From Lambda to Ledger: An Architectural Comparison of Plinth and Plutarch

Seungheon Oh
Input Output
USA
seungheon.oh@iohk.io

Ziyang Liu
Input Output
USA
ziyang.liu@iohk.io

Philip Wadler
University of Edinburgh
United Kingdom
Input Output
United Kingdom
philip.wadler@iohk.io

Abstract

Plinth and Plutarch are two languages for writing smart contracts on the Cardano blockchain. Both are based on Haskell, have been used in production over the past few years, and share the same host ecosystem and compilation target. However, they have strikingly different architectures: Plinth compiles a subset of Haskell through deep integration with GHC, while Plutarch represents its programs as explicitly constructed typed terms embedded within Haskell. These choices lead to different balances between abstraction and performance, reuse and specialization, and user experience and implementation effort.

This paper presents an architectural comparison of these two approaches, grounded in the authors' experience implementing and using both languages. We analyze their design trade-offs across multiple dimensions, and present an empirical evaluation on representative programs, examining how architectural choices affect source code, generated target code, and compiler error messages. Our comparison offers broader insights for language design, both within and beyond the blockchain domain.

CCS Concepts: • **Software and its engineering → Functional languages**; **Compilers**; **Domain specific languages**.

Keywords: Language Architecture, Haskell, Domain-Specific Languages, Compiler Plugins, Functional Programming, Smart Contracts

ACM Reference Format:
Seungheon Oh, Ziyang Liu, and Philip Wadler. 2026. From Lambda to Ledger: An Architectural Comparison of Plinth and Plutarch. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '26), August 24–29, 2026, Indianapolis, IN, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3830438.3830956



This work is licensed under a Creative Commons Attribution 4.0 International License.

FUNARCH '26, Indianapolis, IN, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2868-6/2026/08
https://doi.org/10.1145/3830438.3830956

1 Introduction

Smart contract programming poses interesting language design challenges. On the one hand, programmers want expressive, ergonomic, approachable languages with familiar abstractions. On the other hand, smart contracts execute in constrained environments where the size and execution cost of generated code matter a great deal: they directly impact transaction fees, and are bounded by strict limits imposed by the blockchain. Language design for this setting must tread carefully to balance competing trade-offs, like abstraction against predictable performance, reuse against specialization, and user experience against implementation effort.

Within the Cardano ecosystem, a number of high-level languages targeting the same on-chain execution language have emerged over the years, ranging from standalone languages such as Aiken [2026] to languages built on general-purpose host languages; we give an overview in Section 5. Among these, Plinth [Liu et al. 2025] and Plutarch [2026] are two Haskell-based languages, and have been used in production since 2021 and 2022, respectively. They share the same broad goal — enabling developers to write performant and compact Cardano smart contracts in a high-level functional language — and they are built on the same host language, namely Haskell. Yet they involve strikingly different architectural choices.

Plinth aims to preserve the experience of writing ordinary Haskell as far as possible. It reuses a subset of Haskell syntax and compiles through GHC Core, allowing programmers to benefit from familiar language constructs, existing tooling, and a development workflow close to normal Haskell programming. This is achieved through deep integration with GHC — specifically, GHC plugins. Because Plinth and ordinary Haskell programs execute in different environments, getting the best performance from Plinth involves understanding which Haskell idioms translate directly to efficient on-chain code and which need to be adjusted.

Plutarch takes a rather different route. It is an embedded domain-specific language (eDSL) that defines its own abstract syntax in Haskell, and represents on-chain programs as explicitly constructed typed terms. Because Plutarch is essentially a regular Haskell library, it is easier to implement and maintain than a language like Plinth that requires GHC

integration. This design also gives programmers and implementers more direct control over compilation and generated code. The main trade-off is that users work in a more specialized programming model, where they effectively construct Plutarch ASTs explicitly, using types and combinators such as *PInteger*, *plet*, and *pmatch*, rather than ordinary Haskell types, values, and syntax.

These two languages, both using Haskell as the host language, provide an opportunity to study a broader question in language architecture: when building a practical high-level language, particularly for a constrained execution environment, what trade-offs arise between reusing host syntax through integration with the host compiler (like Plinth) and direct embedding into the host language (like Plutarch)? This question is not specific to Cardano or blockchains: similar tensions arise in many other domains.

This paper presents an architectural comparison of Plinth and Plutarch, grounded in our experience developing and using both languages — in particular, Seungheon Oh has been a core maintainer of both languages. After introducing the two languages and their common compilation target (Section 2), we compare them across dimensions such as user-facing design, optimization, metaprogramming, diagnostics, testability, and maintenance effort (Section 3). We complement this architectural analysis with an empirical evaluation (Section 4) using a set of representative programs, studying how the two designs affect source programs, generated code, and compiler diagnostics. We then discuss related work in Section 5, and conclude in Section 6.

The contributions of our work include: (1) a detailed architectural comparison of Plinth and Plutarch across dimensions covering user-facing design, compiler architecture, and implementation complexity; (2) an empirical evaluation on seven representative programs spanning a range of application domains; and (3) a publicly available repository containing the Plinth and Plutarch implementations, test inputs, injected programming errors and the resulting error messages used in our empirical evaluation.

Overall, the two architectures involve interesting trade-offs. Plinth's strengths follow from reuse: familiar syntax, readable code, ordinary Haskell workflows, and an optimizing compiler that makes reasonable decisions automatically. Plutarch's follow from explicitness: programmers have more direct control over generated code, and the language itself is cheaper to build and maintain. Performance-wise, neither language produces uniformly smaller or cheaper code; the two are broadly comparable when contracts are written following each language's best practices.

2 Background

2.1 Programming on Cardano

Cardano uses an extended UTxO (EUTxO) ledger model [Chakravarty et al. 2020] with support for smart contracts. In this model, assets and other data are held in unspent transaction outputs (UTxOs); smart contracts rely upon on-chain *validator* programs to approve or reject actions performed by transactions, such as spending a UTxO, minting tokens, withdrawing rewards, or voting on governance actions. The dominant programming task is therefore to express validation logic over structured inputs, rather than to perform imperative actions that mutate on-chain state.

On-chain validator programs must be deterministic, compact, and cheap to execute. Determinism is required so that all nodes reach consensus on the outcome of a transaction, while compactness and execution cost translate directly into transaction fees and are subject to per-transaction limits set by the blockchain protocol. Validator programs are expressed in Untyped Plutus Core, which we introduce next.

2.2 Untyped Plutus Core and Datatype Encodings

Both Plinth and Plutarch compile to Untyped Plutus Core (UPLC), the low-level language executed by Cardano nodes. UPLC is a small calculus whose typed variant — Typed Plutus Core (TPLC) — is based on System F_ω [Girard 1972]. It is the common compilation target for Cardano smart contract languages. UPLC has a formal specification [Plutus Team 2026], and, together with TPLC, is formalized in Agda [Chapman et al. 2019]. UPLC programs are evaluated by a CEK machine [Felleisen and Friedman 1987] with cost accounting along two dimensions, CPU and memory units, collectively referred to as execution units. A cost model assigns CPU and memory costs to each evaluation step and each built-in function call. Since UPLC is strictly evaluated, explicit *delay* and *force* nodes are used to suspend and resume terms, both of which require evaluation steps and therefore have nonzero cost.

In UPLC, datatypes are typically represented in one of three ways, and both Plinth and Plutarch support all three. We illustrate each using a list, *three* = [1, 2, 3], together with functions *head* and *tail*.

