Plinth: A Plugin-Powered Language Built on Haskell (Experience Report)

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

Kenneth MacKenzie
Input Output
United Kingdom
kenneth.mackenzie@iohk.io

Roman Kireev
Input Output
United Kingdom
roman.kireev@iohk.io

Michael Peyton Jones
Input Output
United Kingdom
michael.peyton-jones@iohk.io

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

Manuel Chakravarty
Input Output
The Netherlands
manuel.chakravarty@iohk.io

Abstract

The Cardano blockchain is the first to use proof of stake, offers native support for multiple currencies and is evolving toward a distributed governance model. It supports smart contracts through Plutus, a language based on System F_ω with recursion. About half a dozen languages compile into Plutus, the first of which is *Plinth* (formerly Plutus Tx) — a language that reuses a subset of the Haskell syntax, and has been in commercial use since 2021.

Our journey building Plinth has been unconventional in a number of ways. First, Plinth programs are written in a subset of Haskell, using standard Haskell syntax and types, which brings a number of advantages. Second, compilation is primarily handled by a GHC plugin, one of the most intricate we are aware of. Third, while some compiler optimizations mirror those in Haskell, others diverge significantly to meet on-chain execution constraints. Fourth, Plutus programs run on an instrumented CEK machine with a formal specification in Agda. This report reflects on our design choices, outlining effective practices, challenges, and key takeaways, with an emphasis on recent advances in the language, compiler, and runtime.

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

Keywords: GHC Plugin, Lambda Calculus, Domain-Specific Language, Abstract Machine, Blockchain, Cost Model

ACM Reference Format:
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), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3759164.3759347

Haskell '25, Singapore, Singapore

ACM ISBN 979-8-4007-2147-2/25/10
https://doi.org/10.1145/3759164.3759347

1 Introduction

Blockchains and smart contracts offer a rich and distinctive space for programming language design. Different blockchain platforms differ in ledger model, execution semantics, and contract capabilities, leading to diverse constraints on language features and semantics. As a result, numerous domain-specific languages have emerged across the ecosystems.

Even within Cardano alone, over half a dozen smart contract languages exist. They can be grouped broadly into new languages, embedded domain-specific languages (eDSLs), and subsets of existing languages. Plinth belongs to the last category — a language that reuses a subset of Haskell syntax for authoring smart contracts.

There are some key distinctions between Plinth and traditional eDSLs. First, Plinth users write standard Haskell code using familiar syntax. Instead of constructing ASTs explicitly, as is typical for eDSLs, we reuse Haskell's Core AST, and compile it into the target code. Second, rather than programs being interpreted or compiled at runtime, compilation happens mostly when the Haskell program is compiled by GHC, via a GHC Core plugin.[1]

Compared to traditional eDSLs, Plinth avoids unfamiliar syntactic constructs that are not part of regular Haskell and are specifically designed to work with each individual eDSL. This aligns with Conal Elliott's observations [11]: eDSLs often force users to work with expression types (e.g., *Expr Int*) instead of plain values (e.g., *Int*), rely heavily on overloading, and do not support pattern matching or branching; efforts to hide these symptoms can lead to obscure type errors. Cheney et al. [10] have drawn similar conclusions. The rise of AI tooling could further widen the gap in learning curves: LLMs are reasonably effective at generating and explaining simple

[1] There is a mechanism for lifting runtime values into target code, which we'll explain in Section 2.2.

```
anyLessThan10 ::[Integer] →Bool
anyLessThan10 =any (<10)

any ::∀a. (a →Bool) →[a] →Bool
any f =go where
    go []=False
    go (x:xs) =f x ||go xs
```

(a) Plinth

```
(λs →s s)
  (λs ds →
    case
      ds
      [False
      , (λx xs →
          force
            (force ifThenElse
              (lessThanInteger x 10)
              (delay True)
              (delay (s s xs))))
      ]
  )
```

(c) UPLC

```
letrec
  data (List ::∗ →∗) a |List_match where
    Nil :List a
    Cons :a →List a →List a
in
letrec
  !go :List integer →Bool =
    λ(ds :List integer) →
      List_match
        {integer}
        ds
        {bool}
        False
        (λ(x :integer) (xs :List integer) →
          ifThenElse
            {all dead. bool}
            (lessThanInteger x 10)
            (Λdead →True)
            (Λdead →go xs)
            {all dead. dead})
in
go
```

(b) PIR

Figure 1. A Plinth program and the corresponding PIR and UPLC

general-purpose code, but their performance tends to decline with bespoke DSLs and niche languages [16].

