The Calculus of Commitment

A Structural Theory of Computation

Flyxion

February 22, 2026

Abstract

This text develops Spherepop as a minimal structural calculus of merge and collapse, and then uses it as a unifying lens through which classical models of computation may be understood. Rather than beginning with the λ-calculus, typed judgments, or machine architectures, we begin with a primitive algebra of regions. From this foundation we reconstruct higher-order functions, type structure, stack-based execution, and functional evaluation as disciplined refinements of a simpler merge–collapse substrate.

The central thesis is that computational formalisms often presented as independent traditions can be seen as stratifications of a single structural principle: the controlled reduction of optionality under compositional constraint. Spherepop serves as the pedagogical base layer. From it we derive λ-abstraction as region-valued transformation, type theory as invariant preservation under composition, stack-based programming as explicit sequencing of region states, and functional programming as replayable history of pure transformations.

The goal of the text is not to replace established theories, but to provide a structural perspective that makes their common invariants visible.

Preface

This book develops a structural account of computation from a small set of elementary operations. Rather than beginning with machines, syntax, or programming languages, we begin with a simple algebra over regions and equivalence. From that base, familiar formalisms—λ-calculus, type systems, currying, pipelines, monads, algebraic effects, state, and continuation-passing style—are reconstructed as layered refinements of the same underlying structure.

The purpose of this reconstruction is not to replace established theories. The classical results of reduction theory, denotational semantics, and categorical logic remain intact. What changes is the vantage point. By isolating a minimal pair of operations—one that accumulates structure and one that resolves it—many seemingly distinct paradigms can be seen to share a common algebraic backbone. Abstraction becomes controlled identification, composition becomes associative accumulation, and evaluation becomes canonical normalization.

The development proceeds incrementally. We begin with the bare calculus. We then derive abstraction and composition, followed by typing and structured effects. Later chapters examine state, continuation, and mutation as disciplined variations in where and when structural resolution is permitted. Throughout, emphasis is placed on algebraic stability: associativity, identity, and invariance under canonical form.

The formal framework used to organize this presentation is referred to as *Spherepop*. It is not proposed as an alternative model of computability, nor as a replacement for established formalisms. Rather, it serves as a compact vehicle for articulating the structural commitments that classical systems already presuppose but do not always foreground. Its purpose is expository and unifying: to make visible the algebraic backbone shared across otherwise diverse computational paradigms.

Viewed from this perspective, computation is not exhausted by symbol manipulation. It consists in the disciplined introduction, organization, and resolution of commitments within a formal system. The calculus developed in these chapters aims to render that discipline explicit, so that familiar constructions appear not as isolated techniques, but as expressions of a common structural logic.

1 Motivation: Structure Before Symbol

Computation is often introduced syntactically. One begins with symbols, variables, application, reduction rules, and machine models. Only later does one ask what structural invariants those systems preserve.

This text reverses that order.

We begin not with variables or functions, but with *regions*. A region is a finite structural commitment: a collection of distinguishable elements subject to identification. Computation, at its most primitive level, is the controlled evolution of such regions under two operations:

Merge and **Collapse**.

Merge combines commitments. Collapse identifies them.

Everything that follows—abstraction, substitution, typing, stack discipline, and pure functional evaluation—can be understood as increasingly refined disciplines governing when and how merge and collapse may occur.

The guiding philosophical principle is this:

> Structure precedes symbol. Symbol manipulates structure.

By grounding computation in structural commitments rather than symbolic rewriting alone, we expose the invariants common to otherwise distinct computational paradigms.

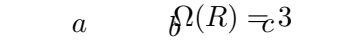

Figure 1: A region as a finite set of distinguishable atoms, with optionality measured by cardinality.

2 Historical Development: From Symbolic Computation to Structural Algebra

The development of computation theory has often been narrated as a sequence of formal breakthroughs: recursive functions, λ-calculus, Turing machines, domain theory, type systems, functional languages, and categorical semantics. Yet beneath these advances lies a quieter continuity: the gradual recognition that computation is structural rather than merely symbolic.

This section traces that development and situates the merge–collapse viewpoint within it.

2.1 The Foundational Era: Symbolic Computability

Church and Turing formalized the notion of effective procedure in the 1930s. The λ-calculus introduced abstraction and substitution as primitive operations, while Turing machines rendered computation as state transition.

Despite their apparent differences, both models reduced computation to structured transformation under explicit rules.

The Church–Turing thesis unified these models, suggesting that computation is invariant under representation.

2.2 Reduction and Confluence

The λ-calculus emphasized reduction. β-reduction formalized substitution, while later work established confluence and normalization properties.

These results revealed a deep invariant: evaluation order may vary, but canonical results remain stable when reduction is well-behaved.

This stability anticipates the role of canonical collapse in merge–collapse semantics.

2.3 Domain Theory and Denotational Semantics

In the 1970s, Scott introduced domain theory to give mathematical meaning to recursive definitions. Functions were no longer mere symbolic rewrites, but continuous maps over structured spaces.

Denotational semantics reframed programs as mathematical objects. This shift from operational step-by-step reasoning to structural interpretation marked a profound turn.

Computation became geometry.

2.4 The Rise of Type Theory

Type systems emerged to control abstraction. Martin-Löf's intuitionistic type theory and Reynolds' parametric polymorphism formalized invariants preserved under transformation.

Typing was no longer annotation, but a structural guarantee: certain collapses are permitted, others forbidden.

In this light, types are disciplined quotient structures.

2.5 Functional Programming as Algebra

Landin and Strachey recognized that programming languages could be understood through the lens of λ-calculus. Backus later called for liberation from the von Neumann style, arguing for algebraic composition over mutable state.

The work of Meijer, Wadler, Moggi, and others made this algebraic view explicit. Catamorphisms, monads, and folds reframed programs as structured transformations over data.

Recursion schemes made collapse systematic. Algebra replaced control flow.

2.6 Category Theory and Internal Structure

Category theory unified many strands. Monoidal structure, internal hom, and adjunctions provided a language for abstraction itself.

The λ-calculus became the internal language of Cartesian closed categories. Programs were morphisms. Types were objects. Composition was structural.

This categorical turn revealed that abstraction is not a syntactic trick, but a reflection of deeper algebraic structure.

2.7 From Algebra to Structural Reduction

Spherepop continues this trajectory, but begins one level lower.

Rather than starting from functions, types, or morphisms, it begins from two primitive structural operations:

$$\text{merge} \quad \text{and} \quad \text{collapse}.$$

Where λ-calculus begins with abstraction, Spherepop asks what abstraction must be at the level of raw structural commitment.

Where type theory imposes invariants, Spherepop interprets types as disciplined quotient structures.

Where functional programming composes pure transformations, Spherepop frames composition as associative merge followed by canonical collapse.

In this sense, Spherepop does not reject classical formalisms. It seeks to expose the minimal structural substrate from which they arise.

2.8 A Structural Continuum

The historical arc may therefore be summarized as:

$$\text{Symbolic Procedure} \longrightarrow \text{Reduction Theory} \longrightarrow \text{Denotational Geometry} \longrightarrow \text{Type Invariants} \longrightarrow \text{Algebraic Composition} \longrightarrow \text{Structural Merge–Collapse}$$

Each stage refines the same insight: computation is structured transformation under invariant-preserving reduction.

Spherepop makes that structure explicit at its base layer.

The remainder of this text develops that layer and reconstructs the higher strata from it.

2.9 Algebraic Recursion Schemes

A particularly important moment in the algebraic understanding of computation came with the explicit treatment of recursion as a structural scheme rather than as ad hoc self-reference.

Work by Meijer, Fokkinga, and Paterson formalized recursion patterns as catamorphisms, anamorphisms, and hylomorphisms. These recursion schemes made precise what had long been implicit: data types carry algebraic structure, and programs over them are homomorphisms.

A fold is not merely a loop. It is a collapse into an algebra.

In merge–collapse language, a catamorphism is precisely a disciplined collapse of a recursively generated region into a canonical form. An anamorphism, dually, is structured merge.

This algebraic viewpoint aligns directly with the thesis of this text: computation is not instruction sequencing, but structured transformation of commitment.

The importance of this development is methodological. Rather than writing programs and then proving properties, one calculates programs from algebraic laws. Structure precedes implementation.

Spherepop inherits this orientation. Its primitive operations are not operational instructions, but algebraic transformations. Higher-order abstraction and typed invariants emerge from these transformations, rather than being imposed externally.

2.10 Spherepop's Structural Commitments

It is important to clarify how Spherepop differs from classical computational formalisms.

First, it does not introduce a new notion of computability. The Church–Turing thesis remains intact. Spherepop is not a competing model of effective procedure.

Second, it does not replace λ-calculus, type theory, or functional programming. Rather, it attempts to identify the minimal structural substrate common to them.

The distinctive commitments of Spherepop are:

1. Structural Primacy. Computation is defined in terms of merge and collapse, prior to variables, substitution, or typing.

2. Quotient-Centric Abstraction. Abstraction is interpreted as disciplined introduction of equivalence relations, not merely syntactic binding.

3. Canonical Normalization. Collapse is canonical projection. Confluence and referential transparency are consequences of this property.

4. Separation of Authority and View. Operational histories (event logs) are distinguished from derived semantic views (regions). This mirrors the distinction between evaluation trace and denotational meaning.

5. Optionality as a Structural Measure. The cardinality of a region measures degrees of freedom. Computation reorganizes optionality under constraint.

These commitments yield a perspective in which λ-calculus appears as disciplined merge–collapse, type theory appears as invariant enforcement, and functional programming appears as replayable structural history.

Spherepop is therefore best understood not as a new calculus, but as a structural lens through which existing calculi may be reinterpreted.