Scott encoding [Abadi et al. 1993], a variant of Church encoding, represents a value as a function taking one continuation per constructor:

$$
\begin{aligned}
\mathit{three} = \ & \textbf{let}\ \mathit{cons} = \lambda x\ \mathit{xs} \to \lambda n\ c \to c\ x\ \mathit{xs} \\
& \quad\ \ \mathit{nil} = \lambda n\ c \to n \\
& \textbf{in}\ \mathit{cons}\ 1\ (\mathit{cons}\ 2\ (\mathit{cons}\ 3\ \mathit{nil})) \\
\mathit{head} = \ & \lambda \mathit{xs} \to \mathit{force}\ (\mathit{xs}\ (\mathit{delay}\ \mathit{error})\ (\lambda y\ \mathit{ys} \to \mathit{delay}\ y)) \\
\mathit{tail} \ = \ & \lambda \mathit{xs} \to \mathit{force}\ (\mathit{xs}\ (\mathit{delay}\ \mathit{error})\ (\lambda y\ \mathit{ys} \to \mathit{delay}\ \mathit{ys}))
\end{aligned}
$$

Sums-of-products (SOP) [Peyton Jones 2023] was introduced later, using *constr* and *case* AST nodes for constructing and matching on values. By avoiding the extra lambda abstractions and applications that Scott encoding requires, SOP is about 30% faster on average, at the cost of adding *constr* and *case* to the AST:

$$\mathit{three} = \mathit{constr}\ 1\ [1, \mathit{constr}\ 1\ [2, \mathit{constr}\ 1\ [3, \mathit{constr}\ 0\ [\,]]]]$$

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# OPTIONS_GHC -fplugin Plinth.Plugin #-}
module Withdraw where
import Plinth.Plugin (plinthc)
import PlutusTx.AsData (asData)
import PlutusTx.Code (CompiledCode)
import PlutusTx.Prelude

asData [d|
  data Withdraw
    = Amount Integer
    | Joint Withdraw Withdraw
    | Deduct Integer Withdraw
    deriving newtype (UnsafeFromData, ToData) |]

netWithdraw :: Withdraw → Integer
netWithdraw (Amount n) = n
netWithdraw (Joint a b) = netWithdraw a + netWithdraw b
netWithdraw (Deduct y from) =
  let x = netWithdraw from in if x ≤ y then 0 else x - y

netWithdrawUPLC :: CompiledCode (Withdraw → Integer)
netWithdrawUPLC = plinthc netWithdraw
```

(a) Plinth

```
module Withdraw where
import GHC.Generics (Generic)
import qualified Generics.SOP as SOP
import Plutarch.Internal.Term (Config (NoTracing), Script, compile)
import Plutarch.Prelude

data PWithdraw s
  = PAmount (Term s (PAsData PInteger))
  | PJoint (Term s (PAsData PWithdraw)) (Term s (PAsData PWithdraw))
  | PDeduct (Term s (PAsData PInteger)) (Term s (PAsData PWithdraw))
  deriving stock (Generic)
  deriving anyclass (SOP.Generic, PIsData)
  deriving PlutusType via DeriveAsDataStruct PWithdraw

netWithdraw :: Term s (PWithdraw :--> PInteger)
netWithdraw = pfix #$ plam $ λself wt → pmatch wt $ λw → case w of
  PAmount n → pfromData n
  PJoint a b → (self # pfromData a) + (self # pfromData b)
  PDeduct y from → plet (self # pfromData from) $ λx →
    plet (pfromData y) $ λy' → pif (x #≤ y') 0 (x - y')