Compared to building a new standalone language, implementing Plinth is considerably easier. We reuse not only GHC's frontend components, but also Haskell's build tools, debugger, testing frameworks, and metaprogramming machinery. Each of these could be a substantial task for developers of a new language. For users, onboarding is smoother thanks to existing Haskell learning resources, and the skills gained are transferable beyond blockchain development.

This approach is unfortunately not without tradeoffs. We'll elaborate more on the challenges encountered in developing Plinth in Section 6, including the semantic mismatch with Haskell's lazy evaluation model, difficulties controlling GHC's optimizations and generating good error messages. Some of these issues are relatively superficial and could be resolved with targeted engineering effort; others are more fundamental to our design choices.

The rest of the report is organized as follows. We describe the compilation pipeline from Plinth to Untyped Plutus Core (UPLC) in Section 2, and discuss compiling data types in Section 3. In Section 4 we compare our optimization strategies to those of GHC. Section 5 covers the UPLC evaluator and costing. Section 6 presents broader lessons learned. Section 7 reviews related work, and Section 8 concludes the report with a discussion on future directions.

2 Plinth and the Compilation Pipeline

Plinth supports a subset of Haskell's syntax and is tailored for Cardano's EUTxO ledger model [7]. Unlike many other blockchains, languages on Cardano validate transactions by returning a boolean rather than performing actions like transferring assets, making functional programming a natural fit. Developers write ordinary Haskell code, which is compiled to Untyped Plutus Core (UPLC) — a small functional language based on System F_ω [13].[2] Sometimes referred to as Plutus, UPLC is executed by validating nodes.

The compilation from Plinth to UPLC proceeds through two intermediate languages: (1) *Plutus IR* (PIR) [24], a typed intermediate representation that builds on TPLC (introduced below) by adding support for recursive bindings and algebraic data types, making it a convenient and readable target for high-level optimizations; (2) *Typed Plutus Core* (TPLC) [9], a System-F_ω based calculus equipped with iso-recursive types and the corresponding term-level constructs for general recursion.

[2] Although UPLC is untyped, it results from erasing types from Typed Plutus Core (TPLC), which is based on System F_ω.

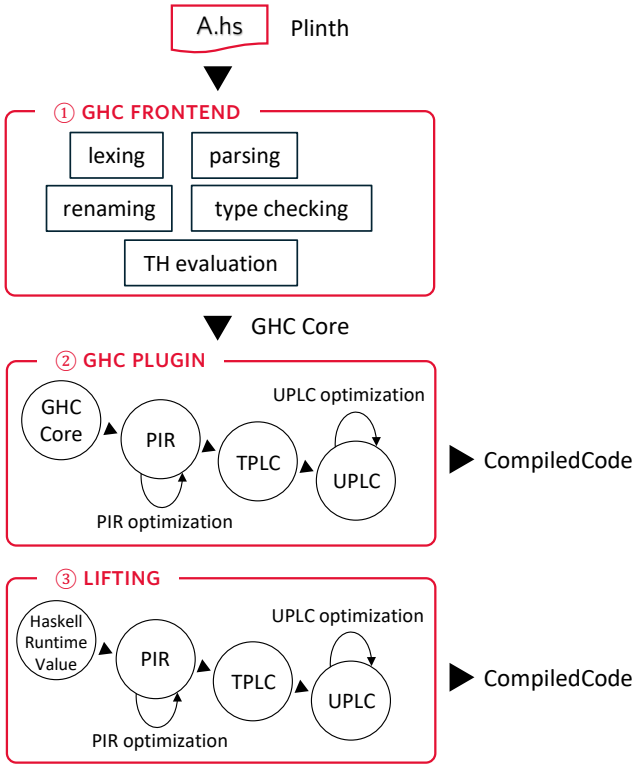

Figure 2. Plinth's Compilation Pipeline

UPLC is produced by erasing types from TPLC, which substantially reduces code size — an important consideration for on-chain storage and processing. TPLC and UPLC are formally specified in [27]. PIR, TPLC, and UPLC include a common and extensible collection of built-in types and functions for efficiency. UPLC is executed using a CEK machine [12]. UPLC programs embedded in Cardano transactions are referred to as Plutus scripts or Plutus validators. Other than native scripts, which validate only simple conditions, all script-validated Cardano transactions use Plutus.

Figure 1a shows a Plinth program which looks just like idiomatic Haskell. Figures 1b and 1c show the corresponding PIR and UPLC. UPLC contains constructs `delay` and `force`, which serve as counterparts to TPLC's type abstractions and instantiations.[3] Originally, UPLC adopted ML-style value restriction [33], requiring the body of type abstractions to be syntactic values. We later chose to drop the restriction to simplify compilation and avoid rejecting programs that users would reasonably expect to be valid. As a result, the body of a type abstraction can be any term, so `delay` is used to prevent premature evaluation after erasing types. `Delay` and `force` are used instead of regular lambdas for better performance, as they appear frequently in Plutus scripts.