In the sections that follow, we return to the formal development, now situated within this historical and conceptual frame.

2.11 Substitution and Quotient: A Structural Comparison with λ-Calculus

The classical λ-calculus is built upon substitution. Application replaces a variable with a term, yielding β-reduction.

From the merge–collapse perspective, substitution is not primitive. It is a consequence of quotient introduction.

In λ-calculus:

$$(\lambda x.\, e)\ a \ \rightarrow\ e[x := a].$$

In merge–collapse semantics:

$$\text{collapse}\left(e \oplus \{a\}\right) \quad \text{under } x \sim a.$$

The essential difference is conceptual. Substitution operates at the level of syntax: it rewrites expressions by replacing bound variables with concrete terms. Its action is textual, proceeding by systematic replacement within a symbolic representation. Quotient, by contrast, operates at the level of structure. It introduces an identification within the governing equivalence relation, thereby collapsing distinctions canonically rather than rewriting symbols.

Substitution can thus be understood as a syntactic mechanism for achieving what quotient accomplishes structurally. Where substitution replaces occurrences of a name, quotient enforces identity between the structures those names denote. The former manipulates expressions; the latter reorganizes commitments.

This distinction clarifies several phenomena.

First, variable capture becomes a non-issue, since identification occurs at the structural level, not through name replacement.

Second, confluence arises from canonical collapse, not merely from syntactic reduction properties.

Third, abstraction becomes the promise of future identification, rather than the binding of a placeholder.

The λ-calculus can therefore be viewed as a symbolic surface representation of deeper quotient dynamics.

2.12 Spherepop and the Internal Language of Categories

Category theory offers a unifying framework for abstraction and composition by treating programs as morphisms and types as objects within a structured system. In particular, a Cartesian closed category supports a monoidal product, internal hom objects, and the operations of evaluation and currying. Within such a setting, the λ-calculus arises naturally as the internal language: abstraction corresponds to the formation of exponential objects, and application corresponds to evaluation.

From the merge–collapse perspective, these categorical constructions admit a structural interpretation. Merge supplies the monoidal structure, providing an associative means of combining regions with an identity element. Collapse functions as canonical projection, ensuring that structural identifications yield stable representatives. Abstraction may then be viewed as the formation of an internal hom object, organizing transformations as first-class regions, while application corresponds to the evaluation morphism that realizes these transformations within a composed structure.

On this reading, categorical semantics does not introduce a fundamentally new layer, but makes explicit the algebraic invariants already present in the merge–collapse substrate. The internal language of a Cartesian closed category can thus be understood as a disciplined refinement of the same structural principles.

More precisely, if $(\mathcal{R}, \oplus, \varnothing)$ denotes regions under merge, then abstraction constructs an object $[A \Rightarrow B]$ such that:

$$A \oplus [A \Rightarrow B] \longrightarrow B.$$

Collapse ensures that this morphism is well-defined under equivalence.

Thus the structural content of a Cartesian closed category is already present in minimal form within merge–collapse semantics.

The categorical perspective does not introduce additional machinery so much as it clarifies the invariants already implicit in the structural calculus. The associativity of merge corresponds to the associativity of the monoidal product. The existence of an empty region as identity aligns with the categorical unit. Stability under canonical projection mirrors the requirement that morphisms respect the equivalence structure governing objects. These properties are not imposed from outside; they are conditions under which composition remains coherent and abstraction remains well-defined.

In this sense, the merge–collapse framework may be regarded as a pre-categorical substrate. When its operations are disciplined and organized according to these invariants, the familiar structures of

categorical semantics emerge naturally. Cartesian closure, internal hom formation, and evaluation morphisms can then be understood as systematic refinements of an already structured algebra, rather than as externally introduced abstractions.

2.13 A Historical Case Study: Deriving Folds from Merge–Collapse

We now illustrate how a classical recursion scheme arises naturally from merge–collapse dynamics.

Consider a recursively generated region representing a list:

$$R = \{x_1, x_2, \ldots, x_n\}.$$

A fold over this structure combines elements using an algebra:

$$\texttt{fold}\ f\ z\ [x_1, \ldots, x_n].$$

In structural terms, the recursive data structure is built by repeated merge:

$$R = \{x_1\} \oplus \{x_2\} \oplus \cdots \oplus \{x_n\}.$$

A fold introduces a collapse governed by algebra f.

Each merge increases optionality. The fold collapses accumulated structure into a canonical result.

Thus:

$$\text{fold} \quad = \quad \text{structured collapse over recursive merge.}$$

This derivation shows that data construction corresponds to merge, while evaluation corresponds to collapse. Recursion schemes, when viewed structurally, amount to disciplined iteration of these two operations. What appears in the algebraic programming tradition as a categorical construction—most notably in the presentation of folds as catamorphisms—can therefore be understood as an organized pattern of accumulation followed by canonical reduction.

From this vantage point, recursion schemes are not external refinements imposed upon an otherwise neutral calculus. They arise naturally once merge and collapse are taken as primitive. The categorical formulation makes their algebraic properties precise, but the structural necessity of their form is already present at the substrate level.

This case study thus reinforces the broader claim developed throughout the text: the abstractions of functional programming are not arbitrary design choices, but systematic refinements of a more elementary structural reduction. The framework employed here does not seek to displace established formalisms. It aims instead to expose their minimal generative core and to clarify the invariants

that sustain them.

3 Regions and Optionality

We formalize the primitive semantic object.

Definition 1 (Region)**.** *A region R is a finite set of atomic names modulo an equivalence relation.*

Concretely, a region may be represented as a finite set

$$R = \{a_1, a_2, \ldots, a_n\},$$

together with a quotient structure identifying certain elements.

The cardinality $|R|$ measures available distinguishable commitments. We refer to this as *optionality*.

Definition 2 (Optionality)**.** *For a region R, define*

$$\Omega(R) := |R|.$$

Optionality measures the degrees of freedom present in a region. Computation reduces, restructures, and composes optionality.

4 The Merge Operation

Merge is structural composition.

Definition 3 (Merge)**.** *Given regions R_1 and R_2, their merge is*

$$R_1 \oplus R_2 := \mathrm{collapse}(R_1 \cup R_2).$$

Intuitively, merge forms the union of commitments and then normalizes under the existing quotient.

Merge is associative up to canonical normalization.

Proposition 1. *Merge is associative modulo representative equivalence.*

Proof. Set union is associative. Collapse is a canonical projection. Hence the composite operation is associative up to normalization. □

This already resembles monoidal composition.

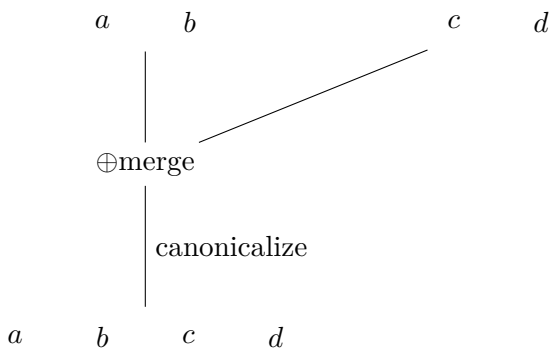

Figure 2: Merge forms union followed by canonical normalization.

5 The Collapse Operation

Collapse performs identification.

Definition 4 (Collapse)**.** *Let $\sim$ be an equivalence relation over atomic names. Collapse maps a region R to its quotient*

$$\text{collapse}(R) := R/\sim.$$

Collapse reduces optionality when identifications are nontrivial.

Proposition 2. *If k identifications are introduced, then*

$$\Omega(\text{collapse}(R)) \leq \Omega(R).$$

Collapse corresponds to substitution, unification, or evaluation in more familiar calculi.

6 Programs as Region Transformations

A *program* in Spherepop is an expression built from:

$$\text{Atom}, \quad \oplus, \quad \text{collapse}, \quad \text{LetQuotient}.$$

Its denotation is a region.

Evaluation is deterministic under a fixed quotient context. This replayability will later correspond to referential transparency in functional programming and confluence in λ-calculus.

7 From Regions to Functions

We now observe a crucial structural step.

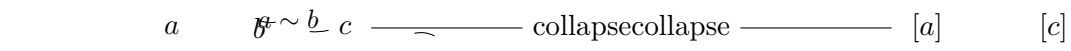

Figure 3: Collapse identifies atoms under an equivalence relation and projects to canonical representatives.

A function

$$f : R \to R$$

is simply a rule that merges input with additional structure and possibly collapses the result.

Abstraction therefore corresponds to:

$$\lambda x.\, e \quad \Longleftrightarrow \quad \text{a region transformer parameterized by } x.$$

Application corresponds to merge followed by collapse under substitution.

Thus, the λ-calculus can be reconstructed as a disciplined fragment of merge–collapse semantics.

In the next section, we formalize this reconstruction.

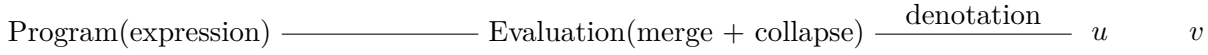

Figure 4: A program denotes a canonical region via deterministic evaluation.

8 Axiomatization of the Merge–Collapse Calculus

We now make explicit the algebraic laws implicitly assumed throughout the previous development. The goal is not to over-formalize, but to clarify the minimal structural commitments required.

8.1 Primitive Objects

We begin by fixing a countable set of atomic names $\mathcal{A}$. These atoms serve as the minimal distinguishable commitments within the system. A *region* is defined to be a finite subset $R \subseteq \mathcal{A}$, representing a finite collection of such commitments.

