The Plutus Platform

An IOHK technical report

Michael Peyton Jones
michael.peyton-jones@iohk.io

Roman Kireev
roman.kireev@iohk.io

Contents

1 The Plutus Platform

The Plutus Platform is a platform for writing *applications* that interact with a *distributed ledger* featuring *scripting capabilities*, in particular the Cardano blockchain. A high-level architecture of the Plutus Platform on Cardano is shown in Fig. 1, with an emphasis on applications.

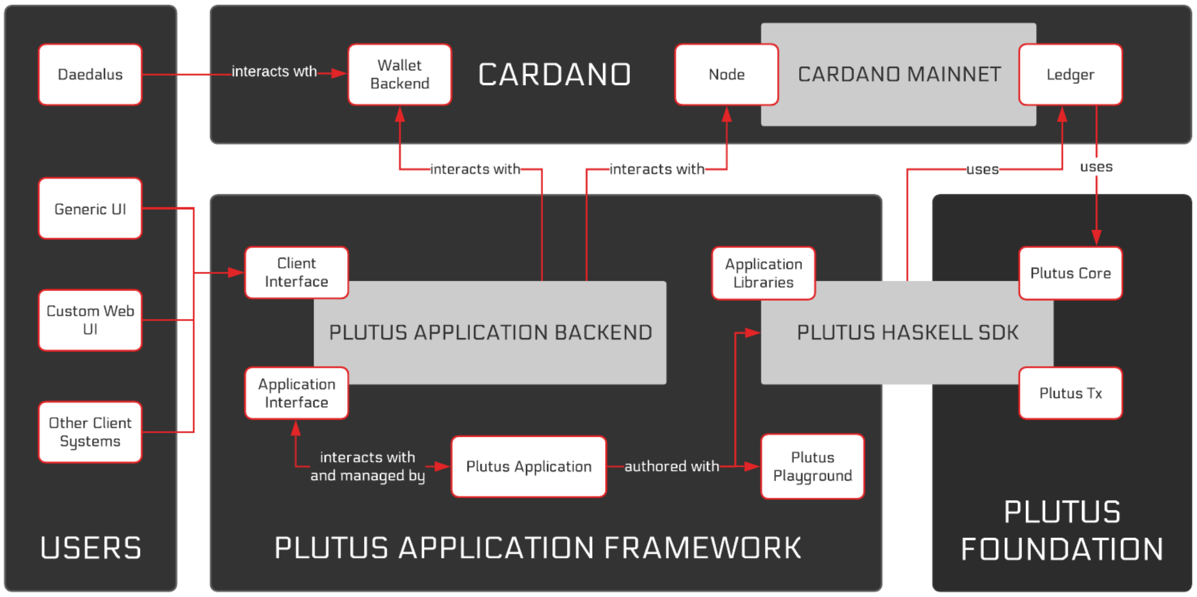

Figure 1: Architecture of the Plutus Platform

Ledgers. The Plutus Platform is designed to work with distributed ledgers (henceforth simply "ledgers").[1] While the design of the Platform supports writing very simple applications that do nothing but occasionally submit asset-transfer transactions to the ledger, the main focus is on the applications which are enabled by a ledger with scripting functionality. Specifically, we look at Unspent Transaction Output (UTXO) ledgers which implement what we call the Extended UTXO Model.

The ledger model is described in Section 2.

Scripting. Ledgers without scripting functionality can only support very simple applications. Much of the interest in applications that interact with blockchains has come from the ability to put some part of the application code *on* the chain itself (such code is generally referred to as a "script"), such that it is guaranteed to be executed correctly as part of the consensus protocol of the blockchain. This enables applications to have small kernels of "trusted" code which ensures that critical safety conditions are met.

The Plutus Platform requires a particular scripting model from the ledger, which is described with the general ledger model in Section 2. This scripting model is agnostic about the scripting *language* which is used for scripts, but the Plutus Haskell SDK is designed to work with a particular scripting language, namely Plutus Core. Plutus Core is described in Section 3.

Applications. In the Plutus Platform a Plutus application is simply a program that interacts with a distributed ledger. We use the rather generic word "application" over the more usual "smart contract" or "distributed application" to emphasize the generic nature of the programs we are considering — indeed, they need not scripting functionality or have distributed participants!

The point of the Platform is to enable such applications by providing support in the ledger, as well as support for authoring, distributing, and running them. While the modifications to the ledger may seem like the most technically substantive part of the work, most of the application behaviour is in the part that runs off the chain, and this requires a commensurate amount of support.

The Plutus application model is described in Section 5, and our support for authoring Plutus applications in Section 6.

[1]More specifically, it is designed to work with the Cardano blockchain, although it could function on other systems that implement the requisite ledger functionality.

2 The ledger: the Extended UTXO Model

Michael: Maybe we should just move the content of the EUTXO doc into here?

The Plutus Platform is designed to work with a specific kind of ledger, namely an extended form of the traditional UTXO ledger introduced by Bitcoin. We refer to our ledger model as the Extended UTXO (EUTXO) Model, and it is discussed in detail in Chakravarty et al. [1, 3, 5].

We give a high-level overview here, further details can be found in the above references.

2.1 Requirements

Requirement 2.1 (Locality)
UTXO ledgers have the desirable property that all the information which is needed to validate a transaction is specified in the transaction. In particular, there is no global state apart from the UTXO set.

This is in contrast to account-based ledgers, where account balances are global.[2]

We aim not to disturb this property, since it is helpful in a number of ways. For example, it allows a greater degree of parallel processing. In Chakravarty et al. [4] this property is used to enable fast optimistic settlement of transactions off-chain, *including* support for the scripting model we use. This would not be possible if we followed Ethereum's actor model and had contract instances with global state.

Requirement 2.2 (Determinism)
We would like our ledger rules to be *deterministic*, that is, the outcome of transaction validation is entirely determined by the transaction in question (and the current state of the ledger of course). This means that the whole process can be simulated accurately by the user *before* submitting a transaction: thus both the outcome of validation and the amount of resources consumed can be determined ahead of time. This is helpful for systems that charge for script execution, since users can reliably compute how much they will need to pay ahead of time.

A common way for systems to violate this property is by providing scripts with access to some piece of mutable information, such as the current time (in our system, the current slot has this role). Scripts can then branch on this information, leading to non-deterministic behaviour.

Requirement 2.3 (Inspectability)
As far as possible, we would like the data which we put on the ledger to be *inspectable*. Hence the off-chain code needs to be able to inspect the datum and interpret it as a value it can work with.

This is primarily important for off-chain code. For example, when implementing Plutus applications which are state machines, we will use the datums of outputs to store the on-chain *state* of the machine. If the off-chain code wants to submit a transaction to advance such a state machine, it may need to read the current state from the chain (after all, the last transition might not have been performed by this agent).

Requirement 2.4 (UTXO size)
Slot leaders are likely to be faced with resource constraints when doing transaction validation. A key source of memory usage is that the whole set of current UTXO entries must be kept in memory at all times to check for double-spending. Hence, it is important to keep the size of UTXO entries small.

2.2 The Data type

We will later require the ability to pass arguments to scripts. However, this requires us to turn such arguments into values in our scripting language of the appropriate type.

We have opted to use a single, structured datatype for interchange with scripts. We call this type Data, and it is described further in Chakravarty et al. [1].[3] This means that the redeemer and datum are of type Data, and the context is encoded as Data before being passed to the validator script.

We also require a way to turn the Data values into values in our scripting language, but we can do this via the lifting machinery in Section 6.3.5.

[2] In fact Cardano has a small maount

[3] In practice it is very similar to other forms of structured data, such as JSON, but optimized to work with CBOR [12].

On the user side, users are responsible for turning their specific types into Data, but this can be done in a standardized way, and we provide Template Haskell functions for generating the appropriate typeclass instances.

2.3 UTXO ledgers

A UTXO ledger represents a ledger as a series of *transactions*, each of which has a set of transaction *inputs* and transaction *outputs*.

A transaction output carries some amount of *value* and is *locked* in some way that restricts how it can be spent. The simplest form of locking is simply to attach a public key.

A transaction input is a reference to a transaction output. For example, a pair of a transaction ID and an index into the (ordered) set of its outputs.

2.3.1 UTXO validity

For a transaction to be *valid* against an existing UTXO ledger, it must be the case that:

- All inputs refer to outputs which exist in the ledger and have not been spent by another transaction in the ledger.
- The transaction balances: the sum of the value on the spent outputs equals the sum of the value on the transaction outputs.
- The transaction is authorized: for each spent output locked by a public key, the transaction contains a witness (a signature over the transaction body) from the corresponding private key.

Hence ownership in a UTXO ledger consists in having the right to spend an output: one owns outputs rather than an aggregate amount of funds.

2.4 EUTXO-1: scripting

Michael: This section is a mess and doesn't tell the story well.

The first addition we make is to support scripting. Our approach is based of existing models for UTXO ledgers with scripts (e.g. Zahnentferner [21]), but we make a number of additions.

We make the following changes to the UTXO model:

- Every transaction has a *validity interval* over slots. A slot leader will only process the transaction if the current slot number lies within the transaction's validity interval.
- The ledger learns about a *scripting language* and how to evaluate scripts in that language.[4]
- Transaction outputs can be locked by a validator script (such outputs are called script outputs).
- Script outputs have an attached datum, which is a value of type Data.
- When spending a script output, the spending transaction must provide a redeemer of type Data for each such output.
- Validator scripts are passed three arguments: the datum of the output being spent, the redeemer for the output, and a value called the context which has a representation of the transaction being validated.

Michael: I'm totally glossing over the details about which things are hashes and the whole data witnesses thing.