[3]We plan to support `case` on booleans in UPLC, which will remove the need for `delay` and `force` in this example, as `case` branches are non-strict, unlike the built-in function `ifThenElse`.

2.1 Differences From Haskell

Plinth differs from Haskell primarily in evaluation strategy: UPLC adopts strict evaluation for simplicity and efficiency of runtime implementation, and predictable evaluation costs. Like Haskell, Plinth supports both strict and non-strict bindings; however, the latter are evaluated by name rather than by need. For example, in **let** $\sim x = e_1$ **in** e_2, e_1 is re-evaluated on each use of x in e_2. Non-strict bindings can be useful for enabling natural expression of certain code patterns, such as contracts with multiple validation conditions, where each condition is relevant in only one execution path.

Short-circuiting behavior is limited to built-in constructs like `if`/`case` branches, and boolean operators `&&` and `||`. As in most other strict languages, users cannot define new non-strict operators.

Plinth supports many standard Haskell features (e.g., algebraic data types, higher-order functions, type classes, and parametric polymorphism), but not more advanced features like existential types, type families, GADTs, IO and FFI.

2.2 Compilation Stages

Figure 2 shows the three stage compilation pipeline.

Stage 1 uses GHC's standard frontend, producing GHC Core [19]. At this stage, users may use Template Haskell for custom compile-time code transformation beyond what the Plinth compiler can do during compilation. One example is encoding data types using the built-in *Data* type, which we discuss in more detail in Section 3.2.

Stage 2 runs in a GHC Core plugin. Users pass the Plinth code to be compiled in a typed quotation to *compile*, which is itself enclosed in a typed splice. This ensures *compile* runs at Haskell compile time. Under the hood, *compile* installs a GHC plugin that does the bulk of the work. For example, the following compiles *codeToCompile* using the plugin:

```
codeToCompile :: τ
codeToCompile = ...
compiled :: CompiledCode τ
compiled = $$(compile ⟦ codeToCompile ⟧)
```

Template Haskell is used here solely to invoke the GHC plugin. Alternatively, one can pass *codeToCompile* to the *plc* function, and use GHC flag `-fplugin` to trigger the plugin:

```
compiled :: CompiledCode τ
compiled = plc (Proxy @"location") codeToCompile
```

Compile and *plc* can therefore be viewed as explicit staging boundaries in multi-stage programming, similar to quotations and splices in Template Haskell, and comparable annotations in other languages [20, 31]. Using *compile* is usually a better choice than *plc*, since the first parameter to *plc* is the code location used in error messages, which is automatically produced when using *compile*.

To compile *codeToCompile*, the plugin needs to access its definition, as well as the definitions it transitively depends on. The standard approach is to mark the relevant definitions as `INLINABLE`. *CompiledCode* τ wraps the compiled PIR and UPLC. It is indexed by a Haskell type, providing type safety in the Haskell environment by ensuring the code being compiled has the correct type. Haskell constructs in *codeToCompile* unsupported by Plinth cause a compile error.

Stage 3 happens during the Haskell program's execution, where Haskell runtime values can be lifted into PIR and compiled to UPLC. Note that this occurs at the runtime of the *Haskell program* — not during the on-chain execution of UPLC. From the on-chain execution perspective, this is still compile time.

Lifting is primarily used when a function takes a parameter, and returns a *CompiledCode* that depends on it. Consider the following:

```
f :: Integer → CompiledCode (Integer → Integer)
f x = $$(compile ⟦ λy → x + y ⟧)
```

This code fails to compile, causing the plugin to emit an error, because x needs to be compiled into a constant in UPLC, but its value isn't available at compile time, when the plugin runs. This reflects a stage constraint imposed by the Plinth compiler: any variable inside the code passed to *compile* or *plc* must either be top-level, or bound within that code.

Instead, the above compilation can be achieved by lifting the value of x into *CompiledCode* at runtime, like so:

```
f :: Integer → CompiledCode (Integer → Integer)
f x = $$(compile ⟦ λx' λy → x'+y ⟧) `apply` lift x
```

Here, *lift* ::*Lift a* ⇒*a* →*CompiledCode a* serializes a runtime value into the corresponding UPLC term, and *apply* constructs a UPLC application node from two subterms. *Lift* is conceptually similar to lifting in Template Haskell, and generally works for types that don't contain functions.