In addition, we assume the presence of an equivalence relation $\sim$ over $\mathcal{A}$. This relation governs which distinctions are to be regarded as structurally irrelevant. All regions are interpreted modulo this equivalence: two regions are considered identical when their elements coincide up to the identifications induced by $\sim$. In this way, equivalence relations provide the formal mechanism by which collapse operates over accumulated structure.

8.2 Primitive Operations

Two operations are primitive:

Merge.

$$\oplus : \mathcal{R} \times \mathcal{R} \to \mathcal{R}.$$

Collapse.

$$\text{collapse}_{\sim} : \mathcal{R} \to \mathcal{R}/\sim .$$

8.3 Structural Laws

The following axioms govern merge and collapse.

(M1) Associativity.

$$(R_1 \oplus R_2) \oplus R_3 = R_1 \oplus (R_2 \oplus R_3).$$

(M2) Identity. There exists an empty region $\varnothing$ such that

$$R \oplus \varnothing = R.$$

(M3) Commutativity (optional). If order is not semantically significant:

$$R_1 \oplus R_2 = R_2 \oplus R_1.$$

(C1) Idempotence of Collapse.

$$\text{collapse}_{\sim}(\text{collapse}_{\sim}(R)) = \text{collapse}_{\sim}(R).$$

(C2) Canonical Projection. Collapse yields a unique representative region for a given equivalence class.

(C3) Stability under Merge.

$$\text{collapse}_{\sim}(R_1 \oplus R_2) = \text{collapse}_{\sim}(\text{collapse}_{\sim}(R_1) \oplus \text{collapse}_{\sim}(R_2)).$$

This law ensures evaluation order independence.

8.4 Derived Laws

From the axioms governing merge and collapse, several important consequences follow. First, normalization is confluent: distinct sequences of merge operations, when followed by collapse under a fixed equivalence relation $\sim$, yield canonically identical regions. This ensures that evaluation order does not affect denotation.

Second, denotation is deterministic once $\sim$ is fixed. Because collapse produces a unique representative for each equivalence class, the meaning of a region depends only on its accumulated commitments and the governing identifications, not on the history by which it was assembled.

Finally, the operations remain compatible with monoidal structure. Merge is associative with an identity element, and collapse respects this structure by acting as a projection that preserves equivalence classes under composition.

These derived properties are precisely those required for functional semantics to remain stable and for type-theoretic invariants to be preserved under transformation. They do not arise from additional constraints; they are structural consequences of the axioms already in place.

8.5 Abstraction as Quotient Introduction

Abstraction is formalized as controlled extension of the equivalence relation.

Given $x \in \mathcal{A}$ and region R, application introduces $x \sim a$ for some $a \in R$, and collapse enforces the identification.

Thus abstraction is not primitive. It is a derived operation: quotient introduction followed by canonical projection.

9 A Thermodynamic Interpretation of Optionality and Abstraction

The merge–collapse calculus admits a natural thermodynamic metaphor. This metaphor is not essential, but it clarifies the directionality of computation.

9.1 Optionality as Degrees of Freedom

Define optionality:

$$\Omega(R) = |R|.$$

Optionality measures distinguishable commitments. It corresponds to degrees of freedom.

Merge tends to increase optionality. Collapse tends to reduce it.

9.2 Commitment and Work

When two regions merge:

$$R' = R_1 \oplus R_2,$$

new structural commitments are introduced. This resembles work input: additional distinctions must be maintained.

When collapse identifies elements:

$$R'' = \text{collapse}_{\sim}(R'),$$

distinctions are removed. This resembles compression.

9.3 Irreversibility and Canonical Form

Collapse is idempotent and lossy: once two atoms are identified, the distinction cannot be recovered without additional structure.

Thus collapse introduces irreversibility.

In functional programming, evaluation similarly moves toward canonical form. Normalization is entropy reduction with respect to representational redundancy.

9.4 Abstraction as Entropy Reduction

Abstraction removes accidental distinctions. If a region R has optionality $\Omega(R)$, and abstraction introduces identifications $\sim$, then:

$$\Omega(\text{collapse}_{\sim}(R)) \leq \Omega(R).$$

This reduction corresponds to compression.

However, abstraction does not eliminate structure. It preserves invariant relationships while discarding irrelevant detail.

9.5 Structured Equilibrium

A fully collapsed region under fixed equivalence represents structural equilibrium. Further collapse has no effect.

Similarly, a normalized functional term is in normal form.

The merge–collapse dynamic therefore mirrors:

$$\text{Expansion (merge)} \longrightarrow \text{Constraint (quotient)} \longrightarrow \text{Equilibrium (canonical collapse)}.$$

9.6 Computation as Controlled Thermodynamic Flow

Computation can thus be seen as controlled redistribution of optionality.

Pure functional evaluation preserves total information within a closed system, but reorganizes it into canonical structure.

Side effects correspond to untracked merge operations. Mutation corresponds to uncontrolled collapse.

Spherepop enforces explicit merge and disciplined collapse, making structural flow transparent.

9.7 Limits of the Metaphor

The thermodynamic analogy developed above should not be pressed beyond its scope. Optionality, as defined in this calculus, is a combinatorial measure over regions; it does not correspond to physical entropy, nor does collapse model physical dissipation. The analogy is heuristic rather than literal.

Nevertheless, it remains instructive. Merge increases structural differentiation, expanding the space of explicit commitments. Collapse reduces differentiation by enforcing identifications under an equivalence relation. Abstraction functions as a controlled form of compression, selecting which distinctions are to be retained and which are to be rendered irrelevant. Canonical form represents a stable configuration under repeated collapse, analogous to equilibrium under a fixed constraint regime.

Under this interpretation, computation can be viewed as the disciplined reorganization of structural freedom. It proceeds not by arbitrary transformation, but by accumulation followed by invariant-preserving reduction. The thermodynamic metaphor serves only to illuminate this dynamic, not to replace the formal account that grounds it.

10 Currying and Composition: From λ-Calculus to Unix Pipes

Currying transforms a function of multiple arguments into a sequence of single-argument functions.

In classical notation:

$$f : A \times B \to C$$

becomes

$$f : A \to (B \to C).$$

This transformation is not merely syntactic. It reveals that multi-argument functions are iterated region transformers.

10.1 Currying as Iterated Merge

In merge–collapse semantics, a function is a region transformer:

$$f(R) = \text{collapse}(R \oplus S_f).$$

If f depends on two inputs, we may write:

$$f(R_1, R_2) = \text{collapse}(R_1 \oplus R_2 \oplus S_f).$$

Currying reorganizes this into:

$$f(R_1)(R_2) = \text{collapse}(R_2 \oplus \text{collapse}(R_1 \oplus S_f)).$$

Thus currying stages merge. The first application partially commits structure. The second completes it.

10.2 Partial Application as Structural Commitment

Consider a function

$$\texttt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}.$$

Applying one argument:

```
add 2
```

does not compute a number. It produces a new transformer:

$$x \mapsto 2 + x.$$

Structurally, this corresponds to merging the commitment for 2 into the function region, while leaving the second parameter uncollapsed.

Partial application is therefore controlled early merge.

10.3 Unix Pipes as Sequential Composition

Unix pipelines provide a useful operational analogy for sequential composition. Consider the command

```
cat file.txt | grep pattern | sort
```

Each component in this pipeline consumes a stream, produces a transformed stream, and passes its output to the next stage. The composition is linear and associative: the output of one transformation becomes the input of the next, and the grouping of stages does not affect the final result so long as their order is preserved.

Currying expresses the same compositional structure within the λ-calculus. By reducing multi-argument functions to sequences of unary transformations, currying enables functions to be chained in a manner analogous to pipelines. Each application produces an intermediate result that serves as the input to the subsequent stage. The pipe operator in Unix and function composition in the λ-calculus thus reflect the same structural principle: sequential accumulation of transformation under associative composition.

Instead of:

$$f(x, y, z),$$

we write:

$$f(x)(y)(z).$$

Each stage produces an intermediate transformer, analogous to a stream flowing through a pipe.

10.4 Pipes as Region Flow

Let S represent a region encoding a data stream. A pipeline:

$$h \circ g \circ f$$

corresponds structurally to:

$$\text{collapse}(S \oplus S_f \oplus S_g \oplus S_h).$$

Associativity of merge ensures that grouping is irrelevant:

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

This mirrors the associativity of pipe composition.

10.5 Why Currying Matters

Currying is significant because it reveals several structural features that are otherwise obscured in multi-argument notation. First, it makes explicit that functions are first-class values: partial application produces new functions rather than immediate results. Second, it highlights the associativity of composition, since multi-stage application reduces uniformly to chaining unary transformations. Third, it allows partial structure to be committed incrementally, with each application contributing to the eventual result without requiring all inputs at once.

In stack-based or imperative settings, composition typically depends on explicit control flow constructs that manage sequencing and intermediate state. In a curried functional setting, by contrast, composition is structural. The chaining of functions follows directly from the algebra of application itself, without recourse to additional control mechanisms. Currying thus clarifies that compositionality is not an external convenience, but an intrinsic property of the underlying calculus.

10.6 Currying as Structural Closure

Currying reveals a deeper invariant: multi-input computation can be systematically reduced to unary transformation over structured input. By transforming functions of several arguments into chains of single-argument functions, currying imposes a uniform discipline on application. Every stage of computation consumes exactly one input and produces either a final value or another unary transformer.

This reduction simplifies reasoning by isolating each commitment. Inputs are incorporated incrementally, merge operations are staged one at a time, and collapse is applied only when sufficient

structure has accumulated. The complexity of multi-argument interaction is thus decomposed into a sequence of elementary steps governed by the same algebraic laws.