[4]In the Plutus Platform this language is Plutus Core.

2.4.1 The contents of the context

The context type is a summary of the information contained in a transaction, made suitable for consumption in a script.

The exact form is subject to change (see Chakravarty et al. [1] for the details), but it should contain all the information about the inputs, outputs, metadata, etc.

2.4.2 Ensuring determinism

The information provided by the context is entirely determined by the transaction itself. This ensures that we satisfy Requirement 2.2.

We handle the issue of time by providing only the validation interval of the transaction. This serves to give the validator script a window within which the current time must lie: in practice this is enough to establish that the current time is "definitely before" or "definitely after" some particular time, which is the most common kind of constraint.

Some additional responsibility passes to the author of the transaction as a result of this. If the validator script they are trying to satisfy requires that the transaction occur before some time T, then they must not only submit the transaction before T, they must ensure that the upper bound of the validity interval is below T. This doesn't make the transaction less likely to validate (it would not have validated after T regardless!), but it does require some more work from the off-chain code.

2.4.3 EUTXO-1 validity

In addition to the conditions in Section 2.3.1, the following condition must hold:

- Scripts must succeed: if a transaction spends a script output, then the validator script on the output must be run with the redeemer, datum, and context and return successfully.

2.5 EUTXO-2: multicurrency

Michael: This section is also a mess and doesn't tell the story well.

The other extension that we make to the ledger is to enable *multicurrency*, that is, to enable our ledger to represent a wider variety of assets than a single one. Multicurrency is a key part of the Plutus Platform not only because custom currencies are an important use case for blockchain systems, but because we use lightweight custom tokens as integral parts of our application designs.

The key design innovation is that currencies are identified with the hash of the Monetary Policy Script (MPS) that controls them. This means that to be allowed to forge tokens of a particular currency, you must present the corresponding MPS with the transaction and it must pass.

This design allows our system to be both *local* and *lightweight*:

- We do not require any kind of central registration of currencies: any hash can be used as a currency id, you just cannot forge any unless you can produce a script with that hash. This ensures we do not violate Requirement 2.1.
- Forging tokens is cheap: all that is required is to provide and run the script at the time of forging.
- Transacting with custom currencies is cheap: they are natively supported and work just like the native token.

Michael: Say more, maybe write out some of the requirements separately.

2.5.1 The Value type

The first change we make is to our notion of "value" on the ledger. In traditional UTXO ledgers, value is just an integer representing a quantity of the (unique) asset tracked by the ledger.

We generalize this to support multiple asset classes as follows: we define a type Value which represents a *bundle* of assets, indexed by a *currency id* and then by a *token name*.

```
type Value = Map CurrencyID (Map TokenName Quantity)
```

The currency id indicates the *controller* of the currency, specifically it is the hash of the MPS that controls when tokens of that currency may be forged or destroyed (see Section 2.5.2). The token name separates different *classes* of token within the currency. Only tokens of the same currency and token name are fungible with each other. Hence one could use a currency to track items in a game: different kinds of items would have different token names, and an asset bundle could consist of e.g. 3 swords, 7 hats, and a cheesecake.

Operations on Value are defined as if it were a finitely-supported function.

The ledger rules do not need to change much in order to support this generalized Value: the only change is that we need to use the appropriate sum operation on Value in e.g. the balancing rules.

2.5.2 Forging

Value may also be created and destroyed, we call this *forging*. To this end transactions are given two additional fields:

1. `forge` which contains Value
2. `forgeScripts` which contains MPSs for all the currencies forged in `forge`

2.5.3 Monetary Policy Scripts

The scripts which control the forging of tokens are called Monetary Policy Scripts (MPSs). Since these are able to see the context, and hence the whole transaction being validated, they can enforce a wide range of properties on the transaction.

2.5.4 Non-Fungible Tokens

A Non-Fungible Token (NFT) is a unique token which can be transferred to another user, but not duplicated. NFTs have proven useful in a number of blockchain applications (see *ERC-721 standard for non-fungible tokens in Ethereum* [8]); for example, they can represent ownership of some object in a game, or shares in a company, or many other kinds of asset. We can implement NFTs as token classes within a currency whose supply is limited to a single token.

Ensuring that the tokens are unique is not trivial, since uniqueness is a global property and transactions only have access to local information. Two approaches are:

- Use a one-shot MPS. This ensures that the MPS can only be run once, and then so long as you do not issue any duplicates there, you will have uniqueness.
- Run a unique state machine (beginning with a one-shot transaction to ensure uniqueness) which keeps all the issued tokens in its state. This effectively introduces global state that must be consulted before each forging, allowing uniqueness to be maintained.

2.5.5 EUTXO-2 validity

In addition to the conditions in Section 2.3.1 and Section 2.4.3, the following condition must hold:

- Forging must be authorized: for each currency id of which a non-zero amount is forged in `forge`, the corresponding MPS must be run with the context and return successfully.

3 Scripting: Plutus Core

Our scripting language for the Plutus Platform is Plutus Core (strictly, type-erased Plutus Core, see Section 3.7). A formal specification for Plutus Core is available in *Formal specification of the Plutus Core language* [9].

We give a high-level overview of the language here, further details can be found in the above reference.

3.1 Requirements

Requirement 3.1 (Conservatism)
Designing a new programming language is hard. It is very easy to make choices that come back to haunt you.

Moreover, whatever language we choose for our scripting language will be very hard to change, since it will be involved in transaction validation, and we want to be able to validate old transactions. So we can release new versions, but we must support old versions forever.

This combination means that it is hard for us to get the language right first time, and it is also hard for us to iterate on it. This suggests a case for *conservatism*: pick things that are tried and tested, and don't try to innovate too much.

Conservatism also makes requirements such as Requirement 3.4 easier to satisfy, as we can build on existing work.

Requirement 3.2 (Minimalism)
The smaller our language, the less there is to go wrong, and the less there is to reason about.

Minimalism makes requirements such as Requirement 3.4 easier to satisfy, since there is less to formalize.

However, this is a tradeoff, as a simpler target language often means more complexity in the compilers that target that language. But it is much easier for us to change the compilers than it is to change the scripting language, so this is worth it.

Requirement 3.3 (Safety)
Once submitted as part of a blockchain transaction, scripts are immutable. Therefore we want as much certainty as we can get about what the code will do, otherwise there is risk that the value involved will be lost or stolen.

Requirement 3.4 (Formalization)
One part of reasoning about what our programming language does is to *formalize* its semantics, so we can be sure that 1. it has a sensible semantics, and 2. the implementation agrees with that semantics.

Requirement 3.5 (Size)
The representation of the scripting language on the chain must not be too large, since 1. users will pay for the size of transactions, and 2. transaction size has a major effect on the throughput of the system.

Requirement 3.6 (Extensibility)
We may want to use our scripting language on other ledgers in the future, and they may well support different basic primitives, so we need this to be configurable. For example, privacy-preserving ledgers may restrict the kinds of computation that can be done, which may mean we only have some kinds of arithmetic operations.

Requirement 3.7 (Multiple source languages)
It would be nice if multiple source languages could be compiled to our scripting language. This would potentially encourage usage by developers who are more comfortable with one or the other of the source languages.

However, this is somewhat speculative and actually implementing other source languages would be a lot of work. But if we can cheaply make this easier then that is good.

3.2 Designing Plutus Core

How do we design our scripting language? Requirements 3.1 and 3.2 suggest adhering to existing languages as much as possible, and picking a small, well-studied language. We also want to use a statically-typed functional programming language, since we intend to use functional programming languages (starting with Haskell) as our source languages.

This suggests using some variant of the lambda calculus. We decided to start with *System F*, also known as the polymorphic lambda calculus [10]. Haskell's internal language, GHC Core [15], is also based on System F (no coincidence), but we make a couple of different decisions:

1. We do not have primitive datatypes and case expressions, rather we base our language on *System* F_ω^μ, which extends System F with recursive types and higher-kinded types.

2. Our language is strict, rather than lazy, by default.

3. We do not support most of the extensions that Haskell has pioneered, such as coercions.

As a result, the formal specification of our language can be described in one line: it is exactly System F_ω^μ with appropriate primitive types and operations.

3.3 Datatypes

If we do not have primitive datatypes, how *do* we deal with datatypes? The answer is that it is up to the compiler to encode them — another example of the tradeoff discussed in Requirement 3.2.

In our case Plutus IR does have datatypes, so this is handled by the Plutus IR compiler. See Section 6.4 for more details.

3.4 Recursive types

The one-line description above turns out not to be as unambiguous as one might hope. We have to choose between equirecursive types and isorecursive types [16, chapter 21].

There is a tradeoff here between simplicity of writing code in the language, and simplicity of the language's metatheory. Since Plutus Core is a compilation target rather than a source language, we opted to go for isorecursive types, which have the simpler metatheory (aiding Requirement 3.4). The complexity is handled by the Plutus IR compiler.

This choice is discussed more in *Formal specification of the Plutus Core language* [9] and Peyton Jones et al. [14].

3.5 Recursive values

While Plutus Core has support for recursive types, it does not have any (direct) support for recursive *values*. It turns out that recursive types are sufficient to implement the usual array of fixpoint combinators, and so encode recursive values [11]. Again, this encoding is handled by the Plutus IR compiler, see Section 6.4.

Doing this in full generality turns out to be surprisingly tricky, see Peyton Jones et al. [14].

3.6 Builtin types and values

Programming languages commonly have some form of foreign function interface. The idea is that:

1. We don't want to reimplement everything from the bottom up in a newly designed language. It makes much more sense to delegate certain computations to an existing tool where such computations already exist in optimized form, with good tests etc.