netWithdrawUPLC :: Script
netWithdrawUPLC = case compile NoTracing netWithdraw of
  Right s → s
  Left err → error (show err)
```

(b) Plutarch

Figure 1. An Example Written in Plinth and Plutarch

$$head = \lambda xs \to case\ xs\ [error,\ \lambda y\ ys \to y]$$
$$tail\ = \lambda xs \to case\ xs\ [error,\ \lambda y\ ys \to ys]$$

Here constructor indices 0 and 1 represent *nil* and *cons*. The brackets represent lists in the UPLC AST, not source code.

The third representation, *Data* encoding, uses the built-in *Data* type, a CBOR-based representation that serves as the standard format for validator arguments and on-chain data stored in UTxOs. Because *Data* cannot contain functions, it is strictly less expressive than Scott or SOP, but this is a desirable safety property for user-supplied data. *Data* is also easier for off-chain tooling to construct and inspect.

Besides being less expressive, matching on *Data* is more verbose, requiring a chain of built-in calls such as *listData* and *unListData*, and is thus less efficient than Scott or SOP. Despite this, *Data* is the most widely used encoding in practice, because it has a decisive advantage: it is the format in which validator inputs and on-chain data are represented, so using *Data* encoding avoids the cost of conversion.

In *Data* encoding, the list example becomes:

$$three = List\ [I\,1, I\,2, I\,3]$$
$$head = \lambda xs \to case\ (unListData\ xs)\ [\lambda y\ ys \to y]$$
$$tail\ = \lambda xs \to listData\ (case\ (unListData\ xs)\ [\lambda y\ ys \to ys])$$

2.3 Plinth

Plinth is a high-level language for Cardano smart contracts that reuses a subset of Haskell syntax. Compilation proceeds through GHC's frontend, augmented with source plugins that customize certain behaviors and capture relevant source information; a Core plugin then translates GHC Core into UPLC. Plinth was previously called Plutus Tx, and sometimes informally just Plutus. The earlier name persists in module names such as *PlutusTx.Prelude*, and readers may also encounter it in existing literature, libraries, and tooling.

Figure 1(a) shows a small Plinth example. It defines a recursive datatype, *Withdraw*, and a recursive function, *netWithdraw*, over it. The code is written largely in ordinary Haskell style, using familiar syntax for datatype declarations, case expressions, recursive functions, and arithmetic. Plinth supports most core Haskell features, but advanced features like existential types, type families, and IO are not supported.

The *-fplugin Plinth.Plugin* flag enables the Plinth compiler. It recognizes occurrences of $plinthc :: a \to CompiledCode\ a$, which marks its argument (in this case *netWithdraw*) for compilation to UPLC. The produced UPLC is wrapped in *CompiledCode*. The *NoImplicitPrelude* extension suppresses Haskell's default prelude, and *PlutusTx.Prelude* is imported instead, which mirrors much of the Haskell prelude, but is tailored for reliable and efficient compilation to UPLC. For example, it uses *OPAQUE* pragmas to prevent unwanted GHC inlining, and it favors the use of single-method type classes, which tend to produce more efficient UPLC.

By default, ordinary Haskell datatypes compile to either SOP or Scott encoding. To use *Data* encoding, Plinth provides the Template Haskell helper *asData*. The datatype definition wrapped by *asData*, in this case *Withdraw*, is compiled to the *Data* representation. Using *asData* rather than a compiler flag conveniently allows selecting between *Data* and SOP

encoding on a per-type basis. The choice between SOP and Scott is instead controlled by a compiler flag, since Scott encoding is retained only for backward compatibility — SOP is uniformly more efficient than Scott, so there is no need to select between them on a per-type basis.

2.4 Plutarch

Plutarch is another Haskell-based language targeting UPLC, implemented as an embedded DSL in the usual sense.

Figure 1(b) shows the Plutarch implementation of the same *Withdraw* example. It illustrates Plutarch's eDSL nature clearly: it is a Haskell library where programs are built as Haskell values of an explicit *Term* type rather than as ordinary Haskell expressions. More precisely, a Plutarch program constructs a value of type *Term s a*, where *s* and *a* are phantom types [Cheney and Hinze 2003]: *a* denotes the type of value the term evaluates to, and *s* is a scope parameter that rules out ill-scoped terms in the style of *ST* [Launchbury and Peyton Jones 1994]. As an eDSL, Plutarch provides its own object-language constructs. Function types use the ":-->" arrow; applications use # (left associated) or #$ (right associated); lambdas are introduced with *plam*; pattern matching is performed with *pmatch*; and recursion requires using the fixed-point combinator, *pfix*, explicitly. These identifiers are provided in *Plutarch.Prelude*.

Plutarch's eDSL architecture makes it natural to select datatype encoding on a per-type basis. Users can do so by deriving an appropriate *PlutusType* instance for the desired representation. In Figure 1(b), *DeriveAsDataStruct* selects the *Data* encoding; one can use *DeriveAsSOPStruct* and *DeriveAsScottStruct* to select the SOP and Scott encodings.

Plutarch lowers a *Term s a* to UPLC using *compile*. Unlike Plinth's compilation, which runs inside a GHC plugin, *compile* runs at Haskell runtime and takes the compilation configuration as an ordinary argument. The resulting *Script* wraps a UPLC program, analogous to Plinth's *CompiledCode*.

3 Architecture and Design Trade-offs

In this section we examine Plinth and Plutarch along a number of dimensions on which their architectural choices have visible consequences, both for users of the languages and for those implementing and maintaining them. We omit aspects where the two languages do not meaningfully differ, such as compilation time.[1]

3.1 Surface Syntax, Jargon, and Readability

Plinth's surface syntax is a curated subset of Haskell. Programmers write functions over familiar types like *Integer*, [*Integer*], and *Bool*, and use standard Haskell syntax such as →, ⇒, **case**, and **if-then-else**. The "jargon level" (language-specific vocabulary a programmer must learn) is correspondingly low: a Haskell programmer encountering Plinth code will find it largely familiar, without first learning a large collection of language-specific constructs.

Plutarch takes a different approach. As an eDSL, it is implemented as a Haskell library for constructing and manipulating expressions that are later compiled to UPLC. Plutarch programs are therefore not written as ordinary Haskell programs over ordinary Haskell values; instead, users construct values of the explicit *Term s a* type and work with Plutarch-specific names and operators such as *PInteger*, :-->, *plam*, *pmatch*, *pif*, #==, and #≤. These constructs are natural consequences of the eDSL approach, and they make Plutarch code look substantially different from idiomatic Haskell and increase the specialized vocabulary a user must learn.

Some of these differences could be hidden: for example, Plutarch uses overloading to reuse Haskell operators where the operators' types permit, such as +. Operators like == and ≤, however, cannot be reused since they return host-level *Bool*, necessitating dedicated operators such as #== and #≤. Beyond overloading, Plutarch could in principle reuse certain Haskell syntax via *RebindableSyntax*. Such techniques can make an eDSL appear more Haskell-like, but they are not a complete remedy, since in practice they tend to obscure what the code does and make error messages less intelligible. As Elliott [2017] observes, "additional efforts can hide more of these symptoms, but the cracks still show, and each new coping mechanism leads to increasingly mysterious type errors." The symptoms of embedding can be mitigated, but not eliminated.

This is a usability advantage for Plinth. Readability is not merely an aesthetic concern: as Buse and Weimer [2008] observe, "a consensus exists that readability is an essential determining characteristic of code quality." In the smart contract setting, where code is often audited, reviewed, and optimized under resource constraints, reducing surface-level jargon directly improves developer experience.

3.2 Type Systems

Both languages rely on Haskell's type system, though to different ends.

Plinth's surface language reuses Haskell's type system directly: a Plinth program is type-checked as Haskell, and the resulting GHC Core is then translated to UPLC. Plinth supports a curated subset of Haskell: features such as existential types and type families are unsupported, and identifiers that lack unfoldings, such as those marked *NOINLINE*, often cannot be compiled (except for special ones the compiler knows about). Passing GHC's type checker is therefore necessary but not sufficient for successful Plinth compilation.

Plutarch, being an eDSL, instead uses Haskell's type system to enforce invariants of its embedded object language, such as ruling out ill-scoped terms (Section 2.4). Enforcing

[1] Both languages typically compile a validator within a few seconds, with Plinth usually somewhat slower, likely due to its heavier pipeline (plugin passes, optimizations). In our experience, compilation speed has not been a practical hurdle or an optimization target for developers.

those invariants requires substantial type-level machinery — type families, multi-parameter type classes, undecidable instances, among other extensions. As a result, many programming mistakes are caught statically, but the corresponding type errors can be indirect and difficult to interpret, compounding the cognitive load discussed in Section 3.1. This trade-off is hard to avoid for eDSLs that aim to provide strong static guarantees.

The *a* parameter in *Term s a* represents the type of the underlying object-language term. It is a phantom type and thus can be coerced, including through unsafe coercions. Plutarch exploits coercions internally, and developers can use them as an optimization tool. For example, a value of type *PInteger* that is known to be positive can be coerced into a more refined *PPositive* type without a runtime check.

The two languages also differ in the typed calculi they use on top of UPLC. Plutarch's *Term* can be understood as intrinsically scoped, intrinsically typed simply typed lambda calculus (STLC), extended with base types and an error term. *Term* itself does not support polymorphism, but polymorphism can be expressed through the host language. This does not mean, however, that instantiating a Plutarch function to multiple concrete types would produce multiple copies of identical code in UPLC: *phoistAcyclic* — which hoists a term to a top-level binding — computes the hash of the function body and uses it to deduplicate, effectively performing a form of common subexpression elimination (CSE) at the UPLC level. General recursion and Turing completeness in Plutarch are recovered via a library function *pfix*, which translates to the Z-combinator — a fixed-point combinator suitable for strict evaluation — in UPLC.[2] Plinth, by contrast, compiles through TPLC, a System F_ω-like calculus with base types, iso-recursive types, and an error term, before erasing types to UPLC. TPLC is more expressive than Plutarch's *Term*: for example, it can represent polymorphic types. Adopting a TPLC-style representation instead of *Term* in Plutarch would likely overcomplicate its already complex source language.

3.3 Metaprogramming

Metaprogramming is an important tool for optimization in resource-constrained settings. A useful pattern, for instance, is unrolling: duplicating the body of a recursive function so that the first few iterations avoid the recursion overhead, trading larger script size for lower execution cost.

Both Plinth and Plutarch support metaprogramming, but the architectures make the experience quite different. In Plutarch, a program is a Haskell value representing an object-language term. Metaprogramming therefore reduces to ordinary Haskell: one performs the construction and manipulation of object-language programs in plain Haskell, and no Template Haskell (TH) is involved. For example, Plutarch provides a library function for unrolling recursion a fixed number of steps before falling back to a fixed point:

```
punrollUnbound :: ∀ a b s. Int
  → (Term s (a :--> b) → Term s (a :--> b))
  → Term s (a :--> b)
punrollUnbound 0 f = pfix #$ plam f
punrollUnbound n f = f (punrollUnbound (n – 1) f)
```

This is ordinary recursive Haskell code over Plutarch terms. An unrolled *length* function can be written as:

```
plength :: ∀ a s. Term s (PBuiltinList a :--> PInteger)
plength = punrollUnbound 10 $ λself → plam $ λxs →
  pmatch xs $ λxs' → case xs' of
    PNil → 0; PCons _ ys → 1 + self # ys
```

In Plinth, metaprogramming must instead go through TH. Its counterpart to *punrollUnbound* is the following library function, *peel*:

```
peel :: ∀ a b. Int → (CodeQ (a → b) → CodeQ (a → b))
  → CodeQ (a → b)
peel 0 f = ⟦ fix $ λself → $$(f ⟦self⟧) ⟧
peel n f = f (peel (n – 1) f)
```

Using it requires the step function itself to be written in TH form:

```
length :: ∀ a. [a] → Integer
length = $$(peel 10 $ λself →
  ⟦ λxs → case xs of [] → 0; _ : ys → 1 + $$self ys ⟧)
```

This generates unrolled code equivalent to the Plutarch version, but the programmer must write a special step function, and manage TH quotation, splicing, and staging explicitly. Any resulting type and staging errors can be time-consuming to diagnose. Reusing TH, however, keeps Plinth's metaprogramming within standard Haskell machinery — a familiar setting for experienced Haskell developers.

3.4 Optimization and Control over Generated Code

Plutarch gives programmers relatively direct control over the generated UPLC, on the assumption that programs will be hand-optimized. The *Term* type sits close to the UPLC AST, and compiling it to UPLC is largely syntax-directed. Inlining control, for example, is entirely in the programmer's hands: a Haskell-level **let** binding inlines the bound term at every use site, whereas *plet* (or *phoistAcyclic* for top-level bindings) explicitly introduces the equivalent of a let-binding in UPLC, preventing inlining and avoiding code duplication and re-evaluation. Generally, no significant automatic optimizations are applied: although UPLC-level optimizers exist in the Plinth compiler and could in principle be applied to any UPLC program, they are not part of Plutarch, and are not invoked by default. Programmers instead optimize mainly by structuring source code to produce the desired target code

[2] The Z-combinator contains self-application that has no STLC type; *pfix* uses *punsafeCoerce* to give it the desired type.

directly. The rationale is that even sophisticated automatic optimizations are often insufficient in performance-sensitive settings, making hand-tuning necessary. Plutarch therefore favors predictability and manual control, so that expert users can shape the generated code with precision. This is a design choice of Plutarch, not an inherent property of eDSLs.

Direct control, however, comes with risks. Because Plutarch performs no substantial source restructuring, any sharing not introduced explicitly is absent from the generated UPLC. A Haskell-level binding holding a Plutarch *Term* and used more than once is duplicated syntactically at each use; metaprogramming combinators such as *punrollUnbound* can amplify this into exponential UPLC growth when the step function uses its binder multiple times. Developers therefore often need to cross-reference Plutarch source against the generated code.

Plinth takes the opposite approach and ships with a suite of optimizers from the outset. Its high-level intermediate representation, Plutus IR (PIR), retains datatypes and explicit recursion and is designed to be readable and suitable for inspection, much as performance-oriented Haskell programmers inspect GHC Core. Most optimizations either run on PIR alone or on both PIR and UPLC, so inspecting PIR is usually enough to diagnose inefficiencies.

Over time, Plinth has narrowed the control gap. Programmers can influence important transformations, especially inlining, using mechanisms such as *INLINE* pragmas and a Plinth-specific variant of *GHC.Magic.inline* (a special function that forces inlining at a specific call site). On this particular axis, we believe that Plinth has largely caught up, and that the vast majority of UPLC code expressible in Plutarch is now also expressible in Plinth. The issue is less what Plinth can express than how naturally the efficient version can be written. One example is polymorphism, which is more likely to incur a runtime cost in Plinth. Consider the following *length* definition in Plinth:

$$
\begin{aligned}
&\mathit{length} :: \forall\, a.\, [a] \to \mathit{Integer} \\
&\mathit{length}\ [\,] = 0 \\
&\mathit{length}\ (_ : \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs}
\end{aligned}
$$

Each recursive call carries a type argument, and type abstractions and instantiations compile to *delay*/*force* nodes in UPLC, incurring unnecessary cost. This can be avoided by using an inner, monomorphic *go* function. In Plutarch, the corresponding type variables are instantiated in Haskell, and thus do not translate to *delay*/*force* in UPLC.

3.5 Coding Patterns and Performance Pitfalls

Plinth's Haskell-like surface language is an ergonomic advantage, but it can mislead programmers into expecting Haskell-like performance characteristics. Function and constructor applications in Plinth are strict, and non-strict bindings (defined using ~) are by-name rather than by-need. Combined with a handful of UPLC-specific quirks, this means idiomatic Haskell is not always efficient Plinth.

Consider field access on values encoded with the built-in *Data* type (Section 2.2). Extracting the k-th field is linear in k, since the structure is unpacked one field at a time.[3] When multiple field accesses are needed, it is therefore cheaper to extract them together in a single pattern match than to access each at its point of use.[4] A Haskell programmer has little reason to think this way.

The lack of laziness creates similar issues. If, for example, a large, expensive expression is needed in two of three branches, and used twice in one branch, the code must be structured carefully to avoid duplicating either code or computation. Achieving this can require writing Plinth in a style that departs from idiomatic Haskell.

Plutarch is less prone to these particular pitfalls because its programs do not look like ordinary Haskell to begin with, offering fewer familiar idioms to lean on and pushing programmers to reason directly in terms of the target code. Inefficient Plutarch code can certainly be written, as noted in Section 3.4, but the causes typically stem from a misunderstanding of how Plutarch lowers to UPLC rather than from misapplied Haskell idioms.

3.6 Error Reporting

Error message quality is central to developer experience. Denny et al. [2021] identify several factors that affect the helpfulness of error messages. We focus on the two most directly shaped by language design — line numbers and jargon level — and set aside factors like verbosity and sentence structure, which are primarily engineering concerns.

Line numbers. Plinth's reuse of Haskell syntax and integration with GHC should, in principle, be an advantage. In practice, the bulk of Plinth compilation runs in a GHC Core plugin, and in GHC Core, most source locations other than those attached to top-level bindings have been stripped — type checking is complete and they are no longer needed. When the Plinth compiler encounters a *CoreExpr* (GHC's representation of Core expressions) it cannot translate, it therefore often has no line number to report.

Plinth addresses this with a source plugin that runs after type checking, when source locations are still available. It wraps selected *HsExpr*s (GHC's representation of typechecked source expressions) with an opaque identity function, $\mathit{anchor} :: \forall\, (\mathit{loc} :: \mathit{Symbol})\ a.\ a \to a$. The type-level string *loc* records the location information, and *anchor* has an *OPAQUE* pragma that prevents GHC from inlining the wrapper away before the Core plugin sees it. Because such wrappers can interfere with downstream optimizations, Plinth

[3] Future UPLC versions are expected to add array support to *Data*, enabling constant-time field access.

[4] Common subexpression elimination (CSE) can mitigate this problem, but in our experience, it is difficult for CSE to do a perfect job, especially due to the complex interaction between different optimizations.

inserts them sparingly — typically only on selected variables. On failure, the compiler walks the stack of enclosing *CoreExpr*s from the offending *CoreExpr* until it finds one carrying a location.