Once we have the desired *CompiledCode*, we can then serialize it, and include it in a transaction for on-chain execution.

3 Compiling Data Types

To support algebraic data types — a core Haskell feature — in Plinth, we explored over time several options for encoding data types in UPLC.

3.1 Scott Encoding and Sums of Products

We initially used Scott encoding [1], a technique for representing data in lambda calculus that supports fast pattern matching, especially when compared to alternatives like Church encoding. This allows keeping the language small and simple. Scott encoding yielded reasonably fast programs, as construction and destruction of values involve relatively small overhead. However, destruction is done via a higher-order function, and strict evaluation in UPLC means all branches need to be evaluated. This overhead was initially expected to be negligible compared to the cost of cryptographic operations. However, later performance measurements showed that this assumption didn't hold.

To address this, we extended the UPLC language with native sums of products [17], introducing two new AST nodes, `Constr` and `Case`, for efficient construction and destruction of values. This simplified data representations, and led to a 30% performance improvement on average.

3.2 *Data* Encoding via Pattern Synonyms and Template Haskell

Despite the above strategies — particularly sums of products — being efficient in principle, we observed that many users bypassed them entirely, instead working directly with the built-in *Data* type. This is a loosely typed representation based on CBOR [5] for on-chain interchange, primarily used to represent arguments passed to Plutus validators. It is unrelated to Haskell's `data` keyword, despite the name.

Matching on *Data* is verbose and costly, involving a chain of built-in calls to extract tags and arguments. It is also lower level and less type safe than using algebraic data types. However, converting between *Data* and other representations is even more expensive, particularly when handling inputs from the blockchain. As a result, users often favor using *Data* throughout to avoid the conversion, prompting us to add support for encoding data types using *Data*.

We considered modifying the compiler to allow compiling PIR data types into *Data*, but rejected this option: *Data* is less expressive (e.g., it cannot contain functions) which would complicate the compiler; handling polymorphic types would be cumbersome; and supporting choosing encodings on a per-type basis would add more complexity. Instead, we added a source-level mechanism leveraging Template Haskell and pattern synonyms, without touching the compiler.

Specifically, users can define a data type inside a Template Haskell quote, which generates a data type that wraps *Data*, along with suitable pattern synonyms — one per data constructor — for construction and destruction. For instance,

```
asData [ d | data Ex a = Ex1 Integer | Ex2 a a |]
```

becomes **newtype** *Ex a =Ex Data*, along with two bidirectional pattern synonyms for constructing and destructing Ex_1 and Ex_2. Ex_2 is shown below and Ex_1 is omitted.

```
pattern Ex2 :: (ToData a, FromData a) ⇒
  a → a → Ex a
pattern Ex2 a b ←
  Ex ( dataAsConstr →
       ( (==) 1 → True
       , [fromData → a, fromData → b] ))
  where
    Ex2 a b = Ex (mkConstr 1 [toData a, toData b])
```

Type classes like *ToData* and *FromData* allow *Ex* to be polymorphic, and it can contain any datatype convertible to and from *Data*.

4 Optimizing PIR and UPLC

Optimization for Plinth diverges from Haskell's optimization due to several factors: (1) Plinth is strict with no laziness; (2) scripts must meet tight size and execution budget limits, and minimizing size is often as important as optimizing speed; and (3) Plinth targets an abstract machine, making certain optimizations useful for targeting machine code — like unboxing — superfluous.

For example, GHC's *full laziness* transformation [18] floats bindings out of lambdas to reduce redundant computation. In Plinth, doing so is far less useful. Since non-strict bindings are not evaluated lazily, floating them out of lambdas doesn't avoid recomputation.

As another example, floating a binding inwards is good for Haskell [18] (as long as it is not floated into a lambda), but it could make a Plinth program more expensive. For instance:

$$\textbf{let}\ \sim a = rhs\ \textbf{in}\ \textbf{let}\ \sim b = \ldots\ a\ \ldots\ \textbf{in}\ b + b$$

Because b is used twice, floating the binding of a into the definition of b causes the overhead of let-bindings to be incurred twice.[4]

In addition, due to Plinth's strictness, transformations like beta and eta reduction may be unsound. For example:

$$\textbf{let}\ and = \lambda x.\ \lambda y.\ \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ False\ \textbf{in}\ and\ l\ r$$

This cannot be rewritten to **if** l **then** r **else** *False*, because the latter doesn't evaluate r if l is false, unlike $and\ l\ r$. However, it remains valid to transform $(\lambda x.e_1)e_0$ into **let** $!x{=}e_0$ **in** e_1.