2. In particular, We certainly don't want to implement our own version of arithmetic (integers, addition, multiplication, etc.) or any other low-level stuff where it will be especially hard to beat external implementations.

We certainly need efficient primitives in Plutus Core, including arithmetic.

To implement this, we could build `Integer` into the grammar of Plutus Core types, and `Integer` constants into the grammar of Plutus Core terms, as well as a few functions (addition, multiplication, etc.). If we need more types or functions, then we simply have to extend the hardcoded list of types and functions.

However, this is very inflexible. The key issue is that we may want to have *different* sets of builtins in different situations. For example:

1. During testing it can be convenient to have a builtin which implements *tracing*: emitting log lines to a global trace output. But we don't want this functionality on-chain, so we want to remove the builtin (or replace it with a no-op).

2. We may need different builtins for different ledgers (see Requirement 3.6).

So we need to make builtin types and functions extensible.

3.6.1 Builtin types

Extensible builtin types are relatively straightforward on the type level where they are opaque — we don't need to know anything about a type other than its kind in order to handle type normalization etc. However, values of builtin types ("constants" in Plutus Core) are trickier. It's easy to say what a `Integer` constant should be: an integer! However, what about constants of extensible builtin types?

We adopt an approach familiar from generic programming in dependently-typed languages, where we use a *universe* of types to represent the types our programs can operate over.[5] In our case, we are entirely parametric over the universe itself, with different universes corresponding to different sets of builtin types. A constant then is then *tagged* with a value corresponding to a type in that universe, and a value of that type.

This does complicate interpretation of Plutus Core programs, since in order to interpret a program, you also need to know what universe of types it is expecting. In practice this is not so bad: for most users (e.g. the Cardano ledger) there will be a single universe that they use, and from there on they can simply act as though we did not have extensible builtins.

Builtin datatypes Adding new types to Plutus Core is not very useful unless they have functions that can operate on them! A particularly common case is where the new type is a datatype, and should support pattern matching. We can support this by also providing a "matching" function as a builtin function.[6]

For example, the matching function for `bool` is `ifThenElse :: forall a . bool -> a -> a -> a`. Matching functions are always polymorphic, which is a key reason why we need to support polymorphic builtin functions (see Section 3.6.2).

Polymorphic builtin types It would be nice to support polymorphic builtin types. For example, that would allow us to provide optimized implementations of container types such as maps.

This seems feasible in principle, but we have not worked the details out yet.

3.6.2 Builtin functions

We could implement extensible builtin functions in a similar way to extensible builtin types: use some type (like the universe for types) that catalogues the available builtin functions. However, in practice we don't need to model the available functions quite as precisely, since the syntax doesn't depend on the available functions as it does on the available types.

So instead we adopt a much simpler approach where the available builtin functions can be provided at runtime simply indexed by a name. We may change this in future to use the explicit catalogue of functions.

Passing arguments to builtin functions Builtin functions must of course be provided with a type signature (since we have a typed language), and a way to evaluate them. The natural way to provide such evaluators is as functions — but these functions must be able to operate on Plutus Core terms! For example, an evaluator for "plus" must be able to take two *terms* of type `Integer` and turn them into another term. But a simple implementation of "plus" just talks about *integers*, not terms.

Solving this problem in generality is difficult, so we make a key simplification: builtin functions may only accept arguments of builtin types (with one exception, as we will see shortly). Along with the fact that (since Plutus Core is eager) function arguments are evaluated to values before being passed to the function, this means that the arguments to builtin evaluators will always be *constants*, which are easy to unwrap into their underlying values.

Polymorphic builtin functions The exception to the above rule about builtin function types is for *polymorphic* builtin functions. These must necessarily be able to process arbitrary terms (since they can be instantiated at any type, not just a builtin type). For example, `ifThenElse` must be able to produce a value of type `Integer -> Integer`, even though this is not a builtin type.

[5] Where do these types themselves come from? We assume that we are working in some "metalanguage" which has types such as `Integer` that we can reuse in Plutus Core.

[6] These functions look just like the "matchers" for Scott-encoded datatypes, which is no coincidence.

It is possible in principle to use polymorphic *evaluators* to implement polymorphic builtin functions [13]. Unfortunately this is quite a heavyweight and complex encoding for such a key usecase.

Instead, we require evaluators for polymorphic functions to be *parametric*, which means in particular that they do not inspect their polymorphic arguments. We can thus pass the *term itself* as the polymorphic argument, instead of needing to extract a value of an appropriate type in the metalanguage, as we do for other arguments. Thus, an evaluator for a builtin function which implements `const :: forall a b . a -> b -> a` would take two *terms* and return the first one.

Saturated builtin application When including builtin function application in the AST, we have two options:

1. Builtin functions can stand alone as a value with the builtin's function type, and are applied to their arguments with normal function application.
2. Builtin functions must always appear *saturated*, that is, with all of their arguments.

There choice is fairly minor, but has a few consequences:

1. Unsaturated builtins can be partially applied, saturated builtins cannot be — but we can easily work around this by wrapping the builtin in a lambda, so this does not matter in practice.
2. Saturated builtins are easier to write an efficient evaluator for. Unlike normal functions, we cannot do any real evaluation of a builtin function until we have *all* the arguments. So it is easier if we only ever see them with all their arguments.

We've chosen to use saturated builtin applications.

Errors from builtin functions Builtin functions *are* currently allowed to fail, i.e. to use the `error` effect in Plutus Core. This allows us to handle cases like $1 \div 0$ without having to complicate the builtin machinery with an explicit error handling facility like `Maybe`.

Builtin functions are *not* currently allowed to catch errors (nothing in Plutus Core can catch an error, currently).

3.7 Erasure

Originally we planned to use typed Plutus Core as the actual scripting language. However, we discovered that the explicit types made up a large proportion of the overall size of the code ($\approx 80\%$). Given that we care about size (Requirement 3.5), this was too compelling an improvement to pass by.

Hence we decided to instead use *type-erased* Plutus Core as our scripting language instead, with typed Plutus Core as a (useful) intermediary language.

We were initially concerned that if we did not typecheck the application of the validator script to its arguments (which we cannot do if we've erased the types of the validator script!), then that might allow a malicious attacker to pass a script ill-typed arguments, potentially causing unexpected behaviour. However, given the current design where all the arguments to the validator script are of type `Data`, and are constructed by the validating node (which is a trusted party), it is no longer possible for the arguments to be ill-typed.

Erased Plutus Core is much closer to the untyped lambda calculus, and as such is also an easier compilation target (e.g. for a dependently-typed language, or a non-statically-typed language), hence this also helps with Requirement 3.7.

3.8 Formalization

As discussed in Requirement 3.4, we would like to formalize Plutus Core. We have done so in Agda: the resulting formalization is partially published in Chapman et al. [6], and the living version is available in *The Plutus repository* [20].

Our formalization includes the type system and semantics, proofs of progress and preservation, and an evaluator which provably implements the semantics. This evaluator can be extracted to a Haskell executable, which we use to cross-test against the Haskell interpreter.

4 Resources: Costing Plutus Core

A key, unusual, requirement of the scripting language we use is that it must have a built-in notion of resource tracking. We are going to run untrusted code on many machines during transaction validation and diffusion - it is important that we carefully control its resource usage.

This is a familiar problem in other systems such as Ethereum (which tracks resource usage using "gas"), but we are focussing on a somewhat different set of requirements.

4.1 Requirements

Requirement 4.1 (Profitable fees)
Slot leaders are compensated for the work that they do validating transactions by being given the transaction fees. This ensures that it is economical (or even profitable) to run a stake pool even just based on transaction fee income. We therefore need to ensure that the fees for script execution don't change this dynamic, e.g. by making it un-economical to run a pool.

If transactions with scripts are un-economical to process, then slot leaders may simply choose not to include them, which would compromise the usability of the system.

Requirement 4.2 (DoS security)
Scripts allow the user who submits a transaction to force every node on the network to perform some computation of the user's choice. This provides an obvious angle for DoS attacks: simply flood the network with pointless transactions that require a lot of computation, thus preventing the network from doing anything else. Of course, any such attack must pay the script execution fees for these transactions. So we need to make the fees high enough that such an attack is un-economical.

Requirement 4.3 (Usable fees)
We do want users of Cardano to actually use scripts. So the cost to run scripts can't be too high, otherwise nobody will use them. The simplest way to encourage this is to make the evaluator faster, which benefits legitimate users without compromising the security of the system.

Requirement 4.4 (Determinism)
Script execution cost affects whether a transaction validates, and as such if the costs are non-deterministic then it is non-deterministic whether a transaction with scripts will validate at all. Predictable costs ensure that users can submit transactions and be sure that they will validate (or if they don't, it will be for other reasons).

See also Requirement 2.2.

Requirement 4.5 (Evaluator abort)
The evaluator must be able to run scripts with limited resources and stop them when they exceed it, so we actually enforce budgets.

Requirement 4.6 (Intuitive costs)
Programmers will want to reason about costs when writing their programs. So ideally costs should track those intuitions. That way, if a user does something that they expect to reduce the cost of the program, then it probably does do so.

Requirement 4.7 (Specifiablity)
We want to formally specify how the costs for a script should be calculated, which means that the method for computing them must be relatively simple and manageable to specify.

Requirement 4.8 (Low overhead)
Tracking costs during script execution should not impose too much overhead. We don't want the process of preventing scripts from running too slow to itself slow them down significantly!

Requirement 4.9 (Parameterizable models)
We are unlikely to get all the numeric choices about how to assign costs correct first time. Or, they might be invalidated by hardware or software changes. Hence we should parameterize the model so that the ledger can change the parameters (e.g. with a protocol parameter update).