The use of *anchor* is a workaround that reflects a GHC limitation rather than an inherent problem with syntax reuse: if GHC supported attaching arbitrary metadata to type-checked *HsExpr*s and preserving it into *CoreExpr*s, *anchor* would be unnecessary. Indeed, Scalus [Scalus 2026] and OpShin [OpShin 2026] intercept Scala and Python at stages that still retain source locations, and need no *anchor*-like mechanism.

In Plutarch, heavy use of type-level machinery catches most programming mistakes at Haskell type-check time, where GHC's normal line-number reporting applies. Errors reported by Plutarch itself, which lack line numbers, can occur when programmers construct raw UPLC directly, for example by circumventing the scope mechanism to introduce a free variable, but this is discouraged and rare in practice.

Both languages work with standard Haskell IDE tooling such as Haskell Language Server, so ordinary GHC errors receive the usual editor support, but errors reported by the languages themselves, common in Plinth but rare in Plutarch, appear only when compilation to UPLC is performed.

Jargon level. When Plinth's Core plugin encounters an unsupported construct, printing the offending *CoreExpr* would often expose unhelpful GHC-internal jargon. To avoid this, Plinth uses another source plugin that scans the type-checked module for unsupported source-level constructs, and wraps each with an *unsupported* marker, analogous to *anchor* but recording the construct used. The Core plugin then fails on this marker with a clear source-level message. The source plugin cannot fail as soon as an unsupported construct is encountered, because modules may mix ordinary Haskell and Plinth code, and unsupported constructs only matter inside Plinth code targeted for compilation to UPLC. Unlike *anchor*s, these *unsupported* wrappers do not raise the same optimization concerns, since if one is encountered, compilation is aborted.

Plutarch's heavy type-level machinery and jargon-heavy surface language both degrade error message readability. Custom errors using *GHC.TypeError* help in some cases, but many common errors remain difficult for inexperienced programmers to interpret. Nevertheless, since most Plutarch errors are GHC type errors rather than Plutarch's custom errors, they are familiar to experienced Haskell developers.

3.7 Program Inspection and Debugging

After a program compiles, developers need ways to inspect and debug its behavior. There are several aspects of this.

Running as Haskell. Every Plinth program is also a valid Haskell program. It can run as ordinary Haskell and be loaded into GHCi, and in most practical cases the behavior matches Plinth's, especially when the *Strict* GHC extension is used. This is useful for diagnosing correctness issues with the ordinary Haskell debugging workflow. For performance diagnosis, however, running as Haskell is of little help.

Unlike a Plinth program, a Plutarch *Term* is not itself a Haskell program one can debug at the source level. Running it means compiling to UPLC and evaluating the result on the CEK machine. This can still be done interactively from GHCi, but Haskell's source-level debugging workflow no longer applies. One possible improvement would be to abstract the term representation, yielding a type such as *repr s a*, where *repr* can be instantiated to either the existing *Term* for UPLC compilation, or something akin to *Identity* to recover an ordinary Haskell program. Such an alternative interpretation would need to faithfully match the CEK machine's evaluation semantics, which can be non-trivial.

Tracing. Both languages support inserting trace messages into UPLC programs, printed when the corresponding term is evaluated. Plinth's plugin architecture, however, enables a richer form of tracing since the plugin has direct access to identifier names and source locations. Plinth takes advantage of this by exposing compiler flags for automatically instrumenting compiled programs, emitting a trace at every function call with the callee's name and location. Such source-aware instrumentation is harder to provide in Plutarch.

Inspecting generated code. Inspecting generated UPLC is important for diagnosing performance issues. This is feasible in both languages, though the workflows differ. Plinth users typically inspect PIR, which retains datatypes and explicit recursion and is designed to be readable, much like GHC Core for Haskell. Plutarch, as Section 3.4 explains, has no optimizer passes that radically restructure the program, so the generated code is more predictable from the source itself. And since Plutarch compilation does not involve GHC plugins, it can be invoked interactively from GHCi, enabling a fast inspect-UPLC-and-adjust workflow for manual optimization.

3.8 Testability

Both languages support convenient testing with Haskell's standard testing frameworks. Testing Plutarch code requires nothing special: Plutarch is an ordinary Haskell library for constructing and manipulating object-language expressions. Plutarch programs are therefore just Haskell values, so tests can be written as ordinary Haskell tests over those values.

For Plinth, the GHC Core plugin transforms Plinth code into a Haskell value of type *CompiledCode*, which wraps the generated UPLC; ordinary Haskell code is compiled normally and is unaffected by the plugin. Plinth and Haskell code can therefore be freely mixed within a module, and test code that inspects or evaluates a Plinth program is just Haskell code consuming a *CompiledCode* value.

This ability for Plinth and Haskell code to easily coexist is one motivation for choosing the plugin architecture over the alternative of building a standalone Plinth compiler using GHC as a library. It also enables a Cardano application to be

Table 1. Validator Corpus

Validator	Script Purpose	Original Language	Contract Type
Smart Tokens [Input Output 2025]	Rewarding	Plutarch	ERC-20-style programmable tokens
Governance Guardrail [IntersectMBO 2024]	Proposing	Plinth	Guardrails for governance proposals
Rosetta Crowdfund [Blockchain@Unica 2026]	Spending	Aiken	Fundraising with refund
Rosetta Vesting [Blockchain@Unica 2026]	Spending	Aiken	Time-locked release of funds
SundaeSwap NFT [SundaeSwap Finance 2025]	Minting	Aiken	Minting of authorizing NFT
SundaeSwap Certifying [SundaeSwap Finance 2026]	Certifying	Aiken	Vote-delegation certificate validation
Credential Manager Voting [IntersectMBO 2026]	Voting	Plinth	Hot committee credential script

developed entirely in Haskell, with on-chain validation logic written in Plinth and the surrounding off-chain code written in ordinary Haskell. A standalone compiler would be more modular and more customizable, but would make it harder to mix Plinth and Haskell code in the same project. The plugin approach also fits Plinth's broader design goal better: keeping development as close as possible to ordinary Haskell, without introducing a separate compiler or toolchain.

Both approaches also contrast with standalone DSLs — entirely new languages with their own compilers. While they can be tailored more precisely to their purposes, their implementers must build not just a full compiler but also testing frameworks, metaprogramming machinery, IDE integration, and so forth. Plinth and Plutarch inherit all of these for free.

3.9 Implementation and Maintenance Effort

Implementation and maintenance effort is where Plutarch has a clear advantage. An eDSL is, at the end of the day, a Haskell library, and writing a Haskell library — however much advanced type-level machinery it employs — is substantially easier than integrating deeply with GHC through compiler plugins. The reason is largely sociological rather than technical: while writing compiler plugins is not inherently difficult, comparatively few developers write them, and the ecosystem effect is correspondingly thin. As a result, APIs exposed by compilers for plugins and other forms of integration may be incomplete, unstable, and user-unfriendly.

This manifests in a few concrete ways for Plinth. Supporting a new major GHC version is non-trivial: the GHC API shifts between releases, and different GHC versions produce subtly different Core for the same source program, complicating the test suite. Plinth also requires its own standard library, mirroring much of Haskell's *base*, because GHC's inlining and specialization behavior is difficult to control precisely, making parts of *base* hard to compile reliably as Plinth. Reimplementing the relevant pieces in a controlled standard library sidesteps the problem.

