Functional Blockchain Contracts

— DRAFT — DRAFT — DRAFT —

Manuel M. T. Chakravarty
Roman Kireev
Kenneth MacKenzie
Vanessa McHale
Jann Müller
Alexander Nemish
Chad Nester
Michael Peyton Jones
Simon Thompson
Rebecca Valentine
Philip Wadler
IOHK
firstname.lastname@iohk.io

Abstract

Distributed cryptographic ledgers —aka blockchains —should be a functional programmer's dream. Their aim is immutability: once a block has been added to the chain it should not be altered or removed. The seminal blockchain, Bitcoin, uses a graph-based model that is purely functional in nature. But Bitcoin has limited support for smart contracts and distributed applications. The seminal smart-contract platform, Ethereum, uses an imperative and object-oriented model of accounts. Ethereum has been subject to numerous exploits, often linked to its use of shared mutable state by way of its imperative and object-oriented features in a concurrent and distributed system. Coding a distributed application for Ethereum requires two languages: Javascript to run *off-chain*, which submits transaction written in Solidity to run *on-chain*.

This paper describes Plutus Platform, a functional blockchain smart contract system for coding distributed applications on top of the Cardano blockchain. Most blockchain programming platforms depend on a custom language, such as Ethereum's Solidity, but Plutus is provided as a set of libraries for Haskell. Both off-chain and on-chain code are written in Haskell: off-chain code using the Plutus library, and on-chain code in a subset of Haskell using Template Haskell. On-chain code is compiled to a tiny functional language called Plutus Core, which is System F_ω with iso-recursive types and suitable primitives.

Plutus and Cardano are available open source, and Plutus Playground provides a web-based IDE that enables users to try out the system and to develop simple applications.

DRAFT, 2019, Submitted
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords Smart contracts, blockchains, Haskell, metaprogramming, System F, embedded languages

ACM Reference Format:
Manuel M. T. Chakravarty, Roman Kireev, Kenneth MacKenzie, Vanessa McHale, Jann Müller, Alexander Nemish, Chad Nester, Michael Peyton Jones, Simon Thompson, Rebecca Valentine, and Philip Wadler. 2019. Functional Blockchain Contracts: — DRAFT — DRAFT — DRAFT —. In *Proceedings of DRAFT*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Distributed cryptographic ledgers (commonly called *blockchains*) are shared, immutable, distributed log data structures comprising a sequence of blocks with multiple *transactions* each. The concrete representation of the blocks and transactions uses cryptographic techniques to render the ledger tamper-resistant, and the overall system uses a monetary incentive system to ensure the collaboration of the providers of the distributed computing infrastructure [Narayanan et al. 2016]. Bitcoin, the seminal proof of the feasibility of the blockchain concept, suffers from a range of problems, including excessive energy usage [Hern 2018] and minimal support for custom transaction validation. Without custom validation, the ledger is essentially confined to providing simple accounting functionality.

Subsequent proposals, such as Ethereum [Wood 2014], provide a general-purpose programming language (in Ethereum's case, Solidity [Sol 2019]) to enable almost arbitrarily complex validation rules. However, this additional expressiveness comes at the cost of a semantically complex computational model, typically favouring object-based programming models that introduce shared mutable state into an already concurrent and distributed system. Moreover, they rely on new, custom-designed languages requiring new toolchains, libraries, educational material, supporting communities, and so on. As a consequence, it is hard to formally or informally reason about the behaviour of the resulting applications: this leads to a wide range of vulnerabilities, some of which have become infamous [Falkon 2017].

The issue of semantic complexity is aggravated by the fact that complete *decentralised applications* require more than just the *on-chain* transaction validation code. The majority of the code is typically *off-chain*, residing on a user's client machine and operating in the context of the user's *cryptographic wallet* —an application that facilitates the management of crypto-currencies and the creation of new transactions for submission to the blockchain. In systems such as Ethereum [Wood 2014], on-chain and off-chain code is implemented in different programming languages (preventing code reuse) and connected via an ad hoc network protocol (preventing type checking across the network boundary).