Requirement 4.10 (Language-agnostic ledger support)
The ledger will support many different languages, which may think about costing in quite different ways. We want to make this as easy as possible, ideally by abstracting away some of the details.

Requirement 4.11 (Block budgets)
We have some constraints on how long block validation as a whole can take, in order to ensure that blocks can propagate across the network fast enough. In addition, we have limits on how much peak memory we can use in a block, based on our expectations of typical hardware.

We may therefore need hard limits on resource usage per-block, in addition to setting prices for resource usage.

4.2 Non-Requirements

There are some things which we could have designed for, but we have chosen not to. A few notable examples are given here.

Units of cost should be persistent across transactions On Ethereum, gas which a contract does not spend in a single transaction is persistent - it can be used later. We don't intend to replicate this: because execution costs are predictable, it is no problem to require users to provide exactly the amount that they need. Moreover, persistent cost units potentially allows undesirable arbitrage where a user can "buy" computation when it is cheap and "spend" it later when it is expensive, which is not what the network wants.

Users should be able to calculate script execution costs for any input to that script Users often care about the "total cost of operation" of an application. But this may consist of running the same script with many different inputs across time, depending on contingent facts which will not be known until later. To give this information to users precisely we would need to be able to predict, for any input, what the execution cost would be. This is something that has been studied, but we believe it would be a lot of work to apply to our situation. [7] Users will still be able to get some idea of total cost of operation by running their script with a selection of indicative inputs.

Slot leaders should be compensated for the opportunity cost of memory usage We want to enforce a peak memory limit to avoid crashes, but we don't bother compensating slot leaders for the opportunity cost of using their memory.

4.3 High-level approach

This section gives a very high-level overview of the approach we take to the problem. Further details on each of these topics are given later.

Abstract resources Rather than computing costs directly in Ada or in real resource units (e.g. seconds), we instead compute them in *abstract resource units*, which do not have a definitional link to real resources. The reasons for this are that:

- Costs must be deterministic and specifiable (Requirements 4.4 and 4.7), so we cannot use *real* resource units which will differ by machine.
- The decision for how to convert these resource units into Ada ("pricing") can be the ledger's responsibility alone.
- Language-specific costing solutions don't need to worry about the (potentially strategic) concerns that affect pricing (Requirement 4.10).

[7] There is research in doing this for languages with explicit recursion and datatypes. We would probably want to do this analysis on Plutus IR, and then try and translate the analysis down to Plutus Core. This seems like a substantial amount of work, probably a PhD thesis or so.

Limiting execution The evaluator can run in a mode ("restricting") where it is given a budget of abstract resources, and terminates if the execution exceeds that (Requirement 4.5). We also provide another mode ("counting") where we do not take a budget, and instead return the minimum budget that the script requires.

4.3.1 Cost model

We call the system which tells the evaluator how much an script should cost a "cost model".

Cost model strategy The cost model for Plutus Core is defined in a fairly simple, compositional way by giving costs for individual operations, and then accumulating them over the course of the evaluation.

This cost model naturally has a lot of parameters: the numbers that influence the costs for the individual operations. We allow all of these to be changed, so the evaluator accepts a bag of parameters which gives all of these values. These parameters will be put into the protocol parameters, so we should have a large amount of flexibility if we need to tweak the cost model later (Requirement 4.9).

Choosing the cost model parameters The decoupling of the abstract resource units from the pricing units allows us to use a fairly simple rubric for picking cost model parameters: try and actually follow (be correlated with) real costs!

This ensures that resource costs will follow programmers' intuitions (Requirement 4.6), while giving plenty of flexibility in the final pricing of those resources.

If we later change our minds about the cost model parameters (e.g. because what we consider "typical hardware" changes), we can re-calibrate our parameters and submit a protocol parameter change.

However, we do want our choices to generally be an *over-approximation*, because anywhere we under-approximate is a potential source of attacks (Section 4.7).

4.3.2 Pricing model

The conversion from abstract resources to Ada is done by the ledger. We call the system which tells us how to do this conversion the "pricing model".

Pricing model strategy The current proposed pricing model simply consists of an Ada price for each of the abstract resources. So a budget in terms of Space and Time will be turned into a price in terms of Ada by multiplying the components of the limit by their respective prices.

These prices also obviously function as parameters for the pricing model. Hence they will also be put in the protocol parameters, so that pricing can be changed with only a protocol parameter change.

Choosing the pricing model parameters Choosing the pricing model parameters is a complex problem that may have strategic considerations. At minimum, the prices will need to be high enough to avoid DoS attacks (Requirement 4.2).

4.3.3 Validating scripts

Putting the pieces together, what happens when a script is validated is:

- The ledger uses the pricing model parameters to compute the price for the stated resource budget in Ada, and ensures that the transaction has sufficient fees to cover this, and that it does not cause us to exceed the per-block budget (Requirement 4.11).
- The ledger runs the script, passing in the stated budget and the cost model parameters.
- The evaluator either terminates normally, with success or an error depending on the program, or stops early if it runs out of budget.

4.4 Abstract resource units and resource budgets

We use two abstract resource units:

- Abstract time/CPU usage ("Time")
- Abstract peak memory usage ("Space")

Script execution is limited in both Time and Space. The overall limit (budget) is therefore a pair of some amount of Time and Space.

Crucially, the interpretation of the Space limit is that it is a limit on (our over-approximation of) *peak* space usage. This is because the limit is mainly to prevent crashes due to exceeding a machine's available RAM, not to compensate the slot leader for RAM usage.

The reason we need to track memory at all is because our evaluators can use unbounded memory. This is not true in e.g. Ethereum where there is a (small) stack limit, so programs can be assumed to fit within a constant memory budget. Plutus Core has both unbounded integers and unbounded recursion, so it is not hard to use a large amount of memory.[8]

4.5 Building the cost model

The high-level description of the cost model said that we would give costs for "individual operations". This means that the model is tied to exactly what kind of evaluator we use. At the moment we use a fairly standard CEK machine.

Michael: Ref into PLC section

Our goal in producing a cost model is to make the computed costs be well-correlated with the real costs. Since the pricing model can be used to control how these are paid for, we can stick to trying to track reality. Hence we start by benchmarking individual parts of execution, and using statistical methods to infer model parameters from that.

4.5.1 Mapping between real and abstract units

Any benchmarks we run will give results in actual units (e.g. microseconds), not our abstract units. So in order to actually use them we need to do one of two things:

1. Set a target mapping for how real units should correspond to abstract units, e.g. 1 Time = 1 microsecond. We can then use this to interpret our benchmarks as benchmarks of ideal abstract unit usage.
2. Pick one of the parameters as a baseline and relativize all the others to that. For example, we could declare that 1 machine step will take 1 Time, and then divide the times for other operations by the time for a machine step to get their abstract Time usage. Then the abstract units would indicate "resource usage relative to the chosen baseline operation".

Option 1 has two advantages:

1. It gives us a way to calibrate abstract resource usage across multiple scripting languages, which is important if we want to use (and price) the same units for all of them.
2. It gives us an obvious way to say how many real units correspond to an abstract unit, which is useful for constructing the pricing model.

So we adopt option 1.

The specific targets we are using at the moment are:

- Time: 1 Time = 1 microsecond
- Space: 1 Space = 1 machine word (8 bytes on a 64 bit machine)

[8]Most programs that use lots of memory will also use lots of time, and we could price them punitively to ensure that the time budget was always a binding constraint. But this requires some subtle reasoning: it is easier to simply talk about memory if that's what we care about.

4.5.2 Simplifying assumptions about memory usage in Haskell

Memory usage in Haskell is quite complicated. However, we can simplify the situation by making some assumptions.

No memory is ever garbage-collected or freed This is a pessimistic assumption, but being pessimistic is fine since we're aiming for an overestimate. It simplifies our calculations, since it means that we can just track all *allocations* of memory, and don't have to worry about the details of when things are de-allocated.

All references to the AST are shared This is an optimistic assumption, but justified by our knowledge about how Haskell works. This means that we can work out how much memory the AST takes up, and then we can assume that references to parts of the AST (e.g. in variable environments) take no memory for the AST part. That simplifies our accounting significantly, since the new allocations will be for small things like entries in variable environment mappings.

4.5.3 Space for the AST

We need to compute Space usage for the AST in several places:

- At the start of execution to account for loading it into memory and to handle references (see the above discussion of sharing).
- When evaluating a builtin operation (see below) the cost may depend on the size of the arguments.

For the AST itself we follow a fairly simple scheme where we make some assumptions about how much space the nodes themselves take up, as well as their children. For constants, we simply think about the underlying type: how much space do we think an integer, say, takes up?

4.5.4 Space and Time for builtin operations

Builtin operations are in some ways the trickiest thing to cost, since they are very heterogeneous, and are implemented by external code. Moreover, their performance characteristics often depend on the inputs to the function: adding two 1000000 digit numbers is much slower than adding two 10 digit numbers.

Size of builtin arguments We use the Space usage of an argument (typically a constant term) as a proxy for its size. This is not perfectly precise, but it is reasonably correlated with the measures of size that e.g. addition is likely to care about.

Builtin accounting happens before execution Builtin operations can in principle use a lot of resources just in the operation itself. For example, multiplying two sufficiently large numbers can require a large amount of memory to hold the result, much larger than that required to hold the arguments. This leads to a conflict: builtin operations are atomic (in that we can't interrupt them), but we want to interrupt evaluation as soon as we exceed our resource budget.

Our solution is to compute the costs for a builtin operation before we run it. Since we are not actually measuring anything, but rather computing the cost from our model, we are able to do this. That way we can terminate evaluation if we would exceed the budget after running the operation, without actually running the operation.