Targeted improvements to GHC itself could narrow these gaps, but are unlikely to eliminate them entirely. A language like Plinth is therefore best developed in close collaboration with engineers experienced in GHC internals, which is fortunately the case at Input Output given its long-standing ties to the GHC community.

4 Evaluation

4.1 Setup

We evaluate Plinth and Plutarch on seven validators drawn from production contracts and public benchmarks, summarized in Table 1. Together they span all six Cardano script purposes — Spending, Minting, Rewarding, Certifying, Voting, and Proposing — and a range of application domains including token management, decentralized exchange, governance, crowdfunding, vesting, stake-credential certification, and committee voting.

For each validator, we compare equivalent Plinth and Plutarch implementations. When a validator was originally written in either Plinth or Plutarch, we ported it to the other; when it was written in neither, we implemented both versions. In all cases, both implementations were lightly optimized to reflect current language features and general best practices, but not aggressively hand-tuned by closely inspecting target code or writing low-level source code. For Plinth, this mainly means relying on the standard optimizer while structuring code to avoid known pitfalls such as those mentioned in Sections 3.4 and 3.5; for Plutarch, it mainly involves managing sharing and inlining properly with **let**, *plet*, or *phoistAcyclic* (Section 3.4), and using lower-level data representations where they avoid conversions. The implementations are available publicly.[5] All measurements use Plinth 1.65 and Plutarch 1.12.

We organize the evaluation around three groups of metrics: properties of the source program (Section 4.2), the generated UPLC (Section 4.3), and compiler error messages (Section 4.4). Ratios reported in this section are Plinth divided by Plutarch, so values below 1× indicate that Plinth is smaller or faster.

4.2 Source Programs

We measure two source-level properties: program size and the amount of language-specific vocabulary.

Source program size. We report effective lines of code (eLOC), following Kosar et al. [2008], counting all lines that are neither blank nor parenthesis-only. This is a reasonable proxy for the amount of source code a programmer must write, review, and maintain. All programs are formatted with Fourmolu [Parsons 2025] before counting.

[5] https://github.com/seungheonoh/plinth-plutarch-comparison

Table 2. Source Program Size and Serialized UPLC Size

	Source (eLOC)			UPLC (bytes)		
Validator	**Plinth**	**Plutarch**	**Ratio**	**Plinth**	**Plutarch**	**Ratio**
Smart Tokens	577	985	0.59×	3902	4080	0.96×
Guardrail	89	202	0.44×	798	769	1.04×
Crowdfund	211	303	0.70×	2203	1888	1.17×
Vesting	155	199	0.78×	1184	1219	0.97×
SundaeSwap NFT	220	345	0.64×	2475	2162	1.14×
Certifying	36	52	0.69×	381	317	1.20×
Voting	58	55	1.05×	244	272	0.90×

Table 3. Source Program Jargon Level

	Jargon Count			Jargon Density	
Validator	**Plinth**	**Plutarch**	**Ratio**	**Plinth**	**Plutarch**
Smart Tokens	350	1724	0.20×	0.14	0.36
Guardrail	44	266	0.17×	0.06	0.30
Crowdfund	112	491	0.23×	0.11	0.34
Vesting	89	295	0.30×	0.10	0.31
SundaeSwap NFT	106	576	0.18×	0.09	0.33
Certifying	22	112	0.20×	0.07	0.31
Voting	46	156	0.29×	0.12	0.37

The left half of Table 2 reports eLOC measurements for the seven validators. Plinth is more compact than Plutarch on six of the seven, with per-validator ratios spanning 0.44× to 1.05×. This gap follows from the architectures: Plinth reuses ordinary Haskell syntax, whereas Plutarch explicitly constructs object-language terms with combinators such as *plam*, *pfix*, *pif*, and *pmatch*, which are more verbose than their native Haskell counterparts. Type signatures that fit on one line in Plinth may span multiple lines in Plutarch, and uses of *punsafeCoerce* require additional type annotations.

Source program jargon level. We approximate cognitive overhead by counting language-specific identifiers and operators, such as *plinthc* and *asData* for Plinth, and *plet*, *pmatch*, and *#==* for Plutarch. This is similar in spirit to the "Identifier Terms in Dictionary" (ITID) measure of Scalabrino et al. [2018]: both metrics quantify how far a program's vocabulary departs from familiar terms.

Table 3 reports both absolute jargon counts and jargon density (jargon as a fraction of all tokens). The gap is large and uniform across the board, reflecting the architectural differences discussed earlier: Plinth reuses ordinary Haskell syntax for binding, pattern matching, recursion, and arithmetic, while Plutarch uses many dedicated DSL identifiers and operators that build object-language terms explicitly.

4.3 Generated UPLC Code

We now examine the generated UPLC, measuring both code size and execution cost.

UPLC size. We measure the size of generated UPLC code by its serialized size in bytes. This is the form in which scripts are submitted to the blockchain, and the serialized size is used to calculate transaction fees and determine whether it fits within Cardano's script size limit.

Table 4. Plinth/Plutarch execution cost ratios. Each cell shows the geometric mean, plus the min–max range. The Cases column gives the number of inputs we used for each validator.

Validator	**Cases**	**CPU ratio**	**Memory ratio**
Smart Tokens	8	1.09× (1.04–1.16)	1.15× (1.10–1.24)
Guardrail	5	1.09× (1.03–1.13)	1.16× (1.09–1.24)
Crowdfund	6	0.78× (0.57–0.94)	0.87× (0.67–1.01)
Vesting	11	0.88× (0.85–0.91)	0.87× (0.85–0.90)
SundaeSwap NFT	23	1.03× (0.98–1.28)	1.19× (1.13–1.39)
Certifying	4	0.98× (0.91–1.09)	0.99× (0.94–1.11)
Voting	4	0.93× (0.91–0.94)	0.89× (0.85–0.90)

The right half of Table 2 reports the results. Overall, Plinth and Plutarch produce similarly sized UPLC. Plinth produces smaller UPLC for three and larger UPLC for the other four, with ratios ranging from 0.90× to 1.20×.

Several factors explain these differences: Plinth's optimizers generally do a reasonable job, and compiler flags offer some control over size/runtime trade-offs, for example by tuning inlining aggressiveness. On the other hand, polymorphism can lead to increased code size for Plinth (Section 3.4). Plutarch also allows more precise control — for example, while Plinth can force inlining, it provides no way to prevent inlining other than disabling the inliner entirely.

UPLC execution cost. We measure execution cost in terms of CPU and memory units (Section 2.2) by evaluating each generated UPLC program on representative inputs covering the validator's success cases. These inputs are drawn from the scripts' repositories, augmented where the provided tests do not cover all validation paths. Since validator outcomes on Cardano are predictable off-chain, users normally never submit transactions that fail to validate, so failing inputs are omitted. Table 4 reports the cost ratios. As it shows, when equivalent contracts are written following each language's best practices, performance is broadly comparable, and neither language is uniformly dominant.

The factors discussed above also apply here. Plinth's inliner and other optimizations generally make reasonable choices automatically, whereas Plutarch users manage inlining themselves — using **let** to inline, or *plet* and *phoistAcyclic* to prevent inlining — which can be challenging to get right without significant tuning. Conversely, Plutarch can avoid some costs that Plinth's compilation introduces. For example, polymorphism may incur a runtime cost in Plinth but almost never in Plutarch, and fine-grained control over the generated UPLC remains easier in Plutarch, where source programs map more directly to target code.

The differences between individual validators appear to be driven by these local factors rather than by a validator's script purpose or application domain. From this corpus of

seven validators, we do not infer that particular classes of validators are intrinsically better suited to either language. Note also that our implementations are not aggressively optimized (Section 4.1). Heavily hand-tuned scripts may show different performance characteristics; we leave a comparison of the two languages under aggressive optimization, including the effort it requires to do so, as future work.

4.4 Compiler Error Messages

