Abstraction as Reduction:

A Proof–Theoretic and Computational Perspective

Flyxion

December 11, 2025

Abstract

Abstraction is often treated as a conceptual distancing from the substrate of a system—a movement upward in a hierarchy of descriptions—yet it is equally accurate to regard abstraction as a disciplined form of evaluation, a reduction step that preserves structure while eliminating the inessential. This essay develops the view that abstraction is not a retreat from implementation but a mode of operational focus, closely analogous to the reduction strategy of the lambda calculus, the type-signature discipline of functional programming, the boxed modules of category theory, and even the incomplete yet resolvable signals of asynchronous dual-rail circuits. When understood through the Curry–Howard correspondence, abstraction becomes isomorphic to proof evaluation: to abstract is to execute, to hide is to compute, and to move upward is to eliminate detail by validating the logical transformation that licenses that elimination.

1 Introduction

It is common to speak of abstraction as though it were a gesture of concealment: one hides the mechanism and exposes only an interface, a boundary surface, a promise of how some internal process will behave without explaining the manner of its internal unfolding. Yet a richer and more precise account emerges when we observe that abstraction is not merely concealment, but a mode of *reduction*: a procedure by which one identifies, evaluates, and then compresses a segment of a structure in order to make the whole more intelligible, portable, and composable.

This sense of abstraction—as an evaluative step in a theory of computation—is not metaphorical. It is deeply aligned with the operational semantics of the lambda calculus, where the difficult work of meaning occurs in the innermost scope, and the remainder of the structure clarifies itself progressively as those nested parts reduce to normal forms. Abstraction thus parallels computation: to hide the details of an evaluated term is simply to acknowledge that one has already reduced that term sufficiently that its internal workings no longer impose obligations on the surrounding context.

The purpose of this essay is to show that abstraction, so understood, is structurally identical to notions of evaluation, execution, normalization, composition, and proof reduction across multiple mathematical and computational frameworks.

2 Abstraction as Innermost Reduction

In the untyped lambda calculus, β-reduction proceeds by identifying an application whose left-hand side is a lambda, substituting the argument into the body, and repeating until no further reductions are possible. The most intuitive mode of performing this procedure is by focusing first on the *innermost* reducible expression; one resolves the deepest dependency before attempting to interpret the structure in which it is embedded.

This pattern mirrors the phenomenology of abstraction: when a programmer or theorist chooses not to describe a function's implementation, they implicitly acknowledge that the internal calculation has already been conceptually reduced—that it has been moved to a state where its behavior is determined, stable, and non-interfering with the outer layers in which it participates. Abstraction therefore functions as a kind of *deliberate normal form*: the inner term is no longer kept as a live, demanding computation but is replaced by a summarizing boundary.

Under this view, abstraction is evaluation. To abstract a component is to take it to the point of being substitutable, stable, and non-interactive except through well-defined parameters. This is precisely the meaning of reduction: removing internal redexes until what remains can be treated as a primitive unit relative to a higher-order structure.

3 Interfaces, Boxes, and the Logic of Concern

A widespread intuition in software engineering interprets abstraction as "putting things into boxes." A module, a class, a function, a file—these are boundaries that hide the internal details of a computation in order to present a stable surface of interaction for other components. But the deeper reason for this boundary is not representational tidiness; it is a theory of *concern*. Abstraction exists because other programmers, other processes, or other layers of logic should not be required to evaluate details that belong to an inner scope.

Functional languages such as Racket or Haskell formalize this principle by expressing the permissible interactions of a function through a type signature. A type is not a description of what the implementation *does* but a contract for how one may interact with it, ensuring that the reduction of the function body is the internal concern of the function alone. The body is a machine, but the type is the license to treat it as a black box.

This is exactly the form of meaning implied by lambda calculus reduction: once a term is reduced to a value, its internal structure need not be revisited. Abstraction, by this interpretation, is the act of turning a potentially burdensome computational object into a value-like entity, whose stability is asserted by a type, an interface, or a conventional boundary.

4 Substrate-Independence and Null Convention Logic

Abstraction is often presented as substrate independence: one does not care whether a function is implemented in Lisp, C, RISC assembly, or a wetsand mould vibrating at 60 Hz. But this independence is not the negation of computation; rather, it is the affirmation that computation

has already been organized into a pattern that admits many possible substrates without altering its formal behavior.

An instructive parallel arises from *Null Convention Logic* (NCL) and similar asynchronous circuit designs. In these architectures, a signal is represented in a *dual rail* encoding: a bit is true or false only when one of two rails is asserted, and incomplete calculations are represented by the absence of assertion along both rails. This incomplete state is not an error; it is a legitimate mode of being that propagates until the necessary causal dependencies are resolved. Once a computation stabilizes, the dual-rail pattern collapses into a determinate value, which may then be treated as a black-box output for the next stage.

This is abstraction in hardware: the transition from incomplete informational potential to a determinate, composable output. The "box" of abstraction corresponds directly to the discipline of asynchronous stabilization: once a stage has settled, its result is authoritative and can be used without reference to the many microscopic timing behaviors that produced it.

5 Abstraction as Mereological Ascent

To abstract is also to make a mereological move: to shift from the parts to the whole, from the microstructure to the mesostructure, from the implementation to the interface. In set theory this appears as the movement from elements to sets, or from sets to the hierarchies that collect them. In mereology proper, one ascends from the fragments of a system to its larger units, preserving relational structure while discarding irrelevant particulars.

Category theory elevates this idea further: a morphism encapsulates the relationship between objects, and a functor encapsulates the relationship between entire categories. The morphism does not describe the internal constitution of either object; it describes the allowable transformations that connect them. Abstraction thus becomes a movement from the specific to the structural, aligning with the categorical preference for arrows over contents.

The essence is that abstraction never abandons structure; it merely chooses the correct level of structure to preserve. It is an epistemic decision about granularity, a choice of resolution.

6 Curry–Howard and the Execution of Proofs

The Curry–Howard correspondence tells us that programs are proofs and proofs are programs. Under this identification, evaluation in the lambda calculus corresponds to the normalization of proofs: a proof reduces to its essential logical content by eliminating detours, redundancies, and bureaucratic steps.

To abstract, therefore, is to run a proof. When a programmer relies on the type of a function rather than its implementation, they are trusting that the function's internal "proof steps" have been normalized into a concise certificate of behavior. Types function as the logical obligations required to treat a computation as a value, wrapping execution in the same way that a completed proof wraps logical inference.

Thus abstraction is not only a computational act but a logical one: a reduction of the proof term to its canonical role, allowing it to serve as a building block in higher constructions.

7 Conclusion: Abstraction as Computation

The claim developed throughout this essay is that abstraction is computationally and logically identical to reduction. Whether one is performing β-reduction in lambda calculus, emitting a typed interface in Haskell, stabilizing a dual-rail asynchronous circuit, or ascending through set-theoretic or categorical levels of structure, the same phenomenon is occurring: the inner details are being evaluated, normalized, or stabilized in such a way that they may be ignored at a higher level of organization.

To abstract is to compute. To box is to evaluate. To hide is to finish a proof. Abstraction is not the negation of implementation but the marker of its successful completion. It is the moment when a system becomes composable, not because we turned away from its workings, but because those workings have been executed, resolved, and reduced into an object fit for participation in a larger structure.