Time for builtin operations The approach we take for Time is very empirical:

1. Micro-benchmark the builtin with arguments of varying sizes.
2. Look at the resulting data and try to pick an appropriate linear model for the running time based on the argument sizes (e.g. "linear in both arguments").[9]
3. Use linear regression to infer the parameters for the model, these are the cost model parameters for this builtin.

[9]This is an imprecise approach, but choice of statistical models is always more of an art than a science.

Space for builtin operations The approach we take for Space is more deductive. Generally it is hard to measure memory allocation, especially when it occurs in a foreign library. So we largely rely on reasoning about or inspecting the libraries in question. Many perform no allocation, or a predictable amount of allocation.

4.5.5 Space and Time for machine steps

The CEK machine itself has many steps that it goes through, which correspond to handling e.g. application of user-defined functions. These also need to have costs associated: naive profiling suggests that the operation of the machine itself (as opposed to builtin functions) is usually more than 50% of evaluation time.

Time for machine steps The approach we initially took for Time is empirical, matching the approach for builtins:

1. Assume that the execution time is linear in all the kinds of machine steps.
2. Benchmark execution of a large number of varied programs that use all the execution steps.
3. Use linear regression to infer the parameters for the model, these are the cost parameters for the machine steps.

However, when we actually did this, we found that the number of machine steps of different kinds are almost perfectly correlated, such that there is little point trying to infer coefficients for them individually. So instead we fit a single-parameter linear model based only on the total number of steps.[10]

Space for machine steps The approach we take for Space is more deductive. Generally we assume that machine steps incur memory usage in certain specific ways, such as creating machine frames or creating mappings in variable environments, and we handle these specifically.

4.5.6 Summary of overall calculation

To sum up, the cost for a program execution is:

- The initial Space cost of the AST.
- The initial Time cost for starting the machine (the intercept of the linear model for the machine steps).
- The Time and Space costs for each machine step that is taken (as determined by the coefficients of the linear model for the machine steps).
- The Time and Space costs for each builtin call (as determined by the linear models for each builtin operation).

4.6 Building the pricing model

The pricing model is out of our control - it is the ledger's responsibility, and as we discussed earlier, the process of choosing the model is likely to be complex and influenced by many non-technical factors. However, anyone making a decision about how to set prices does need at least some input from us, because prices relate to real costs (real CPU seconds, real memory usage), and so we need to know how our abstract units correspond (in reality, at the current time) to real units.

Fortunately, we have this information easily to hand, since we are targeting specific relationships between abstract units and real units when we build our models. So we can assume that those relationships hold (e.g. 1 Time = 1 microsecond).

[10]The argument from correlation holds even if the different steps do actually have very different execution costs! It's possible that they are all indistinguishably similar, but even if they were quite distinct, if they're very well correlated a single-parameter model will still do a better job.

4.7 Security

The top priority of the model is to ensure the security of the ledger against attacks, particularly DoS attacks, but also economic attacks that sap profit from the system.

Basic structure of an attack The basic kind of attack we are worried about is where we have set one of our cost parameters too low, effectively under-costing a particular operation. For example, maybe addition is too cheap, or a certain kind of machine step.

An attacker could then construct a synthetic program that disproportionately uses the under-costed operation. Such a program would then be more expensive to execute in reality than the model predicts, allowing the attacker to force node operators to do more work than they are paying for.

How dangerous this is depends on how badly we under-estimate the parameter. If we under-estimate it by only 10%, then an attacker is only getting a 10% "discount" on computation: probably not enough to make it worthwhile to mount an attack. If we under-estimate it by an order of magnitude or two, then we could be in trouble.

We might think that such malicious programs would be large, and so transaction size limits would help us. But we have loops and recursion in Plutus Core, so it is likely that an attacker could make a relatively small program that does a very large number of the problematic runtime operations.

Simple prevention approaches Our attack prevention approach is fairly simple. We try and make the cost model give an *over*-estimate of reality, and then we try to make the pricing model over-price the resources.

In order to keep this working, we need to:

1. Ensure that we take great care when updating the cost model, especially if we make things cheaper (say, on the basis of the evaluator getting faster)

2. Ensure that we adjust prices as appropriate when the cost of hardware changes.

Tools for node operators to check accuracy of models on their hardware Our benchmarks are going to be run on a reference machine. Of course, not all machines are alike, and it's possible that our hardware may be unusual, or certain node operators' hardware is unusual in such a way that allows an attack.

For example, perhaps addition is unusually slow on operator O's machine: then a program which does lots of addition might be costed cheaply, run fine on our reference machine, but overload O's machine.

A simple way to mitigate this risk is to provide tools for node operators to run the benchmarks on their own hardware. If they get results that indicate that costs should be higher, then that indicates a potential attack and we may need to raise costs.

5 Applications: the Plutus Application Framework

An application that interacts with the blockchain is some kind of program that runs on a users computer. But what does that application actually do?

For starters, Plutus applications that make use of the ledger's scripting functionality will need to create appropriate Plutus Core programs. We discuss how we enable this for Haskell applications in Section 6.3.

However, even very simple applications have some clear needs.

Consider one of the simplest possible Plutus applications:

Metadata Poster Metadata Poster does nothing except occasionally submit transactions to the chain. The transactions which it submits do not move any substantive amount of value, their purpose is just to post a transaction to the chain with some metadata payload that can later be checked by another application.[11]

Firstly, Metadata Poster needs to communicate with a number of other components:

[11]This may seem like a silly example, but many real proposed applications in supply-chain management are essentially just Metadata Poster!

- It needs to submit transactions, so it must talk to a node or a wallet backend.
- It needs to acquire funds to pay fees, so it must talk to a wallet backend.
- It must talk to its users, so it must expose some kind of API or talk to a graphical interface like a wallet frontend.

Moreover, Metadata Poster may care about some of the rollback issues discussed in Requirement 5.8. Finally, there are a number of operational issues common to Plutus applications:

- It may need to be distributed to end users and receive updates.
- It may need to synchronize its state between multiple instances (e.g. desktop and mobile).
- It may need to have its state backed up by systems administrators.
- It may need to provide logging and monitoring for production usage.

It is clear that there is a lot of complexity in the off-chain code part of a Plutus application. Enough that we probably cannot simply leave this in the hands of application developers. The Plutus Application Framework (PAF) is our response to this problem: a disciplined framework for writing Plutus applications that eases many of these problems.

5.1 Requirements

Requirement 5.1 (Backups)
Plutus applications need to be easy to back up, if they are to be used in production.

Requirement 5.2 (Monitoring)
Plutus applications need to be easy to monitor, if they are to be used in production.

Requirement 5.3 (Synchronization)
It should be possible to synchronize the state of an Plutus application between instances on multiple machines, e.g. a mobile and a desktop instance. This is quite important for consumer-type users.

Requirement 5.4 (Reproducibility)
Plutus application behaviour should be reliable and reproducible on different environments and devices. For example, application instances that have had their state synchronized (Requirement 5.3) should behave identically.

Requirement 5.5 (Distribution)
Plutus applications need to be distributed to users somehow. Different users may have different needs here, for example:

- A consumer user may want to download an Plutus application from a centrally managed “app store” in their wallet frontend.
- A business user may want to download a native application via their usual package manager, or directly from the author.

Requirement 5.6 (Flexible, self-describing application endpoints)
Plutus applications will want to expose endpoints to users which trigger the functionality of the application. These need to be accessible to both server-side headless consumers, and also to graphical wallet frontends which mediate interaction with end-users.

Ideally, these endpoints would be *self-describing* so that we can have at least basic generic interfaces in e.g. a wallet frontend.

Requirement 5.7 (Chain data access)
Plutus applications need to access some historical data about the chain. In particular, due to Requirement 2.4 the datums for script outputs are not stored in the UTXO set, but rather in the transaction that creates the output. Plutus applications will need to know about these datums, so we must provide some way of tracking this information from the chain and making it available.

Michael: Should link to wherever we discuss the whole issue with storing datums in detail.

Requirement 5.8 (Rollback resistance)
Rollbacks can cause serious problems for agents (not just applications) trying to take conditional actions. It would be nice if we could mitigate these for Plutus applications, but that may not always be possible.

Here are two scenarios we might care about.

Incoherent choice Suppose that Alice promises to send 10 ada to Bob, provided that Bob sends 20 ada to Carol (perhaps Alice is holding Bob's collateral for a loan from Carol, which Bob is now repaying). The following events occur:

1. Bob pays 20 ada to Carol in transaction T1.
2. Alice observes T1, and proceeds to pay 10 ada to Bob in transaction T2.
3. A rollback occurs. After the rollback, T1 and T2 go back into the mempool, but T1 is now invalid.
4. T2 alone is reapplied.

As a result, Alice ends up making the payment to Bob without Bob paying Carol, so Bob gets away with all the money! Alice ends up committed to an action that she would only have chosen to do the old history of the chain, and which she would not have chosen to do the new history.

Incomplete reapplication Suppose as a variant of the previous scenario that Alice promises to send 10 ada to both Bob and Carol, provided that some off-chain event happens. The following events occur:

1. The off-chain event occurs.
2. Alice pays 10 ada to Bob in transaction T1.
3. Alice pays 10 ada to Carol in transaction T2.
4. A rollback occurs. After the rollback, T1 and T2 go back into the mempool, but T1 is now invalid.
5. T2 alone is reapplied.

As a result, Alice ends up only paying Carol and not Bob. Alice ends up *partially* taking an action that she still wants to take, and would need to reconstruct the missing parts to get back to the state she wants to be in.