Nonetheless, some GHC optimizations remain valid and beneficial for Plinth. For example, a non-strict binding that is sure to be evaluated can be made strict. This is particularly useful for Plinth since the binding would otherwise be evaluated on every use. Case-of-case and case-of-known-constructor optimizations are also safe and effective for Plinth. Our inliner is also heavily inspired by GHC's inliner, though it is often more conservative since inlining can increase code size — a critical concern given the strict size limits of Plutus scripts. Nonetheless, there are ways to force inlining, such as using an *inline* function similar to that provided by `GHC.Magic`, or using Template Haskell to produce inlined code before passing it to the Plinth compiler.

We optimize both PIR and UPLC, depending on which is more appropriate. For instance, PIR has algebraic data types and let-bindings, making it the right place for case-of-known-constructor and floating bindings in/out. Moreover, since PIR is typed, many optimization errors can be detected early by the PIR typechecker, making the process safer.

Common sub-expression elimination (CSE), on the other hand, is better suited to UPLC, for two reasons: (1) unlike inlining which can unlock further optimizations, CSE often destroys optimization opportunities, so running it too early is harmful; (2) with types erased, and datatypes and recursion compiled away, more common subexpressions arise in UPLC.

GHC similarly runs CSE towards the end of the pipeline. One difference is that we run CSE interleaved with other simplifier passes, because we observe that CSE can lead to additional inlining opportunities, which can in turn unlock additional CSE opportunities. Consider this example:

$$\begin{aligned}&\lambda x.\ f\ ((\lambda y.\ 1{+}(y{+}y))\ \ (0{+}x))\\ &\qquad\ ((\lambda z.\ 2{+}(z{+}z))\ \ (0{+}x))\end{aligned}$$

After a round of CSE, the two subexpressions $0{+}x$ are replaced with a variable w. Now w can be inlined (replacing y and z), revealing a new common subexpression, $w + w$, which can be further eliminated by another round of CSE.

Considering this, we first run the main simplifier iterations (which include inlining but not CSE), followed by several passes (four by default) of CSE interleaved with the simplifier.

We apply certain optimizations to both PIR and UPLC; a key example is inlining, as it frequently unlocks further optimization opportunities. We don't currently apply any optimizations to TPLC, as we've found the combination of PIR and UPLC optimizations to be sufficient. All TPLC optimizations can be done equally effectively at the PIR stage, while UPLC optimizations are useful because the erasure of types opens up additional optimization opportunities.

5 Runtime and Costing

Plinth programs are compiled to Untyped Plutus Core (UPLC), and executed by a highly optimized CEK machine written in Haskell. The CEK machine is integrated into the Cardano node and is responsible for executing all UPLC scripts, regardless of the source language. It is engineered for high performance and accurate tracking of execution costs.

Besides performance, another key requirement is backwards compatibility. Old scripts must execute identically in perpetuity to preserve blockchain history. This necessitates retaining legacy behavior even when it is suboptimal. We address this through versioned built-ins, or *semantic variants*. The appropriate variant is selected based on the language version and the blockchain's protocol version in use.

The CEK machine tracks CPU and memory usage via a cost model. Each evaluation step (e.g., looking up a variable or processing a lambda abstraction) incurs a fixed cost, while built-in function calls are charged based on argument sizes via costing functions — numerical models fitted using R to benchmarked data. Costs are measured in CPU and memory units: one CPU unit equals one picosecond of CPU time on

[4] The RHS of a, rhs, is evaluated twice regardless of the floating, but additional overhead associated with let-bindings will occur twice, rather than once, if a is floated inwards.

a dedicated benchmarking machine, and one memory unit equals 8 bytes. In contrast to some blockchains, Cardano ensures that script inputs can be determined before on-chain execution, enabling accurate cost estimation ahead of time.

The design of UPLC and its runtime allows for extensibility: the language is parameterized over a set of built-in types and functions. It also allows for different evaluator implementations, with each free to define its own notion of value (e.g., whether values include environments). While originally intended to support multiple blockchains, the extensibility remains useful for testing and benchmarking.

This flexibility, however, introduces substantial ad-hoc polymorphism, which could hurt performance without careful handling. We address this by specializing and inlining, which can sometimes be quite delicate. The `INLINABLE` pragma is conservative and often fails to inline even with `GHC.Magic.inline` used at the callsites. In practice, we mostly use the `INLINE` pragma, which works well for small definitions, but often still needs to be paired with `GHC.Magic.inline` for larger ones to ensure inlining occurs.

Another layer of complexity arises from the need to cache certain computations. Combined with the above concern, this requires carefully interleaving `NOINLINE` and `INLINE` pragmas. It can be time consuming to get right, but is worth the effort — one example saw a 6.7% speedup on average by adding a few `INLINE` pragmas and calls to `GHC.Magic.inline`.