As discussed in Section 3.6, we focus on the two factors most directly shaped by language design: jargon level and line numbers.

Studies such as Tirronen et al. [2015] have reported common mistakes made by Haskell programmers, especially beginners. We do not use those mistakes directly, because GHC handles generic Haskell errors similarly for both languages, and such errors are not the main learning difficulties. Instead, we identify three categories of common, realistic, and language-specific mistakes for each language, and introduce between three and eight variants of each into the validators.

The Plinth categories are: (1) use of unsupported Haskell constructs, such as GADTs, the *Int* type (Plinth allows *Integer* but not *Int*), or *base* definitions like ==, where one should instead use the equivalent from Plinth's standard library; (2) stage violation: the Plinth compiler attempts to compile a value known only at runtime, such as $\lambda x \to \mathit{plinthc}\,(\ldots x \ldots)$; and (3) misuse of compiler flags and pragmas, for instance a non-existent plugin flag. The Plutarch categories are: (1) choosing a *PlutusType* derivation strategy that does not match the datatype's shape, such as using a single-constructor strategy for a multi-constructor type (this is specific to Plutarch, since Plinth does not expose such choices); (2) confusing Haskell-level and Plutarch-level identifiers, e.g., using == where #== is required; and (3) omitting conversions between values and their *Data*-encoded forms at representation boundaries, a mistake common in Plutarch because *Data*-encoded values must be unwrapped explicitly.

Error message jargon level. We measure jargon as in Section 4.2, but here we distinguish two subcategories. *DSL-specific* jargon is the same language-specific vocabulary measured for source programs. *Compiler-internal* jargon covers tokens used internally by the compiler that do not appear in the source code, such as GHC Core expressions; these are even less helpful than DSL-specific vocabulary. Table 5 reports the average number of jargon tokens emitted per failed compilation in each category.

Plinth's compiler-internal jargon is high for the first category — unsupported Haskell constructs. When the Core plugin encounters a Core expression it cannot compile, it prints the offending GHC Core expression, along with any Core expressions leading up to it until one with a line number is found, which can lead to a high amount of compiler-internal text. On the positive side, for most common errors in this category, it is able to give an informative reason, especially for those caught by the source plugin (Section 3.6). The other two Plinth categories are less jargon-heavy: they either fail before any Core is compiled at all (e.g., in the case of using a non-existent plugin flag), or fail with a short message.

Table 5. Error Message Jargon Level

	Error category	DSL-specific	Compiler-internal
Plinth	Unsupported Haskell constructs	0.4	77.5
	Stage violation	1.0	3.0
	Flag and pragma misuse	0.0	2.0
Plutarch	PlutusType derivation mismatch	4.8	6.0
	Host/object identifier confusion	109.6	0.0
	Missing *Data* conversion	31.8	0.0

Plutarch's failures, by contrast, are almost always pure GHC type errors, and therefore carry fewer compiler-internal tokens. The messages do still name the Plutarch types involved, driving up the DSL-specific count. Host/object identifier confusion produces especially verbose, repetitive messages, while *Data* conversion mistakes lead to jargon-heavy messages because the missing wrapping and unwrapping conversions involve *Term*, *PAsData*, and related types.

Line-number accuracy. We evaluate line-number accuracy only for Plinth. As noted previously, Plutarch's compilation errors are almost always GHC type errors with the usual GHC line numbers, which are generally accurate.

Each Plinth error is scored on a 0–4 rubric. A score of 4 means the reported location points to the exact token the user must edit; 3 means it points to a related but inexact location; 2 means it points to library code; 1 means it points to generated code; and 0 means no line number is reported.

Table 6. Error Message Line-Number Accuracy (Plinth)

Error category	Score
Unsupported Haskell constructs	2.4
Stage violation	4.0
Flag and pragma misuse	0.0

Stage violations score perfectly, since the errors point exactly to the variable that is not known at compile time. Flag and pragma misuses score lowest because GHC passes plugin flags to the plugin as strings with no attached locations. The errors in the first category — unsupported Haskell constructs — have mixed scores: although Plinth uses source plugins to detect unsupported constructs and preserve source locations, they do not cover every case, so an error may, for instance, be reported at a variable occurrence even though the actual unsupported construct appears inside its unfolding.

5 Related Work

5.1 Cardano Smart Contract Languages

Several other languages target UPLC besides Plinth and Plutarch. Aiken [2026] is a standalone, purpose-built Cardano language with its own compiler, language server, and

testing framework, and is currently the most widely adopted Cardano smart contract language. Its success can be attributed in large part to its approachable syntax, simple tooling, good documentation, and strong attention to developer experience. Most of these are not inherent to being a standalone language, and are practices Plinth and Plutarch can and should adopt, independent of architecture. Helios [2026] is another standalone language; its compiler is distributed as a JavaScript/TypeScript library, allowing Helios source to be conveniently embedded in browser-based applications. Plu-ts [2026] is a TypeScript eDSL that inherits many of the strengths and weaknesses of Plutarch discussed in Section 3. Scalus [2026] and OpShin [2026] are syntax-reusing languages like Plinth; they reuse subsets of Scala and Python syntax, respectively. Marlowe [Seijas et al. 2020] is a restricted, non-Turing-complete language for financial contracts that compiles to Plinth; its small fixed set of constructs enables more exhaustive static analysis.

5.2 Smart Contract Languages for Other Blockchains

Bartoletti et al. [2025] survey languages across major blockchain platforms, including Aiken for Cardano. The survey identifies high-level differences in ledger models, programming styles, and platform features, but does not pursue the kind of deeper language architecture analysis presented in Section 3. Our comparison instead fixes the ledger model (Cardano), execution target (UPLC), and host ecosystem (Haskell), and studies two language architectures in depth, from the perspectives of both language developers and users.

As their survey highlights, most other major platforms are account-based with procedural contract languages, whereas Cardano's UTxO model leads to an approval style in which validators approve or reject transactions (Section 2.1).

5.3 Embedded DSLs

Embedded DSLs [Hudak 1996] are commonly classified as either shallow or deep embeddings [Boulton et al. 1992]. Deep embeddings construct explicit ASTs; shallow embeddings are built from overloaded functions. Gibbons and Wu [2014] show that the two are more closely related than they appear: a shallow embedding corresponds to the algebra of a fold over a deep embedding.

Plutarch is deeply embedded. Deep embeddings are well suited to code-generating eDSLs, since the final output is itself an AST, and is therefore naturally produced from a higher-level AST. They are also far more amenable to program optimization and metaprogramming. Shallow embeddings are better suited to non-code-generating DSLs that function as ordinary libraries, such as pretty-printing libraries or property-based testing frameworks.

5.4 Syntax-Reusing DSLs

Plinth is a syntax-reusing DSL: it reuses a subset of Haskell syntax, with compilation performed by a GHC plugin. A key difference from an eDSL is that a syntax-reusing DSL reuses the host language's AST rather than defining its own as a host-language library.

There are alternative methods for implementing syntax-reusing DSLs. One option is a standalone compiler built on top of the host compiler's API. GHC, for example, exposes compiler APIs as a Haskell library. Clash [Baaij et al. 2010] takes this approach: it uses GHC as a library to compile a subset of Haskell into hardware description languages. OpShin similarly uses a standalone compiler, implemented on top of Python's *ast* module. As explained in Section 3.8, we instead compile Plinth via a GHC plugin, both to make it easy to mix Plinth and Haskell code and to keep the development workflow close to ordinary Haskell.