Requirement 5.9 (Testing and emulation)
Users need to be able to test their Plutus applications in an environment that mirrors the real one as closely as possible. However, the real environment is very complex, featuring a multi-agent, distributed system with a number of tricky behaviours: network issues, rollbacks etc.

It is therefore desirable to provide some kind of emulated testing harness which users can use to test their Plutus applications locally, but which allows control and simulation of real issues.

Moreover, this is important for us during development, as it allows us to mock up the system that we expect without having to wait for other components to be ready.

5.2 Lifecycle of a Plutus application

The lifecycle of a Plutus application is as follows:

- Plutus applications are authored and compiled with the Plutus Haskell SDK using the contract API for interacting with other components.
- Plutus applications are distributed via some means to be decided, but manually in the interim.
- Plutus applications are installed into an instance of the Plutus Application Backend (PAB). The PAB just knows about the compiled application executable provided by the Plutus application.
- A Plutus application can be instantiated into a application instance by running the application executable and providing any parameters that it needs. There can be multiple application instances per Plutus application, and they are managed by the PAB.
- The PAB manages and handles the requirements of the application instance throughout its lifecycle, including interaction with external clients such as wallet frontends.

The major component here is the PAB.

5.3 The Plutus Application Backend

WARNING: this component is under heavy development, so this will likely evolve and may not represent the current state of things.

A key component of the Plutus Application Framework (PAF) is the Plutus Application Backend (PAB). This is a backend service (like the wallet backend) that intermediates between Plutus applications, the node, the wallet backend, and users (including the wallet frontend).

The PAB will be run in similar contexts to the wallet backed, e.g. backing a graphical user wallet (e.g. Daedalus), or on a server that runs Plutus applications as part of a larger system.

The purpose of the PAB is to:

Michael: Do this in prose? Also all the provisions here should be expanded and moved to requirements

- Provide a standardized environment for Plutus applications to run in (Requirements 5.2 and 5.4)
- Provide disciplined state management (Requirements 5.1, 5.3 and 5.8)
- Present discoverable interfaces to the external clients (Requirement 5.6)
- Track information from the chain for use by contracts (Requirement 5.7)
- Work in an emulated environment (Requirement 5.9)

The PAB is a series of components which produce/consume events, and a message bus.

Some of the components have additional complexity, e.g. the application management component needs to manage the state of application instances.

5.3.1 Node client

The PAB needs to talk to the node, primarily because it needs to populate the chain index, but it also needs to watch the stream of incoming transactions and rollbacks, and notify the application instances of changes to transactions that they are interested in.

5.3.2 Wallet backend client

The PAB needs to talk to the wallet backend for a number of things:

- Coin selection/transaction balancing
- Transaction signing and submission
- Address creation

You might think that since the PAB has a node client itself, it could do its own transaction submission, and only rely on the wallet backend for signing. However, transactions made by the PAB will likely use outputs "owned" by the wallet backend (e.g. those selected by coin selection from the user's outputs). Hence it is important that the wallet backend knows about such outputs, so that it does not attempt to spend them somewhere else.

5.3.3 Concurrency

application instances managed by the PAB spend most of their time waiting for changes to the blockchain, user input, or the passage of time. When they are not waiting, they are making requests to services managed by the PAB, for example the chain index or the wallet backend.

We are currently using an event-sourced architecture here. However, we plan to switch to a simpler database model.

5.3.4 Application management

Application instance need to be managed, created, destroyed, fed with events, etc.

- Create application instances
- Instantiate and run application executables in a sandbox
- Handle communication with the application executable
- Mediate requests to PAB servicess by the application instance
- Manage/dump/load application instances state
- Create/destroy application instances
- Handle rollbacks

5.3.5 Chain index

Applications need to access datums for outputs (see Requirement 5.7), so we need some kind of system that monitors the chain and records (at least) the datums.

5.3.6 Client interface

For external clients (other programs), including graphical wallet frontends to talk to. Should expose some of the application endpoints and application instance management functionality.

5.3.7 Logging and monitoring

To satisfy Requirement 5.2.

5.4 Emulators

In order to satisfy Requirement 5.9, we need to write emulators for quite a number of components.

At present, we have (or expect to have) emulators for:

- The node using our ledger extensions. In the long run we should be able to use the real Goguen node.
- The parts of the wallet backend that we need. In the long run we will be able to use the real wallet backend.
- Basic wallet frontend functionality, such as displaying balances and interacting with application instances. In the long run we *might* be able to use the real wallet frontend, but this seems unlikely as it is quite heavyweight. Having our own component here has the advantage that we can reuse it in the Plutus Playground.

We also need libraries to bind all of these into an overall, multi-agent simulation, and to allow users to write tests that exercise particular series of events in this simulation.

5.5 The Plutus Playground

The Plutus Playground provides a Web environment for getting started with the Plutus Platform.

The authoring experience in the Plutus Playground is fairly limited (one file only), but it has the best support for specifying ad-hoc scenarios and visualizing the results.

Over time we hope to unify the experiences of working locally and working in the Plutus Playground, by:

- Improving the authoring experience in the Plutus Playground (multiple files etc.)
- Improving the visualization experience locally (sharing components with the Plutus Playground)
- Allowing distribution of simple Plutus applications directly from the Plutus Playground.

5.6 Application design

Michael: Talk about state machines and our ideas for handling rollbacks.

Plutus applications are distributed applications whose state is spread across multiple processes. One of those processes is the blockchain, or (operationally) the set of Cardano nodes that verify transactions. Here the state of the Plutus application takes the form of script outputs. There is no one-to-one correspondence between Plutus applications and script outputs, or even addresss.

5.6.1 On-chain

To reason about the behavior of the on-chain parts of Plutus applications we use a type of state machines called constraint-emitting machines (Constraint-emitting machine (CEM)), state machines that produce constraints on the next transition in every step. This approach has been published in [2]. The Plutus Haskell SDK offers support for writing CEMs in Haskell.

An advantage of writing scripts as CEMs over directly writing the validator script function in Haskell is that the constraints can be used not only to verify the spending transaction on-chain, but also to construct it off-chain. Building a transaction that spends an output with a hand-written validator script often involves code that is very similar, but not quite identical, to the validator code itself. With CEMs we can capture exactly this overlap and reduce duplication.

5.6.2 Off-chain

Plutus applications react to events that happen either on the blockchain or outside the Plutus Platform. The following types of events can be reacted to:

- Wall clock time progresses.

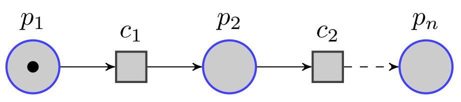

Figure 2: Petri net modeling the passage of time as observed by Plutus applications

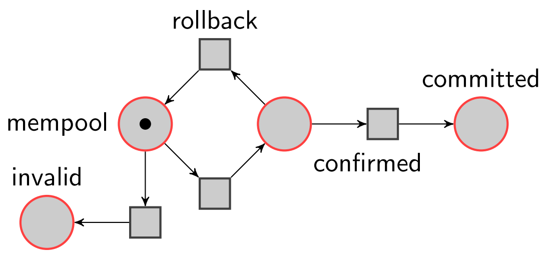

Figure 3: Petri net for the status of transactions

- The status of a transaction changes as a result of transaction validation or a rollback.
- The set of unspent outputs at an address changes as a result of a transaction status change.
- Input is provided to the Plutus application from outside the system.

The meaning of these interactions is described by the following Petri nets.

Slot change For each slot s there is a place p_s. A *clock* transition c_s takes a token from p_s and places it in p_{s+1} to signal that slot $s+1$ has begun. Any other transition that removes a token from p_s is expected to put it back immediately, so that there is always exactly one token in p_s after it has been filled for the first time. See 2 for an illustration.

Transaction status change The status of a transaction changes multiple times after it has been sent to the node. Transactions start out in the node's *mempool*. Then their status changes to *tentatively confirmed* or to *rejected*. Finally, a transaction that is tentatively confirmed can revert back to *mempool* or it can become *permantently confirmed* when enough blocks have been added to make it irreversible.

For each of the four states of a transaction there is one place in the petri net, as shown in 3.

Address change The set of unspent outputs at an address is modified by transactions that spend and produce outputs. Therefore, whenever the status of a transaction changes, the status of its inputs and outputs changes also.

We represent the outputs at each address with two petri nets, one for unspent outputs and one for spent outputs. There is one place each for outputs that are in the mempool, tentatively confirmed, permantently confirmed, rejected.

Endpoint Users and other applications may call endpoints on the Plutus application. Endpoints are places $e_1, \ldots, e_n$ in the petri net (see 4).

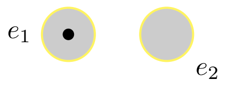

Figure 4: Petri net for endpoints. The token in e_1 signifies that input is available to be consumed by this contract.

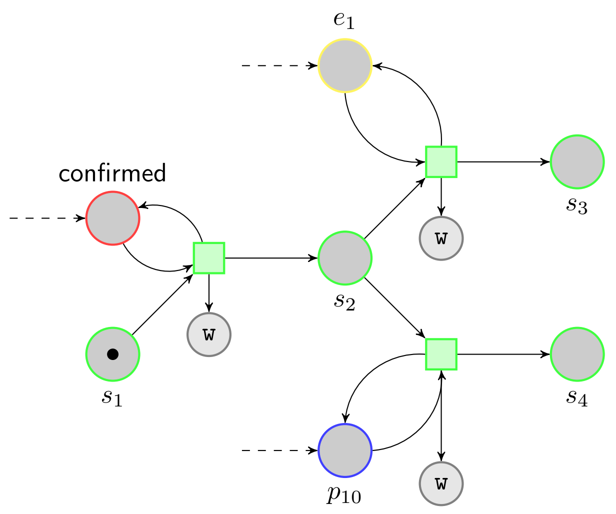

Figure 5: Petri net for an Plutus application (green) that waits for a transaction to be confirmed and then waits for slot number ten to begin, or for user input. The app emits values of type `w` on every transition.