Ensuring correct inlining behavior has, in practice, required manual inspection of GHC Core. This can't be reasonably automated,[5] and can be a painstaking process: GHC Core output can span tens of thousands of lines in a single file, with deeply nested expressions and widespread use of casts that make it hard to follow.

In pursuit of performance, the evaluator often departed from idiomatic Haskell — using custom strict monads, manually written instances, and techniques like unboxed vectors for tight control over allocation and evaluation.

GHC's compilation behavior has also surprised us at times. For example, `-fpedantic-bottoms` can significantly alter operational semantics; strict `let`-bindings may behave differently from strict case expressions. In one case, these led to a 20% slowdown in the evaluator.

Nonetheless, our Haskell evaluator implementation is quite high-level and readable; we are pleased with its performance, and are continually impressed by how well GHC compiles such a tall tower of abstractions down to efficient code operating on unboxed values. As an example, computing Fibonacci numbers on the CEK machine is roughly 12 times slower than in native Haskell — a reasonable performance given the overhead from virtualization and cost accounting. Cardano enforces strict execution budgets: currently 10ms per transaction and 20ms per block. That our evaluator meets these constraints in many real-world scenarios highlights its efficiency and suitability for production, and shows that, with careful design, profiling, and iteration, Haskell remains a compelling choice for building a performant, production-grade blockchain VM.

[5]While tools like *inspection-testing* can check for simple properties, our use case involves more nuanced requirements — certain things need to be optimized in very particular ways — that such tools are not designed to handle.

6 Learnings and Observations

Our experience with Plinth suggests that compiling from a subset of an existing host language — rather than building a new language or an eDSL — can be an effective strategy for designing smart contract languages, and the underlying principles and lessons may apply well beyond this domain. As discussed in Section 1, this approach offers several advantages. Similar strategies have been adopted by other Cardano languages including Scalus [28] and OpShin [23], which reuse subsets of Scala and Python syntax, respectively.

Despite its strengths, this approach also presents some challenges. A main limitation of Plinth is that its evaluation strategy differs from Haskell's: it is a strict language that permits by-name bindings, but not by-need (i.e., lazy) evaluation. This leads to subtle but important differences in programming style compared to idiomatic Haskell.

For example, composing list operations is inefficient, as it eagerly materializes all intermediate results — unlike in Haskell, where laziness avoids this overhead. Guarded recursion, a common Haskell idiom, is generally not useful in Plinth. Tail recursion may perform worse than regular recursion due to how the CEK machine executes UPLC.

As a result, while Plinth's surface syntax — a simple subset of Haskell — makes it easy for developers to get started, writing optimized code requires an adjustment in mindset, especially for experienced Haskell developers. This, however, can be addressed with future enhancements to the language. Lazy evaluation could in theory be implemented for UPLC, though doing so would be challenging in terms of efficiency, security, and predictability of evaluation costs. Or we could introduce limited built-in lazy constructs (like lazy lists) to bring back some of the Haskell idioms in Plinth.

Due to the semantics of Plinth, we generally recommend enabling the Haskell language extension `Strict`. This makes all bindings strict by default, ensuring they are evaluated at most once, whereas non-strict bindings may be evaluated multiple times, leading to high execution cost.

Furthermore, our reliance on GHC comes with its own challenges. It can be tricky to suppress certain unwanted GHC transformations (e.g., inlining, specialization) while preserving others. Some GHC transformations can complicate the process of identifying and compiling specific Haskell functions to domain-specific constructs. To work around this, Plinth comes with its own version of `base`, where some functions are annotated with the `OPAQUE` pragma to prevent

inlining, and some data types are declared in specific ways (e.g., using `data` instead of `newtype`, and using non-strict fields) to prevent certain GHC optimizations.

Generating user-friendly error messages can also be tricky. GHC Core is desugared and often diverges from the surface code, making it difficult to provide meaningful diagnostics. Supporting multiple major GHC versions has also proven difficult, due to the instability of GHC API across major releases. These issues are less fundamental — for instance, leveraging a source plugin could enhance error reporting, which we plan to explore.

Initially, we envisioned a unified Haskell-based workflow, where both on-chain and off-chain components would be written in Haskell with shared code — an appealing idea in principle. However, developers over time have gravitated toward off-chain frameworks in more mainstream languages like JavaScript and Python. We believe several factors contributed to this shift. Off-chain code often integrates with user-facing systems like wallets and web applications, areas where Haskell sees limited adoption. Moreover, unlike developing a language targeting UPLC, there is little inherent advantage to implementing off-chain frameworks in Haskell. This led to more community effort being directed toward JavaScript and Python frameworks — especially due to the network effects of these popular languages — which quickly became more mature and better supported. Finally, the overlap in practice between on-chain and off-chain code is usually limited to data type definitions, and this is made manageable by Plutus Contract Blueprints [4], which enable serialization and cross-language sharing of schema definitions.

