}, [\text{include}_1, \text{refuse}_2, \text{include}_3])$.

Collapse. Collapse(S_3) reads the history $[\text{include}_1, \text{refuse}_2, \text{include}_3]$, discards the atom tagged *rejected*, and reduces the remaining selected atoms — carrying values 3 and 5 — under the fixed summation rule for this program, producing Obs $= 8$. Typed: $\vdash \text{Collapse}(S_3) : \mathsf{Obs}$, by the collapse rule, with Obs $= 8$ as required.

The full term. Written as a single sequenced composition using the grammar's sequence operator, the entire program is

$$\text{Bind}_{f_{\text{pos}}}(S_0)\ ;\ \text{Pop}_{\text{include}_1}\ ;\ \text{Bind}_{f_{\text{pos}}}\ ;\ \text{Refuse}_{\text{include}_2}\ ;\ \text{Bind}_{f_{\text{pos}}}\ ;\ \text{Pop}_{\text{include}_3}\ ;\ \text{Collapse}$$

six primitive term applications plus one collapse, every one of them an instance of exactly four operators, none of them a special-cased control-flow keyword. Compare this to the C++ version of the same program, which requires the programmer to select among at least three structurally different idioms — a hand-written loop with a mutable accumulator and an `if` guard, `std::accumulate` composed with a filtering predicate passed as a lambda, or a C++20 ranges pipeline chaining `views::filter` and `views::transform` before a final `ranges::fold_left` — each idiom carrying its own rules for how the accumulator is captured, whether the lambda closes over its environment by reference or by value, and whether the container is consumed, copied, or borrowed. The Spherepop derivation above requires no such choice, because filtering, selection, rejection, and accumulation are not four different mechanisms borrowed from four different corners of the language; they are four applications of the same four typed operators, composed the only way the grammar allows them to compose.

A caveat on canonicity. What this derivation establishes is that the reduction from the informal program to a well-typed Spherepop term exists, is mechanical, and produces the correct observable result — the earlier claim that the mapping is exact rather than metaphorical is now discharged for this example rather than merely asserted. What it does not establish, and what a fuller formal treatment would still need to address, is uniqueness: nothing here rules out a second, differently structured Spherepop term that also reduces to Obs $= 8$ from the same input, for instance one that filters via Refuse on every element uniformly and reserves Pop for a separate accumulation pass. The claim this appendix defends is narrower than global canonicity — that C++ and Swift require *multiple incompatible idiom families* to express what Spherepop expresses with *one family of four operators applied uniformly* — and that narrower claim is what the derivation above actually proves, independent of whether some other valid Spherepop derivation of the same result also exists.

References

[1] Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[2] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[3] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[4] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000. (Lecture notes originally circulated 1967.)

[5] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.

[6] Alan C. Kay. The Early History of Smalltalk. *ACM SIGPLAN Notices*, 28(3):69–95, 1993.

[7] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[8] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[9] Rolf Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[10] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, 2014.

[11] Scott Chacon and Ben Straub. *Pro Git*, 2nd edition. Apress, 2014.

[12] Martin Fowler. Event Sourcing. *martinfowler.com*, 2005.

[13] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018.

[14] International Organization for Standardization. ISO/IEC 14882:2020, Programming Languages — C++. ISO, 2020.