Existing proposals favouring functional programming and a rigorous formal treatment, such as Simplicity [O'Connor 2017], are concerned with on-chain code only and completely ignore the additional complexity introduced by on-chain off-chain code composition.

Cardano is a third generation blockchain that solves the energy usage issue by moving to an energy efficient *Proof of Stake* protocol, namely *Ouroboros* [Kiayias et al. 2017]. It also uses the *functional ledger representation*—essentially a dataflow graph, called the *UTxO* ledger—on which Bitcoin is based, but which was abandoned in many later systems for an object-based representation centring around a notion of accounts that exchange messages.

The core thesis of this paper is that we can build a purely functional system on the basis of the UTxO ledger representation and that we can support both sophisticated (on-chain) validation rules and off-chain code by way of a single, existing functional language. We illustrate this for Haskell, including its *Template Haskell* [Sheard and Jones 2002] template metaprogramming facility, but other languages could be used as well. More precisely, we make the following contributions:

- We describe a modest extension to UTxO (which we call *Extended UTxO*) which gives validation code greater context awareness and enables one to thread explicit state through a sequence of transactions (Section 3).
- We show how to use template metaprogramming to embed on-chain custom transaction validation scripts into off-chain contract logic code, facilitating code reuse and type checking between on-chain and off-chain code (Section 4).
- We outline a minimalistic and purely deterministic representation of on-chain validation code in the form of a variant of System F^{μ}_{ω} whose semantics are amenable to fully formal description (Section 5).
- We show that the resulting system is a suitable basis for embedded domain-specific languages, such as *Marlowe* [Lamela Seijas and Thompson 2018], a blockchain variant of [Peyton Jones et al. 2000]'s DSL for financial contracts (Section 6).

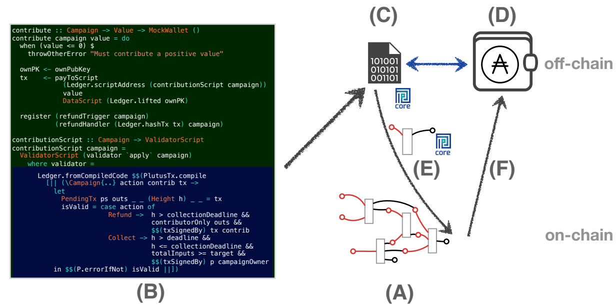

Figure 1. Plutus Architecture

We discuss related work in Section 7 and provide an overview over the Plutus programming model in the following section. All of Plutus is open source [Plutus Team 2019a] and we provide a Web environment, *Plutus Playgrounds,* to get started with writing Plutus contracts [Plutus Team 2019b].

2 The Plutus Programming Model

For the purposes of this paper, we can regard a blockchain as a ledger consisting of a sequence of transactions that move cryptocurrency around. In the *UTxO* model, popularised by Bitcoin, each transaction consists of a set of *inputs* (consuming currency) and a set of *outputs* (supplying currency). Inputs and outputs of the transactions in a ledger are connected to form a directed acyclic graph rooted in the *genesis block* of the blockchain. At any point, the dangling (*unspent*) outputs at the fringe of the graph form a set called the *unspent transaction outputs* (*UTxO*). This UTxO set fully determines the current state of the chain, and in particular, the distribution of funds.

Ownership of funds is conveyed indirectly by having the ability to spend a particular (as of yet) unspent output. This requires possessing the private cryptographic key that matches the public key *locking* the output. More specifically, a transaction containing an input that spends a given output from the UTxO set will only be admitted if it is cryptographically signed with the private key matching the public key in that output. A transaction is only admitted to the chain (i.e., it is only *valid*) if its signatures permit all spending specified by its inputs and if the sum of the cryptocurrency values in the outputs is smaller than that in the inputs (the difference is called the *transaction fee* and is used to pay for the infrastructure provided by the blockchain operators).

2.1 Plutus architecture

Figure 1 contains a partial UTxO graph, labelled (A), where the inputs are in red and the outputs are in black. An input connecting to an output symbolises that a signature on the input's transaction matches the public key in the output. This is our representation of the blockchain. We will discuss the UTxO model and our novel *Extended UTxO* model in more detail in Section 3.