Unix pipelines render this staged transformation visually intuitive: each command accepts a stream and emits a transformed stream. Currying provides the formal analogue. It converts this intuitive pattern into a mathematically precise principle of structural closure under unary composition.

10.7 Merge–Collapse Interpretation of Pipelines

Let each program P_i correspond to a structural commitment S_i. Then a pipeline is:

$$S \mapsto \text{collapse}(S \oplus S_1 \oplus S_2 \oplus \cdots \oplus S_n).$$

The pipe symbol | becomes an implicit merge, and evaluation corresponds to final collapse.

Thus currying and Unix pipelines share the same algebraic backbone: sequential composition via associative merge and canonical projection.

11 Summary

Currying exposes a structural unity across several forms of computation. Multi-argument functions, higher-order abstraction, stream processing, and Unix pipelines all instantiate the same compositional discipline. In each case, computation advances through staged commitment of structure, where intermediate transformations are accumulated sequentially without disrupting the associative framework that governs composition.

The pipe operator in Unix and the arrow of the λ-calculus both articulate this invariant. They denote not merely order of execution, but the chaining of unary transformations under a stable algebra of composition. The surface syntax differs, yet the structural principle remains constant:

compose transformations; defer collapse until necessary.

This is the compositional discipline that makes functional programming scalable.

12 Deriving the λ-Calculus from Merge–Collapse Semantics

We now show how the classical untyped λ-calculus emerges as a structured refinement of merge–collapse dynamics.

12.1 Variables as Atomic Regions

In Spherepop, an atomic name a denotes a singleton region

$$a = \{a\}.$$

A variable is therefore not a symbol awaiting substitution, but a minimal structural commitment.

12.2 Abstraction as Region Parameterization

Consider an expression $e(x)$ whose denotation is a region depending on a distinguished atomic name x.

We interpret abstraction

$$\lambda x.\, e$$

as a region-valued transformer that, when given a region R, merges R into the structure of e and collapses the result under identification of x with the representative of R.

Formally:

Definition 5 (Abstraction)**.** *Let e be an expression containing x. Define*

$$\lambda x.\, e(R) = \text{collapse}\left(e \oplus R\right) \quad \textit{with } x \sim R.$$

Thus abstraction is not syntactic binding alone, but a promise to perform a quotient when applied.

12.3 Application as Merge with Identification

Application in the λ-calculus,

$$(\lambda x.\, e)\; a,$$

is traditionally explained through symbolic substitution. The term a replaces the bound occurrences of x in e, and reduction proceeds syntactically. Within the merge–collapse framework, however, application admits a more structural interpretation.

First, the region corresponding to the argument a is merged into the region corresponding to the body e. This accumulation introduces the commitments carried by a into the structural context of e. Second, an identification is introduced equating the placeholder x with the structure supplied by a. This identification extends the governing equivalence relation. Finally, collapse is applied, enforcing the identification canonically across the merged region.

Under this interpretation, symbolic substitution is not primitive. It is the surface manifestation of a deeper operation: the introduction of a quotient followed by canonical collapse. Application is therefore not merely textual replacement, but structural merge with disciplined identification.

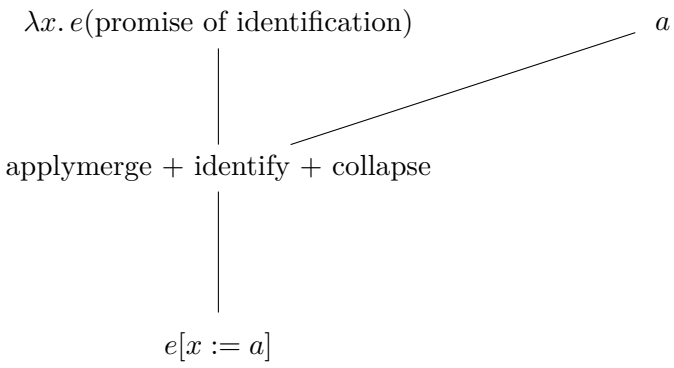

Figure 5: Application is merge with the argument together with the identification $x \sim a$, followed by collapse.

12.4 β-Reduction as Structural Collapse

In classical λ-calculus:

$$(\lambda x.\, e)\ a\ \rightarrow\ e[x := a].$$

In Spherepop terms:

$$\text{collapse}\,\big(e \oplus \{a\}\big).$$

β-reduction is simply collapse after merge.

Theorem 1. *β-reduction corresponds to quotient introduction followed by canonical collapse.*

Proof. Substitution identifies occurrences of x with a. Identification is equivalence relation extension. Collapse computes the canonical representative structure. □

12.5 Confluence and Canonical Regions

The λ-calculus is well known to enjoy confluence: if a term reduces along distinct paths, the resulting normal forms are joinable. This property is central to its stability as a formal system. Evaluation order may vary, but the denotational outcome remains invariant.

Within merge–collapse semantics, confluence follows from structural features of the underlying operations. Merge is associative, and—where order carries no semantic weight—commutative. Collapse acts as a canonical projection onto equivalence classes. As a consequence, distinct sequences of merge operations correspond merely to different parenthesizations or orderings of accumulation, not to genuinely different structural commitments. Once collapse is applied, the result is determined solely by the equivalence relation in force, not by the path taken to assemble the region.

Thus, different evaluation strategies can be interpreted as alternative merge schedules over the same underlying structure. Because collapse is canonical, these schedules converge to an identical

normalized region. Confluence is therefore not an incidental property of symbolic reduction, but a structural consequence of associative accumulation combined with canonical resolution.

13 Typed Structure as Invariant Preservation

Having reconstructed untyped λ-calculus, we now introduce types.

Types are invariants preserved under merge and collapse.

13.1 Regions as a Base Type

The base type is:

```
Region
```

Every well-formed expression denotes a region.

13.2 Function Types

A function type

$$A \to B$$

corresponds to a transformation that, when merging an input region of type A, produces a region of type B without violating structural invariants.

Typing judgments take the familiar form:

$$\Gamma \vdash e : T.$$

But semantically, this asserts:

> Under the quotient and structural commitments encoded in Γ, expression e collapses to a region satisfying invariant T.

13.3 Product Types

Product types arise naturally from pairing:

$$(e_1, e_2)$$

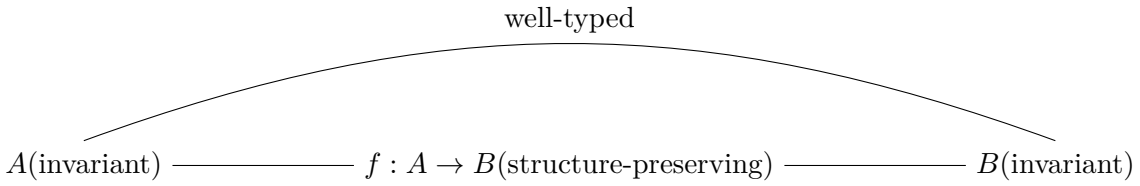

Figure 6: Typing asserts that a transformation preserves the intended invariants from inputs to outputs.

which corresponds to a region whose structure is the merge of two typed components, together with projections implemented as collapse onto the appropriate factor.

Thus product types are structured merges with invariant partitioning.

14 Stack-Based Programming as Explicit Region Sequencing

We now consider stack-based computation from the same structural perspective. In a stack machine, computation proceeds through a disciplined sequence of push and pop operations over an ordered store of intermediate values. The operational presentation emphasizes manipulation of a stack, but structurally the process can be understood in terms of staged accumulation and resolution of commitments.

A push operation corresponds to merging a new region into the current stack state. The added value is not isolated; it becomes part of the evolving structural configuration. A pop operation, by contrast, corresponds to a projection that selects and resolves a particular portion of that configuration, effectively collapsing part of the accumulated structure under an ordering discipline. The stack thus functions as a region whose elements are ordered commitments, where order encodes evaluation context rather than intrinsic identity.

Evaluation in a stack machine is therefore sequential merge–collapse governed by explicit positional constraints. Each instruction transforms the region representing the stack into a new region, and the history of these transformations constitutes the computation. What functional semantics often treats abstractly as evaluation of expressions appears here concretely as a succession of region states. The stack makes visible what is otherwise implicit: computation unfolds as a structured sequence of accumulated commitments and controlled projections.

stack[ ] —— push(merge) —— stack[v] —— pop(collapse) —— stack[ ]

Figure 7: A stack machine makes sequencing explicit: pushes accumulate commitments and pops project a desired view.

15 Currying and Stack-Based Composition

Stack-based languages such as Forth and PostScript compose programs by concatenation. A program is a sequence of operations acting on an implicit stack.

For example, in postfix notation:

```
3 4 + 5 *
```

The operational interpretation is:

1. Push 3.
2. Push 4.
3. Apply $+$.
4. Push 5.
5. Apply $*$.

Composition is simply juxtaposition.

15.1 Concatenation as Merge

In structural terms, each word in a stack language is a region transformer:

$$W : \Sigma \to \Sigma.$$

Program concatenation corresponds to composition:

$$W_1 \; W_2 = W_2 \circ W_1.$$

This is associative.

Thus postfix composition and curried composition share the same algebraic invariant: associative chaining of unary transformers.

15.2 Unary Discipline

Stack languages make unary composition explicit. Each word consumes and produces a stack state. There are no multi-argument functions at the syntactic level; arguments are implicit in stack position.

Currying performs the same reduction:

$$f(x, y, z) \quad \rightsquigarrow \quad f(x)(y)(z).$$

Both systems reduce multi-input computation to chained unary transformations.