Apps Given the places for time, transaction status and endpoints we can describe Plutus applications as sequences of transitions. The states of the app are represented by places. 5 shows a Plutus application with three possible state, s_1, s_2 and s_3. As soon as a transaction is confirmed, the state can progress from s_1 to s_2.

After that, endpoint e_1 becomes *active*, meaning that the app can make progress as soon as input is provided. The next state depends on which of two possible events happens first: The endpoint being called by the user, or the clock reaching slot ten.

The app transitions (green) involve queries to the chain index, transaction submission, etc. These requests are not shown in the petri net. We still record their responses however, in order to meet the replayability requirements (see 5.3 and 5.4).

Observable state Plutus applications need to be able to notify the outside world of changes. To this end, the application can emit values of some user-defined type `w` whenever one of its transition fires. In the Haskell library, this is realised using the `Writer w` effect, with `Monoid w` constraint. Clients of the Plutus application can subscribe to receive updates whenever the accumulated total of all values changes.

Extensions There are some possible extensions of the basic model of apps and events. For example, we could model an on-chain state machine (in a Plutus script) as a petri net. Then we could describe the interactions of multiple application instances interacting with that state machine, including possible race conditions. The petri net model is simple, but it scales easily over multiple machines and contracts.

6 Authoring: the Plutus Haskell SDK

Just like web applications, Plutus applications have two separate components which are deployed in separate execution environments:

- on-chain code which is stored on the blockchain and executed during the validation of new transactions (similar to the server component of a web application), and
- off-chain code code which is deployed to and executed on the client machine of a blockchain user with access to the user's wallet (similar to the client portion of a web application running in a user's web browser[12])

[12]In fact, in other systems off-chain code often executes in a web browser as well!

Why is this decomposition necessary? The on-chain code contains the Plutus application's enforceable components. It needs to enforce that only transactions that meet the enforceable obligations are successfully validated and added to the chain. In other words, the integrity of an Plutus application depends on the integrity of the on-chain code: thus we need to store it on the cryptographically immutable blockchain to prevent tampering. Moreover, slot leaders need to execute on-chain code to check that it is in fact satisfied.

Conversely, the off-chain code, which submits new transactions to the chain for validation and inclusion, necessarily needs to run in close association with a contract user's wallet. After all, each transaction needs to be paid for with transaction fees, and a wallet is the only place where the necessary credentials are held (anything else would compromise the security of the funds).

Existing blockchains and their smart contract and dapp frameworks use separate languages for the on-chain code and off-chain code (in Ethereum, Solidity and JavaScript), and they tend to invent new languages for the on-chain component (e.g., Solidity). This comes with the same disadvantages as using different languages for the client and server component of web apps. However, when new languages are invented the situation is even worse because of the enormous overhead involved in creating a new language, compilers and other tools, libraries, teaching material, and generally growing a new language community. We would like to avoid this (Requirement 6.1).

We overcome these problems by using Haskell for both on-chain code and off-chain code code. This enables us to build on the existing Haskell ecosystem and to share datatypes and code between the two. However, a downside of this approach is that GHC forms part of our compilation toolchain, which we do not control and which may change unexpectedly (this can make it hard to satisfy Requirements 6.3 and 6.4, for example).

The Plutus Haskell SDK is our library and tooling support for writing Plutus applications — both their on-chain code and their off-chain code together.

6.1 Requirements

Requirement 6.1 (Conservatism)
Designing a new programming language is hard. Designing a new *source* programming language (one written by users directly) is even worse. One must worry about syntax, tooling, build systems, libraries and their ecosystem, etc.

Ideally, we would like to avoid all this by reusing existing languages as much as possible.

Requirement 6.2 (Lifting values at runtime)
The programs which we put on the chain cannot be entirely static (i.e. determined at off-chain code compile time). It must be possible to parameterize them or partially generate them at runtime.

One reason for this is that we may want to configure our code. For example, a crowdfunding Plutus application might want to parameterize its on-chain code by the crowdfunding target, or the beneficiary of the funds.[13] How can we parameterize a Plutus Core program? One way is to write the program as a function, compile that function statically, then construct the argument at runtime and apply the program to the argument. In pseudo-Haskell:

```
compile (\parameter -> ...) ‘apply‘ lift argument
```

This requires a way to "lift" a runtime value into an appropriate Plutus Core term so we can actually apply our compiled program to it.

This is just generally a very handy thing to be able to do and we want to be able to do it.

Requirement 6.3 (Stability)
When the Plutus Core program that makes up the on-chain code of a Plutus application changes, so does its hash, and hence its address. This can be a big problem for applications: you cannot spend a script output without presenting a script with *exactly* that hash. If your tooling won't produce such a script any more, then you can't get the money!

[13]In some instances one can get away with putting this information in the datum. The crowdfunding example is interesting because many people pay to the address spontaneously, and the owner cannot control what those people put in the datums. But they can control the address people send to, and hence the script. So it is necessary to bake the parameters into the script in this instance.

We therefore want our tooling to be as stable and deterministic as possible, so we don't change the output unnecessarily.

At the very least, we must not be sensitive to:

- The platform we are working on (linux/macos etc.)
- Any random or mutable conditions

Requirement 6.4 (Compilation reproducibility)
As discussed in Requirement 6.3, if the output of compilation changes unexpectedly, then that can be a big problem. But if the user changes their source or their tooling (e.g. their version of GHC), then that may just genuinely change the input to our compiler.

In this instance there is not a great deal we can do at a technical level, except help people to build their applications reproducibly, so that they can at least revert to previous states reliably.

6.2 Haskell

We need a source language for users to write Plutus applications in. We don't want to write our own one (Requirement 6.1), so we would really like to reuse an existing one. We decided to use Haskell for a number of reasons:

- It is a powerful functional programming language.
- It has an industrial-grade compiler, adequate tooling, and a good community and ecosystem.
- It has good metaprogramming facilities.
- We are familiar with it as a team and a company.

However, this means we need a way to compile (as subsection of) a Haskell program into Plutus Core. Our solution to this is Plutus Tx.

6.3 Plutus Tx

The Plutus Tx compiler compiles GHC Core into Plutus Core. That is, it takes Haskell after it has been desugared from its source representation into GHC's internal representation (GHC Core), and compiles that further. This approach allows us to support all of source Haskell while only having to deal with the much smaller GHC Core language.

6.3.1 Plugins for custom compilation

GHC core-to-core plugins enable us to inject our own code including the Plutus Tx compiler into the GHC pipeline. The Plutus Tx plugin,

1. locates GHC Core fragments representing to on-chain code,
2. compiles them to Plutus Core, and
3. replaces each GHC Core AST subtree representing on-chain code code with an AST representing a serialised version of the generated Plutus Core.

Overall, we end up with compiled off-chain code that embeds blobs of on-chain code in its serialised Plutus Core representation, ready to be submitted to the blockchain attached to transactions generated by the off-chain code.

There just seem to be two problems: 1. how does the plugin identify on-chain code and 2. how do we ensure that the type of the serialised on-chain code lines up with the source code?[14] We achieve this using a trick that to the best of our knowledge was first used in the `inline-java` package embedding Java into Haskell. This packages uses GHC plugins to extract type information at a Template Haskell splice point

[14]GHC Core is a typed intermediate language; hence, any code transformation needs to be type-preserving.

[7]. The idea is to wrap the target Haskell code inside a splice of a Template Haskell function that inserts a marker around that AST fragment.

This function does not actually compile the AST of the target program fragment. Instead, it inserts a marker function that is picked up by the Plutus Tx compiler injected with the plugin. When the plugin runs, it finds the marker, compiles the code, and inserts the serialized form back into the program. By taking a little care over the types of our marker function, we can ensure that the expression in question remains well-typed at each stage of this process.

6.3.2 Compiling GHC Core to Plutus Core

Both GHC Core and Plutus Core are extensions of System *F*. GHC Core is a much more generous extension. It adds mutually-recursive binding groups, algebraic data types, case expressions, coercions, and more. In contrast, Plutus Core, as discussed in Section 3, is much more minimal.

How do we deal with the extra features of GHC Core? First, we split the problem in half, by defining an intermediate language, Plutus IR (see Section 6.4), which is much closer to GHC Core. Most of the theoretical complexity is therefore moved to the Plutus IR compiler.

The remaining work of the Plutus Tx compiler is then to *lower* the GHC Core terms and types into their corresponding Plutus IR variants, emitting errors as appropriate if we encounter features we do not support.

6.3.3 Supporting Haskell's features

As alluded to in the previous section, we do not support the entirety of Haskell. Thanks to the design of GHC, we get a great deal for free, as we compile programs after they have been converted to GHC Core, which means that most of the complex source-level features of Haskell have already been desugared into a smaller set of simpler features.

While we support most "standard" Haskell, there are quite a few things we do not support. A non-exhaustive list of features that we do not support is:

- Not implemented yet
  - Mutually recursive datatypes (should be done by release)
- Incompatible with the design of Plutus Core
  - `PolyKinds`, `DataKinds`, anything that moves towards "Dependent Haskell"
- Technically difficult
  - Literal patterns
- Requires access to function definitions (might be fixed with some GHC work)
  - Function usage without `INLINEABLE` or `-fexpose-all-unfoldings`
  - Typeclass dictionaries
- Use of coercions required
  - GADTs
  - `Data.Coerce`
  - `DerivingVia`, `GeneralizedNewtypeDeriving`, etc.
- Assumes "normal" codegen
  - FFI
  - Numeric types other than integers
  - Unlifted/`MagicHash` types
  - Machine words, C strings, etc.