On the left-hand side of Figure 1 we have a Plutus contract, labelled (B), that contains an off-chain part (green background) and an on-chain part (blue background). Both on-chain and off-chain code are interleaved in a single Haskell program, with on-chain code embedded into off-chain code using Template Haskell, as explained in Section 4.

Our toolchain—effectively extending GHC using its plugin support [ghc 2019, Section 13.3]—compiles a Plutus contract into an off-chain executable (C) embedding the compiled on-chain code in our core language *Plutus Core*, which we elaborate on in Section 5. The off-chain code executes in the context of a crpytocurrency wallet (D) which holds the funds to pay for transactions and the cryptographic keys to sign the transactions. When off-chain contract code submits a new transaction (E) to the blockchain, it typically locks (some of) the outputs with some of its embedded Plutus Core on-chain code.

The validation, and hence inclusion, of these and other transactions into the chain triggers *blockchain events* (indicated by the arrow labelled (F)) which are observed by the wallet. The wallet forwards events that are relevant for its off-chain contract code to that code, which in turn leads to new transactions, and so on.

The initial execution of off-chain contract code is typically triggered by the user of the wallet by way of *contract endpoints* (i.e., toplevel functions) that are made accessible through wallet UI elements. Hence, we may regard off-chain contract code as a form of plugin for cryptocurrency wallets.

2.2 An example contract: crowdfunding

As an example of a blockchain contract, consider a simple crowdfunding scenario. One person, the *campaign owner*, proposes a project and invites other users of the blockchain, the *contributors*, to fund that project by each contributing a (typically small) fraction of the costs. Part of such a proposal is the *funding target*, the minimum amount of funds that need to be raised to be able to complete the proposed project. If the funding target has not been reached by a certain time, the *campaign deadline*, the project is not viable and it is crucial that the contributors are refunded. It is also possible that the campaign owner abandons the campaign and doesn't collect the funds, even if the funding target has been reached. To ensure that the contributors are also refunded in this case, the campaign is also parameterised by a *collection deadline*—i.e., the point in time by which the campaign owner has to collect the contributions. Let's have a look at how we can write such a contract with Plutus.

We start by bundling the contract parameters in the record type:

```
data Campaign
  = Campaign
    { fundingDeadline    :: Slot
            -- campaign ends at that point
    , target             :: Ada
            -- funding target for campaign success
    , collectionDeadline :: Slot
            -- funds need to be collected by that point
    , owner              :: PubKey
            -- crypto key needed to collect funds
    }
```

`Ada` is the cryptocurrency of the Cardano blockchain, `Slot` specifies a time frame indirectly via the length of the blockchain, and `PubKey` is a cryptographic public key (identifying the campaign owner, in this case).

For now, let's just look at the code for the *contract endpoint* that lets a blockchain user contribute to an existing campaign—we can regard a contract endpoint as simply an action defined in the contract that a blockchain user can execute through their cryptocurrency wallet. The code makes use of Plutus library functions qualified with module names `L` and `W`. They originate from the Plutus *ledger* and *wallet API*, respectively.

```
contribute :: MonadWallet m
           => Campaign -> Ada -> m ()
contribute campaign value = do
 unless value > 0 $
   throwOtherError "Needs positive value"
 ownPK <- ownPubKey   -- key for refunds to us
 let
   dataScript = DataScript (L.lifted ownPK)
   validator  = contributionValidator campaign
   valAddress = L.scriptAddress validator
   range      = W.interval 1 (deadline campaign)
                    -- funding interval
   -- generate and submit contribution transaction
 tx <- payToScript range validator
                (Ada.toValue value) dataScript
   -- callback when refund conditions are met
 register (refundTrigger campaign)
          (refundHandler (L.hashTx tx) campaign)
```

This contract endpoint takes a campaign specification and an amount of Ada that the user would like to commit to that campaign. On the basis of that, the endpoint does two things: (1) with `payToScript` it generates and submits a transaction to the blockchain that pays the stated amount into the campaign in such a manner that the campaign owner can only retrieve those funds if the campaign is successful, and (2) with `register` it registers an *event trigger* (essentially a conditional callback) monitoring the blockchain for whether or not the campaign proceeds successfully: if not, the callback will submit a transaction claiming a refund.