The difference between stack-based composition and curried functional composition is primarily representational. In stack languages, sequencing is effected through explicit manipulation of state: values are pushed onto and popped from a visible stack, and control flow is expressed through ordered state transitions. By contrast, in a curried functional setting, structure is passed implicitly through nested application. Intermediate values are not stored in an explicit stack; they are incorporated through the staged transformation of arguments into functions.

Despite this surface divergence, the structural invariant remains the same. Both paradigms implement iterated merge under a disciplined regime of collapse. Stack operations make the accumulation and projection of commitments explicit, while curried functions encode the same process through unary composition. The underlying algebra of accumulation and canonical resolution is shared; only the representational strategy differs.

16 Pipes, Monoidal Composition, and Associativity

Unix pipelines are associative:

```
(f | g) | h = f | (g | h)
```

This property is not accidental. It reflects monoidal structure.

16.1 Monoidal Interpretation

Let programs be morphisms:

$$f : A \to B, \quad g : B \to C, \quad h : C \to D.$$

Composition:

$$h \circ g \circ f : A \to D.$$

Associativity follows from the associativity of composition in a category.

In merge–collapse semantics, this arises from:

$$(R \oplus S_f) \oplus S_g \oplus S_h = R \oplus (S_f \oplus S_g \oplus S_h).$$

Collapse then produces a canonical result.

16.2 Pipes as Monoidal Product in Time

If merge is monoidal product, then a pipeline is monoidal composition ordered temporally.

The pipe operator is therefore an operational notation for associative structural merge.

This explains why pipelines scale: they rely on algebraic laws.

17 Worked Example: Translating a Unix Pipeline into λ-Terms

We now trace a simple pipeline:

```
cat file | grep foo | sort
```

Abstractly, this is:

$$\texttt{sort} \circ \texttt{grep foo} \circ \texttt{cat file}.$$

Let us model streams as regions S.

Define:

$$\texttt{cat} : \text{File} \to \text{Stream}$$

$$\texttt{grep} : \text{Pattern} \to \text{Stream} \to \text{Stream}$$

$$\texttt{sort} : \text{Stream} \to \text{Stream}$$

Curried form:

$$\texttt{grep} : \text{Pattern} \to (\text{Stream} \to \text{Stream}).$$

The pipeline becomes:

$$\texttt{sort}\big((\texttt{grep foo})(\texttt{cat file})\big).$$

Fully expanded:

$$\lambda s.\, \texttt{sort}((\texttt{grep foo})\ s) \circ \texttt{cat file}.$$

In merge–collapse form, let $S_{\texttt{cat}}$, $S_{\texttt{grep}}$, $S_{\texttt{sort}}$ be structural commitments.

Then the final stream is:

$$\text{collapse}(S_{\texttt{file}} \oplus S_{\texttt{cat}} \oplus S_{\texttt{grep}} \oplus S_{\texttt{sort}}).$$

The pipeline is simply staged merge.

17.1 Structural Insight

The translation from pipelines to curried λ-terms reveals that what appears as command chaining in one setting corresponds precisely to function composition in another. Pipelines implement composition directly; the pipe operator enforces the sequential transformation of input through successive stages. At the structural level, this composition is associative merge: intermediate results accumulate in a manner that is independent of grouping, provided order is preserved.

Currying makes this unary structure explicit. By reducing multi-argument application to staged unary transformations, it aligns the formal calculus with the operational intuition of pipelines. Collapse then enforces canonical output, ensuring that different compositional paths converge to an invariant result.

The syntactic surfaces differ—shell commands, postfix notation, or λ-terms—but the structural invariant is the same. Each embodies associative accumulation followed by canonical resolution under a stable algebra of composition.

18 Unifying View

We may now summarize:

Stack Languages	Unix Pipes	Curried λ
Concatenation	| operator	Function composition
Implicit state	Stream flow	Explicit argument passing
Unary words	Unary filters	Unary functions
Associative chaining	Associative pipes	Associative composition

All three implement the same algebraic discipline:

Iterated unary transformation under associative composition.

This is precisely the discipline required for merge–collapse semantics to scale.

Composition is not syntax. It is structure.

19 Functional Programming as Replayable History

Pure functional programming is characterized by the absence of side effects, deterministic evaluation, and referential transparency. These properties are not merely stylistic constraints; they enforce a specific structural discipline. When a program is forbidden from mutating shared state or performing uncontrolled effects, each transformation becomes an explicit contribution to the overall computation rather than an implicit alteration of context.

Within the merge–collapse framework, this discipline corresponds to an append-only accumulation of structural commitments. Each step contributes additional structure through merge, while collapse produces a canonical representation of the accumulated result. Because no operation erases or mutates prior commitments, the entire computation may be regarded as a history that can be replayed deterministically from its initial conditions. Referential transparency follows from this stability: identical structural inputs yield identical canonical outcomes.

Functional evaluation thus takes the form of deterministic replay over a structured log of transformations. The absence of side effects ensures that collapse is applied only to internally accumulated structure, not to an external environment whose state might vary independently. In this sense, a functional program is a controlled sequence of merge–collapse operations whose result depends solely on its explicit history.

The kernel implementation introduced earlier can be understood as an abstract replay engine of precisely this sort. It records structural commitments in append-only form and derives observable state through canonical collapse. Functional programming, viewed in this way, is not merely a paradigm of immutability; it is a disciplined method for preserving the replayability of computational history.

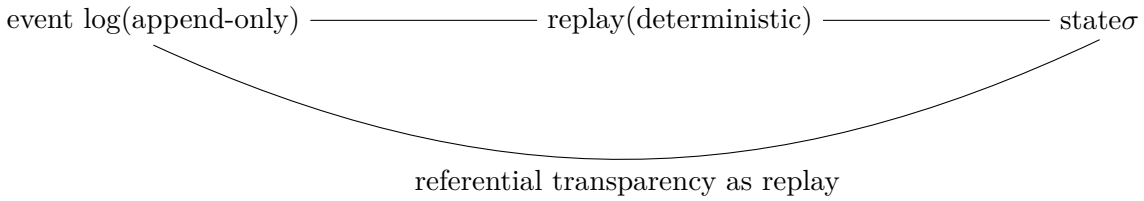

Figure 8: Pure functional evaluation corresponds to deterministic replay of history into a canonical state.

20 Toward a Unified Structural View

We may now summarize the correspondences:

Spherepop	Classical Computation
Merge	Application / Composition
Collapse	Substitution / Evaluation
Optionality	Degrees of Freedom
Quotient	Type Constraint / Unification
Replay	Referential Transparency

The point is not that Spherepop replaces established formalisms, but that it exposes their shared invariant:

> Computation is structured reduction of optionality under compositional constraint.

21 Computation as Structured Transformation

Beginning from regions and two primitive operations, we have outlined how a substantial portion of the theory of computation can be reconstructed from a minimal structural core. The λ-calculus appears as a disciplined regime of merge and collapse, where abstraction introduces controlled identification and evaluation enforces canonical form. Type theory emerges as the preservation of invariants under these operations, constraining which identifications are admissible. Stack-based models can be read as explicit sequencing of region transformations, while functional programming presents computation as deterministic replay over structured commitments.

Seen from this vantage point, what often appears as a collection of separate techniques resolves into a common algebra. Symbolic presentations emphasize syntax and reduction rules; the structural perspective emphasizes accumulation, normalization, and invariance. The geometry underlying computation becomes visible before its surface notation.

The sections that follow deepen this account. In particular, the discussion of abstraction as structured reduction will examine more carefully how identification compresses structure without destroying coherence. Beyond that, the same minimal calculus can be refined toward richer type systems, categorical semantics, or cost-aware interpretations. Even at the present level of development, however, the merge–collapse substrate suggests that computation is most clearly understood as structured transformation prior to syntax.

22 Abstraction as Structured Reduction

Abstraction is often presented as a purely syntactic device: a means of binding a variable so that it may later be substituted. In the structural perspective developed here, abstraction is instead a disciplined act of reduction. It removes incidental detail while preserving invariant structure.

To abstract is not merely to hide a name. It is to quotient a family of concrete configurations into a single structural pattern.

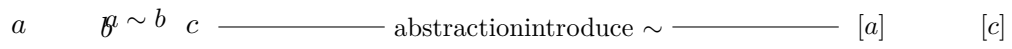

Figure 9: Abstraction is controlled identification: fewer distinctions, preserved invariant.

22.1 Abstraction and Invariance

Let R be a region constructed from atomic commitments

$$R = \{a_1, a_2, \ldots, a_n\}.$$

Suppose that a subset of these elements differ only in accidental labeling. Introducing an equivalence relation

$$a_i \sim a_j$$

collapses these distinctions.

Abstraction is precisely this introduction of equivalence: we identify differences that are irrelevant to the invariant we wish to preserve.

Definition 6 (Structural Abstraction)**.** *Given a region R and an equivalence relation $\sim$ encoding irrelevant distinctions, the abstraction of R under $\sim$ is*

$$\text{collapse}_{\sim}(R).$$

Abstraction therefore reduces optionality:

$$\Omega(\text{collapse}_{\sim}(R)) \leq \Omega(R).$$

The art of abstraction lies in choosing $\sim$ so that essential structure remains intact.

22.2 Abstraction in λ-Calculus

In the λ-calculus, abstraction is written

$$\lambda x.\, e.$$

From the merge–collapse perspective, abstraction performs two acts:

1. It marks x as a parameter rather than a fixed commitment.
2. It promises a future identification when an argument is supplied.

Formally, abstraction lifts a region-valued expression into a transformation

$$R \longmapsto \text{collapse}(R \oplus e).$$

Thus abstraction removes the particularity of a name while retaining the structural pattern of dependence.