Another approach is quoted DSLs, or QDSLs [Najd et al. 2016], which manipulate the host language's surface AST through quotation-based metaprogramming. This introduces syntactic overhead and staging concerns for users. Surface ASTs are also typically much larger and harder to work with than an intermediate representation like GHC Core.

5.5 Comparative Studies of Language Design

Our comparison of Plinth and Plutarch is close in spirit to work that evaluates language designs not only by their formal properties, but also by their effects on implementers and users. Kosar et al. [2008] compare ten approaches to DSL implementation using criteria such as implementation effort and end-user productivity. We follow the same general philosophy, but study two mature languages in the same production ecosystem, targeting the same low-level language. This fixed setting enables a deeper architectural comparison than a broad DSL survey.

Several of our evaluation criteria are motivated by empirical work on readability and diagnostics. Buse and Weimer [2008] propose a metric for software readability and show that it correlates with software quality. Scalabrino et al. [2018] argue that readability depends also on textual features such as identifier vocabulary; and Denny et al. [2021] study the readability of programming error messages. Together, they motivate our treatment of surface syntax, jargon level, and line numbers as key design concerns.

Finally, the tension between Plinth's automatic optimization and Plutarch's manual control echoes the compiler autotuning literature [Ashouri et al. 2018], which both reinforces Plutarch's premise that generic, automatic optimizers often fall short of careful hand-tuning, and suggests directions Plinth could draw on to narrow the gap.

6 Conclusions

Plinth and Plutarch offer two Haskell-based paths to the same goal: writing efficient, compact smart contracts for Cardano. Plinth prioritizes familiarity and reuse by compiling a subset of Haskell through GHC; Plutarch prioritizes

explicit control by representing programs as typed terms in a deep embedding. This architectural difference shapes nearly every aspect of the two languages, and neither approach is uniformly superior. In brief, Plinth is the better choice when familiarity, readability, conventional Haskell workflows, and automatic optimization matter most; Plutarch is preferable when fine-grained control over code generation or a lower implementation and maintenance effort is more important.

These two languages make for a revealing comparison because they share the same host ecosystem, compilation target, and ledger model. Comparing them isolates architectural trade-offs that may be obscured in a broader context. The resulting lessons offer guidance to language designers facing similar architectural choices in other domains, and to developers choosing between languages.

References

Martín Abadi, Luca Cardelli, and Gordon D. Plotkin. 1993. Types for the Scott Numerals. https://api.semanticscholar.org/CorpusID:126166954

Aiken. 2026. Aiken | The modern smart contract platform for Cardano. https://aiken-lang.org/.

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. doi:10.1145/3197978

Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *Euromicro Conference on Digital System Design*. 714–721. doi:10.1109/DSD.2010.21

Massimo Bartoletti, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, Alvise Spanò, Viacheslav Tkachenko, Roberto Tonelli, and Roberto Zunino. 2025. Smart contract languages: A comparative analysis. *Future Gener. Comput. Syst.* 164, C (March 2025), 19 pages. doi:10.1016/j.future.2024.107563

Blockchain@Unica. 2026. Rosetta Smart Contracts. https://github.com/blockchain-unica/rosetta-smart-contracts.

Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *International Conference on Theorem Provers in Circuit Design*. 129–156.

Raymond P.L. Buse and Westley R. Weimer. 2008. A metric for software readability. In *International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, 121–130. doi:10.1145/1390630.1390647

Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. 2020. The Extended UTXO Model. In *Financial Cryptography and Data Security*. Springer, 525–539. doi:10.1007/978-3-030-54455-3_37

James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction (MPC) 2019*. Springer, 255–297. doi:10.1007/978-3-030-33636-3_10

James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types*. Technical Report CUCIS TR2003-1901. Cornell University.

Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *CHI Conference on Human Factors in Computing Systems (CHI '21)*. ACM, Article 55, 15 pages. doi:10.1145/3411764.3445696

Conal Elliott. 2017. Compiling to Categories. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 27 (Aug. 2017), 27 pages. doi:10.1145/3110271

Matthias Felleisen and Daniel P. Friedman. 1987. Control Operators, the SECD-Machine, and the λ-Calculus. In *Formal Description of Programming Concepts*. 193–222.

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *International Conference on Functional Programming (ICFP '14)*. ACM, 339–347. doi:10.1145/2628136.2628138

Jean-Yves Girard. 1972. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. *PhD Thesis, Université de Paris VII* (1972).

Helios. 2026. Helios Lang. https://github.com/HeliosLang/compiler.

Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (Dec. 1996), 196–es. doi:10.1145/242224.242477

Input Output. 2025. CIP-143 Reference Implementation. https://github.com/input-output-hk/wsc-poc.

IntersectMBO. 2024. Cardano Constitution. https://github.com/IntersectMBO/plutus/tree/master/cardano-constitution.

IntersectMBO. 2026. Constitutional Committee Credential Management System. https://github.com/IntersectMBO/credential-manager.

Toma Kosar, Pablo E. Martínez López, Pablo A. Barrientos, and Marjan Mernik. 2008. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology* 50, 5 (April 2008), 390–405. doi:10.1016/j.infsof.2007.04.002

John Launchbury and Simon L Peyton Jones. 1994. Lazy Functional State Threads. In *Programming Language Design and Implementation (PLDI '94)*. ACM, 24–35. doi:10.1145/178243.178246

Ziyang Liu, Kenneth MacKenzie, Roman Kireev, Michael Peyton Jones, Philip Wadler, and Manuel Chakravarty. 2025. Plinth: A Plugin-Powered Language Built on Haskell (Experience Report). In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium (Haskell '25)*. ACM, 30–37. doi:10.1145/3759164.3759347

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old Is New Again: Quoted Domain-Specific Languages. In *Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, 25–36. doi:10.1145/2847538.2847541

OpShin. 2026. OpShin: A simple pythonic programming language for Smart Contracts on Cardano. https://github.com/OpShin/opshin.

Matt Parsons. 2025. Fourmolu: A formatter for Haskell source code. https://github.com/fourmolu/fourmolu.

Michael Peyton Jones. 2023. Sums-of-products in Plutus Core. https://cips.cardano.org/cip/CIP-85.

Plu-ts. 2026. Plu-ts. https://github.com/HarmonicLabs/plu-ts.

Plutarch. 2026. Plutarch. https://github.com/Plutonomicon/plutarch-plutus.

Plutus Team. 2026. Formal Specification of the Plutus Core Language. https://github.com/IntersectMBO/plutus/tree/master/doc/plutus-core-spec.

Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (June 2018), e1958. doi:10.1002/smr.1958

Scalus. 2026. Scalus - DApps Development Platform for Cardano. https://scalus.org/.

Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon J. Thompson. 2020. Marlowe: Implementing and Analysing Financial Contracts on Blockchain. In *Financial Cryptography and Data Security (LNCS, Vol. 12063)*. Springer, 496–511. doi:10.1007/978-3-030-54455-3_35

SundaeSwap Finance. 2025. SundaeSwap Contracts. https://github.com/SundaeSwap-finance/sundae-contracts.

SundaeSwap Finance. 2026. SundaeSwap Treasury Funds. https://github.com/SundaeSwap-finance/treasury-contracts.

Ville Tirronen, Samuel Uusi-Mäkelä, and Ville Isomöttönen. 2015. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming* 25 (2015), e11. doi:10.1017/S0956796815000179

Received 2026-05-31; accepted 2026-06-17