7 Related Work

While GHC plugins are a powerful mechanism for program transformation, complex production uses remain rare. This makes Plinth's architecture a relatively novel case. One related effort compiles GHC Core to Cartesian Closed Categories [11], which can be interpreted into different concrete targets. Categorifier [6] applies this to compile a subset of Haskell to C for flight control systems. The same categorical approach has also been applied to automatic differentiation for deep learning [30]. In contrast, Plinth targets a specific backend (PIR) rather than a general categorical abstraction. This gives us greater control over compilation: we support a broader range of Haskell constructs (e.g., non-function values, type class methods) and achieve faster compilation.

Rather than integrating Plinth compilation directly into GHC, it is also possible to build a standalone compiler using GHC as a library. A notable example is the compiler for Clash [3], which leverages GHC as a library to compile a subset of Haskell into hardware description languages like VHDL and Verilog. We chose the plugin approach for several reasons. It makes developing Plinth programs feel similar to regular Haskell development — using GHC, and tools like Cabal, GHCi, and Template Haskell as usual — which is especially attractive to teams already invested in the Haskell ecosystem. Besides, this approach produces *CompiledCode* values in Haskell, enabling users to inspect, manipulate and test them with regular Haskell. This further keeps the workflow close to idiomatic Haskell, lowering the learning curve and tooling overhead. A standalone compiler, by contrast, offers more control, simplifies the surface syntax (e.g., by automatically enabling flags and deriving instances), and could improve error reporting — a direction we'd like to explore.

Formal methods play a significant role in the development of Plinth and its runtime. While Plinth does not have a fully verified semantics, TPLC and UPLC are formalized in Agda [9], with progress and preservation proofs and executable semantics used in conformance tests. A recent overview is given in [8]. A certified compilation framework for PIR and UPLC is also under development [21]. Structured Contracts [32] introduces a framework for reasoning about the specifications of Cardano smart contracts.

Besides Plinth, there are several other Cardano languages targeting UPLC. These include Plutarch [26] — a traditional Haskell eDSL. The advantages of Plinth's approach compared to eDSLs have been discussed in Section 1. Plutarch, however, has the benefit of being easier to develop and maintain, as it does not require deep integration with GHC — an eDSL is a regular Haskell library for manipulating expressions.

Apart from Plutarch, there are also standalone languages [2, 14], and Scala, Python and TypeScript-based languages [23, 25, 28]. Plinth also serves as the compilation target for Marlowe [29], a high-level Haskell eDSL for financial contracts.

Quoted DSLs (QDSLs) [10, 22] offer another approach to designing domain-specific languages. Like Plinth, QDSLs reuse the host language's AST, allowing use of simple values and native branching. Compared to QDSLs, Plinth has much less syntactic overhead since no explicit quoting or splicing is required, and benefits from working, via a plugin, directly with GHC Core, which is much smaller and simpler to manipulate than Template Haskell ASTs.

8 Conclusions and Future Work

Plinth demonstrates that compiling from a subset of Haskell is a viable and practical approach for developing domain-specific languages. In this report we presented the language, the compilation and runtime processes, and reflected on the design choices, highlighting both benefits and limitations.

Looking ahead, we aim to support modules in UPLC [15] to enable modular on-chain code and better reuse. We also plan to investigate using a GHC source plugin to reduce unwanted transformations, and improve error reporting by preserving source-level context. Furthermore, we are exploring ways to introduce laziness into the language, along with developer experience improvements, including support for multiple major GHC versions.

Acknowledgments

Plinth owes its existence to the efforts of many people, especially our current and former team members. We are particularly grateful to Ana Pantilie, Andrew Sutherland, James Chapman, Kasey White, Lorenzo Calegari, Lucas Rosa, Marshall Swatt, Mauro Jaskelioff, Nikolaos Bezirgiannis, Philip DiSarro, Ramsay Taylor, Seungheon Oh, and Yuriy Lazaryev, for advancing the design, implementation, and formalization of Plinth and Plutus, and to the many others whose work and ideas have shaped the experiences described in this report.

References

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

[2] Aiken. 2025. Aiken | The modern smart contract platform for Cardano. https://aiken-lang.org/.