22.3 Example: Abstracting Over Addition

Consider a simple symbolic expression:

$$x + 1.$$

In traditional algebra, abstraction yields the function

$$\lambda x.\, x + 1.$$

Structurally, we begin with a region containing commitments for x and 1. Abstraction removes the concrete identity of x and replaces it with a placeholder to be identified later.

When applied to 3, the identification $x \sim 3$ is introduced, and collapse produces a concrete region.

The abstraction step therefore consists of:

- Removing a specific atomic identity.
- Preserving the dependency relation.

This is a reduction of specificity while preserving pattern.

22.4 Abstraction as Factorization

Another perspective is factorization.

Suppose a computation produces

$$R = \{a, b, c, f(a), f(b), f(c)\}.$$

We observe a repeated structural dependency on f. Abstraction factors this pattern:

$$R \quad \rightsquigarrow \quad \lambda x.\, f(x).$$

The region is compressed by recognizing repetition. Collapse encodes equivalence classes of behavior, yielding a smaller description with preserved generative power.

22.5 Abstraction and Type Formation

Types formalize abstraction by enforcing invariant preservation.

When we write

$$\lambda x : A.\, e : B,$$

we assert that for every region R satisfying invariant A, the merged and collapsed result satisfies invariant B.

Type abstraction is therefore abstraction at the level of constraints.

22.6 Abstraction and Stack Discipline

In stack-based programming, abstraction appears as macro definition or function definition. A sequence of stack manipulations is encapsulated and named.

Structurally, this:

- Packages a region transformation.
- Suppresses intermediate commitments.
- Exposes only the boundary behavior.

The stack itself is a sequence of region states; abstraction hides the internal merge–collapse steps.

22.7 Abstraction as Reduction of Description Length

We may quantify abstraction in terms of description length.

If a region R has many distinguishable atomic commitments, but its behavior depends only on a smaller invariant subset, then abstraction reduces the effective description.

Let $D(R)$ denote a description complexity measure. If abstraction introduces equivalences $\sim$, then ideally

$$D(\text{collapse}_{\sim}(R)) < D(R)$$

while preserving behavioral invariants.

This perspective aligns abstraction with compression.

22.8 Hierarchies of Abstraction

Abstraction operates at multiple strata within a computational system, and each stratum refines the granularity at which structural distinctions are preserved or ignored. At the most elementary

level, abstraction begins with naming: atomic commitments are assigned stable identifiers, allowing recurring structure to be referenced rather than duplicated. This act already introduces a form of equivalence, insofar as repeated occurrences of a name are treated as structurally identical.

A second layer arises through parameterization over values. Functions abstract over concrete inputs, replacing particular commitments with placeholders that may later be identified with specific regions. Here abstraction takes the form of controlled identification under application, enabling families of transformations to be expressed generically.

Type formation introduces a further level of abstraction. By grouping values under shared invariants, type systems collapse distinctions that are irrelevant to well-formedness while preserving those necessary for correctness. Types thus function as disciplined equivalence classes over terms, enforcing structural constraints across potentially diverse representations.

At a higher level still, module encapsulation abstracts over internal implementation details. Distinctions within a module are rendered opaque to external observers, and only selected interfaces remain visible. This establishes a boundary across which certain equivalences are maintained and others suppressed.

Finally, interface design governs abstraction at the level of system architecture. It specifies which commitments are exposed and which remain internal, shaping the granularity at which components interact. In each case, abstraction introduces new equivalence relations that collapse finer distinctions into coarser invariants. The progression from names to interfaces is therefore not a change in kind, but a change in scale: abstraction repeatedly restructures commitment by regulating which differences are preserved and which are canonically identified.

22.9 Abstraction and Confluence

Because abstraction operates via canonical collapse, it preserves confluence when properly structured.

Different evaluation paths correspond to different orders of merge. If collapse is canonical, the abstracted structure is invariant.

Thus abstraction, when grounded in merge–collapse, is stable under reordering.

23 Abstraction as Controlled Forgetting

A final observation is in order before proceeding further. Abstraction is often described as the act of omitting detail. Yet omission alone would amount to ignorance. The form of abstraction operative in computation is more precise. It is a controlled and principled form of forgetting, one that discards certain distinctions in order to preserve and clarify others.

In structural terms, merge increases commitment. Each merge accumulates additional distinctions

and relationships within a region. Collapse, by contrast, eliminates distinctions by identifying elements under an equivalence relation. Abstraction consists in selecting which distinctions are to be rendered irrelevant and enforcing their identification through disciplined collapse. What remains after this process is not impoverished structure, but a more stable invariant, one that supports composition and reuse.

The interplay between accumulation and forgetting is therefore not accidental. Computation advances by introducing commitments and then reorganizing them through controlled identification. Without accumulation there would be no structure to reason about; without forgetting there would be no tractable invariants. The dynamic balance between these operations forms the engine of abstraction.

In the next section, we examine how this process manifests in higher-order functions and compositional closure, where abstraction no longer operates merely over values, but over transformations themselves.

24 Higher-Order Abstraction and Compositional Closure

Abstraction becomes genuinely powerful when it is allowed to range over transformations themselves. In the λ-calculus this is expressed by higher-order functions: functions that take functions as arguments or return them as values.

From the merge–collapse perspective, higher-order abstraction is abstraction over region transformers.

$$F(\text{transformer}) \xrightarrow{\text{input is a function}} H(F)(\text{higher-order}) \xrightarrow{\text{output is a function}} G(\text{transformer})$$

Figure 10: Higher-order abstraction treats transformers themselves as manipulable values.

24.1 Functions as Structured Regions

Recall that a function in our setting is interpreted as a rule

$$f : R \longmapsto \text{collapse}(R \oplus S_f)$$

for some structural commitment S_f.

Thus a function is determined by the region S_f together with the discipline of how collapse is applied.

Higher-order abstraction therefore merges region transformers and collapses over transformer identity.

24.2 Composition as Iterated Merge

Function composition

$$(f \circ g)(x) = f(g(x))$$

corresponds structurally to:

1. Merge x with S_g and collapse.
2. Merge the result with S_f and collapse again.

Associativity of composition follows from associativity of merge and canonicity of collapse.

Proposition 3. *Function composition in this model is associative up to canonical normalization.*

Proof. Merge is associative and collapse is idempotent projection. □

Thus higher-order structure emerges naturally from the base calculus.

24.3 Fixed Points and Recursive Abstraction

Recursion introduces self-reference. In classical λ-calculus, fixed-point combinators allow a function to refer to itself.

Structurally, a fixed point identifies a transformer with its own application.

Let F be a region transformer. A fixed point satisfies

$$X = F(X).$$

In merge–collapse semantics, this corresponds to introducing an equivalence

$$X \sim F(X)$$

and collapsing.

Recursion is therefore a quotient over a self-generated region. The stability of recursion depends on whether repeated merge converges under collapse.

This connects termination to structural contraction.

24.4 Abstraction and Closure

In implementation, abstraction produces closures. A closure pairs:

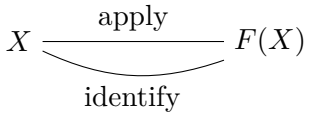

Figure 11: A fixed point is a disciplined self-identification: $X \sim F(X)$ under a collapse that stabilizes the recursion.

- A body expression.
- An environment of bindings.

In structural terms, a closure is simply a region whose commitments include:

- Structural pattern.
- Contextual quotient structure.

Application merges argument region with closure region, and collapse resolves bindings.

Thus closures are region transformers with embedded context.

25 Abstraction and Category-Theoretic Structure

The merge operation endows regions with monoidal structure. Collapse acts as canonical projection. Abstraction can then be viewed as internal hom formation.

25.1 Monoidal Structure

Let $(\mathcal{R}, \oplus, \varnothing)$ denote the collection of regions under merge.

Then:

- $\oplus$ is associative.
- $\varnothing$ acts as identity.

Thus $(\mathcal{R}, \oplus)$ forms a monoidal category when equipped with morphisms preserving collapse structure.

25.2 Internal Hom

Abstraction corresponds to constructing an object $[A \Rightarrow B]$ such that:

$$A \oplus [A \Rightarrow B] \to B.$$

This is precisely the defining property of internal hom in a monoidal closed category.

Thus λ-abstraction is not an arbitrary syntactic device, but a categorical reflection of merge structure.

26 Worked Example: From Region to Typed Functional Program

We now trace a simple example from primitive structure to typed abstraction.

Step 1: Atomic Commitments

Let the initial region be:

$$R_0 = \{a, b\}.$$

Step 2: Merge with Structure

Introduce a structural commitment S:

$$R_1 = R_0 \oplus \{f(a), f(b)\}.$$

Step 3: Observe Pattern

We detect repeated dependency on f. Abstract over the argument:

$$\lambda x.\, f(x).$$

This replaces two concrete instances with a generative rule.

Step 4: Introduce Type Constraint

Declare:

$$f : A \to B.$$

This enforces invariant preservation.

Step 5: Interpret as Functional Program

We now have a pure function:

$$\texttt{map}\ f.$$

Evaluation becomes replayable transformation over region states.

The original explicit structure has been compressed into typed abstraction.

27 Abstraction as Compression of Histories

Finally, consider a computation as a sequence of region states:

$$R_0 \to R_1 \to R_2 \to \cdots \to R_n.$$

Abstraction can compress this history into a single transformer F such that:

$$R_n = F(R_0).$$

Functional programming systematically replaces explicit history with composable transformers.

In merge–collapse terms, we compress a chain of merges and collapses into a higher-order region commitment.

28 Closing Reflection

Abstraction is the mechanism by which computation rises from concrete manipulation to compositional structure. It permits local operations to participate in larger patterns without forfeiting coherence. By regulating which distinctions are preserved and which are identified, abstraction makes scalable composition possible.