6.3.4 Strictness

Haskell is a lazy language and Plutus Core is a strict language. How can we compile a lazy language into a strict language efficiently?

The answer is that we handle this partially. We generally compile Haskell as though it were strict, but the key exception is for non-value let-bindings. That is, if we see a let-binding whose right-hand-side is not a value (i.e. may evaluate further), then we compile it as a non-strict let-binding (see Section 6.4.1).

Unfortunately we have no proof that this approach is sound, which is an area for future work.

6.3.5 Lifting values at runtime

We are going to great lengths to compile on-chain code validator scripts at off-chain code compile time. However, we may also need to create some Plutus Core programs from *runtime* values (Requirement 6.2).

Unfortunately we cannot use the main Plutus Tx compiler for this: the Plutus Tx compiler turns Haskell *programs* represented as GHC Core into Plutus Core. It cannot do anything with the *runtime* representation of a Haskell value!

We therefore need to replicate what we *would* do with the Plutus Tx compiler, but at runtime. Fortunately, we can reuse the Plutus IR compiler, which helps a lot. We define a pair of typeclasses inspired by the Haskell typeclasses for "lifting" runtime values into metaprograms: `Lift` and `Typeable`. Our typeclasses look something like this (`Term` and `Type` are the types for Plutus IR terms and types; class constraints on the methods are omitted for simplicity):

```
class Lift a where
    lift :: (...) => a -> m (Term TyName Name ())

class Typeable a where
    typeRep :: (...) => Proxy a -> m (Type TyName ())
```

With some effort, we are able to generate instances for these with Template Haskell, so the burden on users is minimal.

Why do we output Plutus IR here, rather than running the Plutus IR compiler each time and just producing Plutus Core? The reason is that the Plutus IR compiler has some support for *sharing* definitions, and it is important that programs generated from multiple calls to `lift` (e.g. from one implementation calling another, as is common) share the same definition of their shared types.

6.4 Plutus IR

Plutus IR is an intermediate language that sits between GHC Core and Plutus Core. Many of the core ideas are published in Peyton Jones et al. [14], including the complex parts of compilation and the typesystem.

We give a high-level overview of the language here, further details can be found in the above reference.

Plutus IR is essentially Plutus Core, but with the addition of:

- Datatypes, including recursive and mutually recursive datatypes
- Let terms, including recursive and mutually recursive bindings

Compiling recursive datatypes and recursive values are the two trickiest compilation problems, and are covered in Peyton Jones et al. [14].

6.4.1 Non-strict let bindings

Plutus IR has an additional feature which isn't discussed in Peyton Jones et al. [14]: non-strict let-bindings. By default (and in the paper), let-bindings are *strict*, meaning that the right-hand-side of the binding is evaluated before the body of the term.

However, it is useful to support *non-strict* let-bindings, particularly because these correspond more closely to the semantics of Haskell (see Section 6.3.4). We can desugar these into strict let-bindings simply by inserting a `delay` on the binding right-hand-side and a `force` at every use site.

6.4.2 Optimization

We do a small amount of optimization in the Plutus IR compilation pipeline. We don't want to do too much in case we make the generated code too unstable (Requirement 6.3).

Dead code elimination Dead code elimination is a straightforward optimization and close to a clear win: 1. it reduces code size, 2. it makes the code easier to read, and 3. it has no effect on the semantics.

It is particularly helpful as the Plutus Tx compiler introduces definitions for all the builtins, some of which will be unused.

6.4.3 Compilation

The Plutus IR compiler works via a series of small passes that eliminate individual features of Plutus IR in turn, until the remaining program is pure Plutus Core and can simply be lowered into that AST type.

The passes are:

- Non-strict let-bindings into strict let-bindings by inserting thunks
- Type bindings and datatypes into simple type and lambda abstractions
- Recursive term bindings into non-recursive term bindings
  - We do another dead code elimination pass (Section 6.4.2) as this can introduce dead bindings.
- Non-recursive term bindings into lambda abstractions

6.5 Cross-compilation

To support Requirement 5.5, we want to be able to compile Plutus applications into easily redistributable application executables.

The current approach is to target Javascript or WebAssembly as our format for distribution, and leverage cross-compilation of Haskell to actually produce the executables.

6.5.1 Cross-compilers

At the moment IOHK is working on two cross-compilation efforts:

GHCJS GHCJS is a Haskell cross-compiler which targets Javascript [18].

Asterius Asterius is a Haskell cross-compiler which targets WebAssembly [17].

We may use either or both of these in the end.

6.5.2 haskell.nix

Cross-compilation of Haskell projects is not easy. Neither of the major Haskell build tools (`cabal` and `stack`) support cross-compilation well.

To address this issue, IOHK has developed the `haskell.nix` framework for building Haskell projects using Nix [19]. In addition to supporting cross-compilation well, Nix is well-suited to ensuring that builds are reproducible, which helps with Requirement 6.4.

6.6 Developer tooling

Since the Plutus Haskell SDK uses Haskell for development, there is much less need to create specialized development tooling, since generic Haskell tooling will work perfectly well.[15]

It is possible that we may want to develop some tools, particularly for testing and visualization, but this has not been decided yet.

[15]It is true that Haskell development tooling is generally considered to not be very good. However, it is improving rapidly, and while it might be sensible for IOHK to contribute to the community's efforts on this front, that will be significantly less work than developing completely new tools.

Glossary

This document is incomplete. The external file associated with the glossary 'main' (which should be called `plutus.gls`) hasn't been created.

Check the contents of the file `plutus.glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `plutus.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "plutus"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "plutus"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

References

[1] Manuel M. T. Chakravarty et al. "The Extended UTXO Model". In: *Proceedings of Trusted Smart Contracts (WTSC)*. Vol. 12063. LNCS. Also available at https://github.com/IntersectMBO/plutus. Springer, 2020.

[2] Manuel M. T. Chakravarty et al. "UTXO$_{\text{sf ma}}$: UTXO with Multi-asset Support". In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12478. Lecture Notes in Computer Science. Springer, 2020, pp. 112–130. DOI: `10.1007/978-3-030-61467-6\_8`. URL: `https://doi.org/10.1007/978-3-030-61467-6%5C_8`.

[3] Manuel M. T. Chakravarty et al. "UTXO$_{\text{ma}}$: UTXO with Multi-Asset Support". In: *International Symposium on Leveraging Applications of Formal Methods*. Also available at https://github.com/IntersectMBO/plutus. Springer. 2020.

[4] Manuel MT Chakravarty et al. "Hydra: Fast Isomorphic State Channels". In: (2020).

[5] Manuel MT Chakravarty et al. "Native custom tokens in the extended UTXO model". In: *International Symposium on Leveraging Applications of Formal Methods*. Also available at https://github.com/IntersectMBO/plutus. Springer. 2020, pp. 89–111.

[6] James Chapman et al. "System F in Agda, for fun and profit". In: *International Conference on Mathematics of Program Construction*. Springer. 2019, pp. 255–297. URL: `https://hydra.iohk.io/job/Cardano/plutus/papers.system-f-in-agda.x86_64-linux/latest/download-by-type/doc-pdf/paper`.

[7] Facundo Domínguez and Mathieu Boespflug. *GHC compiler plugins in the wild: typing Java*. URL: `https://www.tweag.io/posts/2017-09-22-inline-java-ghc-plugin.html`.

[8] *ERC-721 standard for non-fungible tokens in Ethereum*. URL: `http://erc721.org/`.

[9] *Formal specification of the Plutus Core language*. URL: `https://hydra.iohk.io/job/Cardano/plutus/docs.plutus-core-spec.x86_64-linux/latest/download-by-type/doc-pdf/plutus-core-specification`.

[10] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". Thèse d'État. Université Paris 7, June 1972.

[11] Robert Harper. *Practical Foundations for Programming Languages*. New York, NY, USA: Cambridge University Press, 2012.

[12] IETF. *RFC 7049 - Concise Binary Object Representation (CBOR)*. URL: https://tools.ietf.org/html/rfc7049.

[13] Sam Lindley. “Embedding F”. In: *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming*. 2012, pp. 45–56.

[14] Michael Peyton Jones et al. “Unraveling recursion: compiling an IR with recursion to system F”. In: *International Conference on Mathematics of Program Construction*. Springer. 2019, pp. 414–443. URL: https://hydra.iohk.io/job/Cardano/plutus/papers.unraveling-recursion.x86_64-linux/latest/download-by-type/doc-pdf/unraveling-recursion.

[15] Simon L. Peyton Jones and André L. M. Santos. “A Transformation-based Optimiser for Haskell”. In: *Sci. Comput. Program.* 32.1-3 (Sept. 1998), pp. 3–47. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(97)00029-4. URL: http://dx.doi.org/10.1016/S0167-6423(97)00029-4.

[16] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002. ISBN: 0-262-16209-1.

[17] *The Asterius repository*. URL: https://github.com/tweag/asterius.

[18] *The GHCJS repository*. URL: https://github.com/ghcjs/ghcjs.

[19] *The haskell.nix repository*. URL: https://github.com/input-output-hk/haskell.nix.

[20] *The Plutus repository*. URL: https://github.com/IntersectMBO/plutus.

[21] Joachim Zahnentferner. “An Abstract Model of UTxO-based Cryptocurrencies with Scripts”. In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 469. URL: https://eprint.iacr.org/2018/469.