[3] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 714–721. doi:10.1109/DSD.2010.21

[4] Matthias Benkort. 2025. Plutus Contract Blueprint. https://cips.cardano.org/cip/CIP-57.

[5] Carsten Bormann and Paul E. Hoffman. 2020. Concise Binary Object Representation (CBOR). RFC 8949. doi:10.17487/RFC8949

[6] Categorifier. 2025. Categorifier: Interpret Haskell programs into any cartesian closed category. https://github.com/con-kitty/categorifier.

[7] 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: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*. Springer, 525–539. doi:10.1007/978-3-030-54455-3_37

[8] James Chapman, Arnaud Bailly, and Polina Vinogradova. 2024. Applying Continuous Formal Methods to Cardano (Experience Report). In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture, FUNARCH 2024, Milan, Italy, 6 September 2024*. ACM, 18–24. doi:10.1145/3677998.3678222

[9] 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

[10] James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-Integrated Query. *SIGPLAN Not.* 48, 9 (Sept. 2013), 403–416. doi:10.1145/2544174.2500586

[11] Conal Elliott. 2017. Compiling to Categories. *Proc. ACM Program. Lang.* 1, ICFP (2017), 27:1–27:27. doi:10.1145/3110271

[12] Matthias Felleisen and Daniel P. Friedman. 1987. Control Operators, the SECD-Machine, and the λ-Calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*. North-Holland, 193–222.

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

[14] Helios. 2025. Helios. https://github.com/HeliosLang/compiler.

[15] John Hughes. 2025. Modules in UPLC. https://github.com/cardano-foundation/CIPs/pull/946.

[16] Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. 2024. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. arXiv:2410.03981 [cs.SE] https://arxiv.org/abs/2410.03981

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

[18] Simon L. Peyton Jones, Will Partain, and André L. M. Santos. 1996. Let-floating: Moving Bindings to Give Faster Programs. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*. ACM, 1–12. doi:10.1145/232627.232630

[19] Simon L. Peyton Jones and André L. M. Santos. 1998. A Transformation-Based Optimiser for Haskell. *Sci. Comput. Program.* 32, 1-3 (1998), 3–47. doi:10.1016/S0167-6423(97)00029-4

[20] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*. Springer International Publishing, Cham, 86–102.

[21] Jacco O.G. Krijnen, Manuel M.T. Chakravarty, Gabriele Keller, and Wouter Swierstra. 2024. Translation Certification for Smart Contracts. *Science of Computer Programming* 233 (2024), 103051. doi:10.1016/j.scico.2023.103051

[22] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old Is New Again: Quoted Domain-Specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. ACM, 25–36. doi:10.1145/2847538.2847541

[23] OpShin. 2025. OpShin: A simple pythonic programming language for Smart Contracts on Cardano. https://github.com/OpShin/opshin.

[24] Michael Peyton Jones, Vasilis Gkoumas, Roman Kireev, Kenneth MacKenzie, Chad Nester, and Philip Wadler. 2019. Unraveling Recursion: Compiling an IR with Recursion to System F. In *Mathematics of Program Construction (MPC) 2019*. Springer, 414–443. doi:10.1007/978-3-030-33636-3_15

[25] plu-ts. 2025. plu-ts. https://github.com/HarmonicLabs/plu-ts.

[26] Plutarch. 2025. Plutarch. https://github.com/Plutonomicon/plutarch-plutus.

[27] Plutus Core Team. 2025. Formal Specification of the Plutus Core Language. https://github.com/IntersectMBO/plutus/tree/master/doc/plutus-core-spec.

[28] Scalus. 2025. Scalus - DApps Development Platform for Cardano. https://scalus.org/.

[29] 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 - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12063)*. Springer, 496–511. doi:10.1007/978-3-030-54455-3_35

[30] Mike Sperber. 2023. Fast Deep Learning with Categories. https://icfp23.sigplan.org/details/FHPNC-2023/6/Fast-Deep-Learning-with-Categories.

[31] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. ACM, 203–217. doi:10.1145/258993.259019

[32] Polina Vinogradova, Orestis Melkonian, Philip Wadler, Manuel M. T. Chakravarty, Jacco Krijnen, Michael Peyton Jones, James Chapman, and Tudor Ferariu. 2024. Structured Contracts in the EUTxO Ledger Model. In *5th International Workshop on Formal Methods for Blockchains, FMBC 2024, April 7, 2024, Luxembourg City, Luxembourg (OASIcs, Vol. 118)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:19. doi:10.4230/OASICS.FMBC.2024.10

[33] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp Symb. Comput.* 8, 4 (Dec. 1995), 343–355. doi:10.1007/BF01018828

Received 2025-06-09; accepted 2025-07-17