Within the framework developed here, merge accumulates structure by introducing new commitments, while collapse enforces canonical form by resolving distinctions under an equivalence relation. Abstraction governs this process by selecting the invariants that are to be maintained across trans-

formations. Composition then extends those invariants across increasingly complex constructions, allowing local structure to propagate coherently through larger systems.

Taken together, these operations form a disciplined cycle of accumulation, identification, normalization, and recomposition. Through this cycle, structure is introduced, reorganized, stabilized, and then extended to new contexts. It is this recurring process that sustains abstraction as a generative force within computation, allowing increasingly complex systems to be constructed without forfeiting coherence.

From these structural dynamics arise the familiar architectures of theoretical and practical computation. The λ-calculus emerges as a calculus of disciplined identification under application. Type systems arise as invariant-preserving constraints on permissible collapses. Stack machines make explicit the sequencing of accumulated commitments. Pure functional languages encode evaluation as canonical normalization over an append-only history of transformations.

Abstraction is therefore not an auxiliary feature layered atop computation. It is the regulated collapse of distinctions deemed irrelevant to a chosen invariant. In performing this regulation, abstraction reveals the geometry that computation inhabits: a space structured not merely by symbols, but by the disciplined organization and resolution of commitments.

29 Monads as Delayed Collapse: Controlling Side Effects

Side effects complicate functional reasoning. Output, mutation, exceptions, and state appear to violate referential transparency.

Monads provide a structural discipline that allows side effects to be represented without abandoning purity.

From the merge–collapse perspective, monads control when collapse is allowed.

29.1 The Problem: Premature Collapse

Consider output.

If a program directly writes to the console, the effect is irreversible. Collapse has occurred in the external world.

Once a character is printed, it cannot be unprinted.

In structural terms, this is uncontrolled collapse: distinctions between “output planned” and “output executed” have been eliminated.

Functional programming avoids this by delaying collapse.

29.2 IO as Structured Commitment

Instead of performing output immediately, a functional program constructs a value of type:

$$\texttt{IO}\ A.$$

This value represents a structured description of a computation that, when executed, will produce an A and perform effects.

In merge–collapse language:

- The program merges commitments describing effects.
- Collapse into the external world is deferred.

An IO value is therefore a region encoding an effectful plan.

29.3 Monad as Sequencing Discipline

A monad consists of:

$$\texttt{return} : A \to MA$$
$$\texttt{bind} : MA \to (A \to MB) \to MB.$$

Structurally:

- `return` embeds a pure region into an effect region.
- `bind` merges effect descriptions sequentially.

Bind does not collapse effects. It accumulates them.

Only the runtime system performs the final collapse, interpreting the effect region externally.

29.4 Output as Accumulated Region

Suppose we write:

```
do
  putStrLn "Hello"
  putStrLn "World"
```

In IO monad form:

$$\texttt{putStrLn}\ \text{"}Hello\text{"}\ >>=\ \lambda\_.\ \texttt{putStrLn}\ \text{"}World\text{"}.$$

Each `putStrLn` contributes a structural commitment:

$$S_{\texttt{Hello}},\quad S_{\texttt{World}}.$$

Bind merges them:

$$S = S_{\texttt{Hello}} \oplus S_{\texttt{World}}.$$

The program constructs S. It does not yet perform collapse.

Only at the top level does the runtime interpret S and produce observable output.

29.5 Monad Laws as Structural Laws

The monad laws:

$$\texttt{return}\ a\ >>=\ f = f(a)$$

$$m\ >>=\ \texttt{return} = m$$

$$(m\ >>=\ f)\ >>=\ g = m\ >>=\ (\lambda x.\, f(x)\ >>=\ g)$$

mirror merge–collapse laws.

Associativity of bind corresponds to associativity of merge.

Left and right identity correspond to identity under empty region.

Thus monads are algebraic wrappers around merge.

29.6 Delayed Evaluation as Deferred Collapse

The key structural insight:

> A monad delays collapse.

Effects are accumulated structurally inside the monadic region. They are only collapsed when interpreted.

This preserves referential transparency.

Two IO expressions with identical region structure are equal, even before execution.

29.7 Side Effects as Controlled Boundary Collapse

We may now interpret:

- Pure computation: merge + collapse internally.
- Effectful computation: merge internally, collapse externally.

The monad provides a boundary. Inside the boundary, structure accumulates. Outside, effects occur.

This separation parallels the distinction between event log (authoritative history) and derived semantic view.

29.8 Why This Matters

Without monads, side effects collapse structure prematurely. Reasoning becomes unstable.

With monads:

- Effects are values.
- Sequencing is associative.
- Collapse is controlled.
- Purity is preserved.

In Spherepop terms, monads enforce discipline over where and when collapse is allowed.

They are not mystical containers. They are structural postponements of irreversibility.

30 Conclusion: Effects as Structured Plans

The merge–collapse lens reveals that functional purity does not deny side effects. It reclassifies them.

Effects become structured commitments, composed associatively, collapsed only at controlled boundaries.

Monads are therefore not an add-on. They are a principled extension of the same algebraic discipline that governs merge, collapse, and abstraction.

31 Algebraic Effects as Structured Effect Algebras

Monads provide a mechanism for sequencing effects. Algebraic effects refine this idea by separating the description of effects from their interpretation.

31.1 Effect Signatures

An algebraic effect system begins with a signature:

$$\Sigma = \{\texttt{Print} : \text{String} \to 1, \texttt{Read} : 1 \to \text{String}, \dots\}$$

Each operation describes a possible effectful action.

In merge–collapse terms, these operations correspond to atomic effect commitments:

$$e \in \mathcal{E}.$$

A computation is not a direct collapse into the world. It is a region built from these effect atoms.

31.2 Free Construction

Given effect atoms $\mathcal{E}$, we construct formal effect expressions by merge:

$$E ::= \varnothing \mid e \mid E_1 \oplus E_2.$$

Associativity of merge ensures that effect sequencing is structurally stable.

Thus an effectful computation is simply a region over $\mathcal{E}$.

31.3 Handlers as Collapse Interpreters

An effect handler provides a mapping:

$$-_H : \mathcal{R}_{\mathcal{E}} \to \mathcal{W},$$

where $\mathcal{W}$ represents the external world state.

The handler performs collapse:

$$\text{collapse}_H(E).$$

Thus algebraic effects make explicit what monads encode implicitly: effects are accumulated first, interpreted later.

31.4 Structural Advantage

This separation provides clarity:

- Merge constructs effect structure.
- Collapse interprets it.
- Different handlers correspond to different collapse maps.

The calculus remains pure. Interpretation is modular.

In Spherepop language, handlers are parameterized collapse operators.

32 IO as a Free Monoid over Effect Actions

We now make the event-log interpretation precise.

32.1 Free Monoid Structure

Let $\mathcal{E}$ be a set of primitive effect actions, such as:

$$\texttt{Print("Hello")}, \quad \texttt{Print("World")}.$$

The free monoid over $\mathcal{E}$ is:

$$\mathcal{E}^*,$$

the set of finite sequences of effect actions, with concatenation as the monoidal operation and the empty sequence as identity.

32.2 IO Values as Effect Sequences

An IO computation can be modeled as:

$$\texttt{IO}\ A \cong \mathcal{E}^* \times A.$$

That is, a pair consisting of:

- A sequence of effect actions.
- A pure result.

Bind composes these sequences:

$$(e_1, a) \ >>= \ f = (e_1 \cdot e_2, b),$$

where:

$$f(a) = (e_2, b).$$

Thus bind is concatenation of event logs.

32.3 Associativity and Identity

Concatenation is associative:

$$(e_1 \cdot e_2) \cdot e_3 = e_1 \cdot (e_2 \cdot e_3).$$

The empty sequence is identity.

Therefore the monad laws follow directly from monoid laws.

32.4 Collapse into the External World

Execution corresponds to interpreting the sequence:

$$e_1 \cdot e_2 \cdot \cdots \cdot e_n.$$

This interpretation is an irreversible collapse. It produces observable effects.

Before interpretation, the computation is pure structure.

After interpretation, collapse has occurred.

32.5 Event Logs and Structural Transparency

This free monoid view clarifies:

- IO does not perform effects internally.

- It builds a log.
- The runtime system collapses the log.

Thus IO is structurally identical to an append-only event log.

Merge corresponds to concatenation. Collapse corresponds to replay.

32.6 Reconciliation with Merge–Collapse

In earlier sections, we described computation as:

$$\text{collapse}(R \oplus S_1 \oplus S_2 \oplus \dots).$$

The IO monad makes this explicit.

Each S_i is an effect atom. Merge accumulates them. Final collapse replays them.

Monads therefore encode a free monoidal structure with delayed collapse.

33 Unified Structural Picture

We can now see:

Pure Functions $\longrightarrow$ $ImmediateCollapse$

Monadic Effects $\longrightarrow$ $AccumulatedMerge, DeferredCollapse$

Algebraic Effects $\longrightarrow$ $ParameterizedCollapseviaHandlers$

All three preserve the same algebraic backbone.

The only difference lies in where collapse is permitted.

This reinforces the central claim:

> Computation is merge. Semantics is collapse. Effects are controlled postponements of irreversibility.

34 State, Continuations, and Mutation Through Spherepop

We now extend the structural lens to three topics that often feel separate in standard presentations: state monads, continuation-passing style, and mutation. In the merge–collapse view, they differ

primarily by where a program is permitted to store commitment and where it is permitted to collapse it.

34.1 The State Monad as an Explicit Evolving Region

A stateful computation is commonly typed as:

$$\texttt{State}\ S\ A\ \cong\ S \to (A \times S).$$

This says: a computation consumes a state and returns both a value and a new state.

In the merge–collapse interpretation, the state S is a region recording current commitments. A stateful program is therefore a transformer over regions:

$$F : \Sigma \to \Sigma,$$

where Σ is the space of admissible state regions.

Execution becomes iteration of merge–collapse updates:

$$S_{t+1} = \text{collapse}\left(S_t \oplus \Delta_t\right),$$

where Δ_t is the structural increment contributed by the program at step t.

In purely functional state, this update is not destructive. It produces a new region S_{t+1} without altering S_t. The "history" of state is therefore a sequence of regions, and the discipline of the state monad ensures that this history is compositional.

34.2 Bind for State as Sequential Merge of Updates

Bind for the state monad composes two computations:

$$m : S \to (A \times S), \qquad f : A \to (S \to (B \times S)).$$

Their composition is:

$$(m\ >>=\ f)(s) = \text{let}\ (a, s') = m(s)\ \text{in}\ f(a)(s').$$

Structurally, the first computation produces an intermediate region s', which is then fed into the next transformer. This is the same staged-merge pattern seen in pipelines:

$$s \xrightarrow{m} s' \xrightarrow{f(a)} s''.$$

State monads therefore implement Unix-pipe style composition, but with an explicit state channel and an explicit pairing of result and updated region.

34.3 Continuation-Passing Style as Explicit Control of "What Happens Next"

In continuation-passing style, instead of returning a value, a function receives an additional argument: a continuation k describing the rest of the computation.

A CPS-transformed function has the schematic type:

$$A \to (A \to R) \to R.$$

In structural terms, a continuation is a promise of future merge–collapse operations. It is the remainder of the pipeline packaged as a function.

Thus CPS makes explicit what ordinary composition leaves implicit:

> A computation is not merely a value-producing function, but a transformer embedded in a larger chain.

The continuation is the chain.

From the Spherepop viewpoint, CPS is a method for controlling the staging of collapse. Instead of collapsing a result and returning it, one passes it forward as raw structure to the continuation, allowing later stages to decide how and when to normalize.

34.4 Continuations and Deferred Collapse

In ordinary evaluation, a computation might collapse intermediate structure repeatedly, forcing canonicalization early. In CPS, intermediate results can be forwarded without committing to a particular collapse regime until the continuation applies it.

This creates a spectrum:

- Eager evaluation: frequent collapse, early canonicalization.
- Lazy or staged evaluation: delayed collapse, later canonicalization.
- CPS: collapse becomes explicitly governed by the continuation structure.

Thus continuations are a control structure for collapse.

34.5 Mutation as Collapse Without History

We now address mutation.

Mutation is often treated as the defining feature of imperative computation: update a variable in place. From the merge–collapse perspective, mutation is not mysterious. It is a specific structural violation: collapse that overwrites the ability to replay history.

In a pure setting, we represent updates as new regions:

$$S_{t+1} = \text{collapse}(S_t \oplus \Delta_t).$$

The prior state S_t remains available as a value.

In mutation, we instead enforce:

$$S_t \leftarrow S_{t+1}$$

and discard the ability to treat S_t as a stable semantic object. The distinction between "old" and "new" state is collapsed in the meta-level representation.

This is a collapse of time-indexed identity.

34.6 Mutation as Untracked Collapse at the Boundary

In earlier sections, we emphasized the separation between authority and view: an event log as the authoritative substrate, and derived regions as semantic views.

Mutation occurs when collapse is allowed to happen at the authoritative layer without recording it as an event. In that regime:

- The system changes.
- But the structural cause is not preserved.
- Replay is no longer possible.

This is precisely why mutation complicates reasoning: it is collapse without explicit trace.

Conversely, one can reconstruct the benefits of mutation while retaining structural clarity by recording updates as events. The event log then restores the separation:

$$\text{authority} = \text{append-only merge of events}, \qquad \text{view} = \text{collapse by replay}.$$

In this light, state monads and event sourcing are not stylistic choices. They are structural disciplines preventing collapse from erasing provenance.

34.7 A Unifying Structural Summary

We may now summarize:

Paradigm	Structural Mechanism	Spherepop Interpretation
State monad	$S \to (A \times S)$	region transformer with explicit state channel
CPS	$(A \to R) \to R$	explicit "rest of pipeline" controlling collapse staging
Mutation	in-place update	untracked collapse that erases replay structure

All three are variations on one theme: where does the system store commitment, and where is it permitted to collapse it?

35 Control of Collapse as the Deep Invariant

Across λ-calculus, types, pipes, stacks, monads, algebraic effects, state, CPS, and mutation, the same structural invariant repeats.

Merge accumulates commitments. Collapse resolves commitments canonically.

The paradigms differ only in their discipline about when collapse is permitted, and whether the history of commitment remains replayable.

Spherepop provides a minimal vocabulary for this discipline.

36 Final Synthesis: The Calculus of Commitment

We are now in a position to integrate the threads developed throughout this text.

We began not with machines, syntax, or evaluation rules, but with a minimal structural substrate: finite regions equipped with two primitive operations, merge and collapse. From this sparse beginning, increasingly rich computational phenomena were reconstructed. What appeared at first as a pedagogical device revealed itself as a generative core.

Merge accumulates commitment. Collapse resolves commitment under equivalence. Optionality measures structural freedom. Abstraction introduces disciplined identification. Composition extends invariants across accumulated structure. These operations, taken together, generate the dynamic cycle that underlies all subsequent constructions.

The λ-calculus emerged as a disciplined fragment of this substrate. Variables were interpreted as atomic commitments. Abstraction was reinterpreted as quotient introduction. Application became

merge followed by canonical collapse. Confluence followed not from syntactic accident, but from associativity of accumulation and stability of projection.

Type theory arose as invariant preservation under merge and collapse. Typing judgments were recast as structural guarantees: certain identifications are permitted, others prohibited. Product types reflected structured merge; function types expressed controlled region transformation; categorical semantics clarified the algebra already implicit in the base operations.

Stack machines made explicit what functional semantics leaves implicit: evaluation as sequenced region transformation. Currying revealed that multi-argument computation reduces to iterated unary transformation. Unix pipelines, postfix composition, and higher-order functions were shown to share a single associative backbone. Their surface differences masked an identical structural discipline.

Functional programming appeared not merely as a style, but as a commitment to replayable history. Pure evaluation preserved structural provenance. Monads were interpreted as delayed collapse: effect structure accumulated internally and was interpreted only at controlled boundaries. Algebraic effects separated description from interpretation. State, continuations, and mutation differed only in where collapse was permitted and whether commitment history remained visible.

Across these domains, the same invariant recurred. Computation is not arbitrary transformation. It is structured reorganization of commitment under compositional constraint. Merge introduces distinctions. Collapse removes distinctions in accordance with chosen invariants. Abstraction governs the boundary between them.

The perspective developed here does not replace established theories. It does not challenge the Church–Turing thesis, nor displace λ-calculus, type systems, or categorical semantics. Rather, it identifies a minimal algebraic substrate from which these theories can be seen as stratified refinements. Where classical presentations emphasize syntax and rule systems, the present account emphasizes structural accumulation and canonical resolution.

In this light, computation acquires a geometric character. Before symbols are rewritten, commitments are accumulated. Before evaluation rules fire, equivalences are introduced. Before types are checked, invariants are selected. The visible calculi of programming languages become surface expressions of a deeper structural logic.

The calculus of commitment therefore names not a new formalism, but a way of seeing. It proposes that beneath abstraction, typing, composition, effects, and control lies a simple discipline: accumulate structure, identify what is irrelevant, normalize canonically, and compose again. From that discipline, the architectures of modern computation follow.

If the exposition has succeeded, the reader may now recognize familiar formalisms as particular crystallizations of this underlying dynamic. The metaphor introduced under the name *Spherepop* was intended only as a compact vessel for that dynamic—a way of speaking about regions, commitment, and collapse without presupposing the vocabulary of any single tradition.

What remains is refinement. The same substrate can be extended toward dependent type theory,

richer categorical models, cost semantics, or effect systems with finer granularity. Yet even at this level, the central insight stands: abstraction is controlled collapse; composition is associative accumulation; computation is structured transformation of commitment.

References

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

[2] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[3] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[4] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[5] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[6] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[7] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. *Information Processing 83*, 513–523, 1983.

[8] Philip Wadler. Theorems for Free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 1989.

[9] Peter J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

[10] John Backus. Can Programming Be Liberated from the von Neumann Style? *Communications of the ACM*, 21(8):613–641, 1978.

[11] Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD Machine, and the Lambda-Calculus. In *Formal Description of Programming Concepts*, 1992.

[12] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[13] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1998.

[14] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[15] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[16] Christopher Strachey. Fundamental Concepts in Programming Languages. Oxford University Computing Laboratory Technical Monograph, 1967.

[17] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North-Holland, 1958.

[18] Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 1995.

[19] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture*, 1991.

[20] Erik Meijer. Calculating Compilers. PhD Thesis, University of Nijmegen, 1992.

[21] Erik Meijer. Your Mouse is a Database. ACM Queue, 2008.

[22] Philip Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, 1995.

[23] Eugenio Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

[24] Richard Bird and Lambert Meertens. Nested Datatypes. In *Mathematics of Program Construction*, 1998.

[25] Donald E. Knuth. *The Art of Computer Programming*, Vol. 1: Fundamental Algorithms. Addison-Wesley, 3rd edition, 1998.

[26] Dana Scott. Outline of a Mathematical Theory of Computation. Technical Monograph PRG-2, Oxford University Computing Laboratory, 1970.