In terms of the architecture in Figure 1, `contribute` is part of the (green) off-chain code and an invocation of the contract endpoint `contribute` originates in the wallet (D) and executes in the off-chain code (C). It submits a transaction (E) to the blockchain (A).

Two crucial components of the transaction's single output (the black outgoing edge) are the output's *validator script* (`validator`) and *data script* (`dataScript`). The validator script is on-chain code that guards the output in such a manner that it can only be used under the conditions stipulated in the contract. In the crowdfunding contract, the campaign owner can use it only (a) in case the campaign is successful and (b) only between the funding deadline and the collection deadline. Moreover, a contributor can only use the output holding their own contribution and only if the campaign fails. The data script is contract state information that may be used by the validator to establish context. In the crowdfunding campaign, it holds a public key `ownPK` of the wallet belonging to the contributor of that particular commitment. This enables us to guarantee that refunds go to the right person.

Both the validator and data script are executable on-chain code represented by expressions in Plutus Core,[1] the System *F*-based core language that we discuss in more detail in Section 5. There is a crucial difference between the two, though: a transaction only contains the cryptographic hash of the validator script (`valAddress` in the definition of `contribute`), but it is accompanied by the full value of the `dataScript`. We will discuss the reasons for this difference in detail in Section 3. However, it is worthwhile pointing out that determining the validator script hash only after the `contributionValidator` has been applied to the specific campaign in question provides us with a campaign-specific script address, which serves as a unique, unforgeable identifier for the contract.[2]

The validator script for crowdfunding contributions looks like this:

```
data CampaignAction = Collect | Refund

contributionValidator :: Campaign -> ValidatorScript
contributionValidator campaign =
 ValidatorScript
   (L.applyScript validator (L.lifted campaign))
 where
  validator = $$(L.compileScript [||
      — here begins the on-chain code
    λCampaign{..} (contributor :: PubKey)
    (action :: CampaignAction) (tx :: PendingTx) ->
      let
        PendingTx inputs _ _ _ slots _ _ = tx
          — transaction information
      in
      case action of
        Refund ->    — validate a refund action
          (L.from collectionDeadline)
            `L.contains` slots
          && tx `V.txSignedBy` contributor

        Collect ->  — validate fund collection
          let campaignTotal
                = sum [ Ada.fromValue value
                  | PendingTxIn _ _ value <- inputs]
              collectionRange
                = deadline
                  `L.interval` collectionDeadline
          in
             collectionRange `L.contains` slots
          && campaignTotal >= campaignTarget
          && tx `V.txSignedBy` campaignOwner
    ||])
```

The on-chain validation logic is defined by `validator` and wrapped in a typed Template Haskell quotation in `[|| .. ||]` brackets, which is fed to a function `L.compileScript` in a Template Haskell splice. We will discuss the details of our use of Template Haskell in Section 4. For the moment, it suffices to say that we use a combination of Template Haskell and GHC plugins to employ GHC's frontend and desugarer to provide us with GHC Core [Sulzmann et al. 2007] for the quoted on-chain code, which we translate to Plutus Core with our own custom compiler.

The on-chain code is a lambda abstraction that gets (1) the campaign parameters, (2) the contributor's public key, (3) the kind of `CampaignAction` to validate, and (4) the transaction that wants to make use of the funds contributed to the campaign (i.e., this is the transaction that we need to validate for conformance to the contract). After extracting the set of `inputs` and the range of `slots` (in which this transaction can be validated), validation is a matter of distinguishing between the two situations in which spending from a campaign commitment is permitted. In case of a `Refund`, we need to have a transaction whose slot range (`slots`) is entirely after the campaign `collectionDeadline` and the transaction needs to be signed by whoever committed the contribution that we are about to spend. In the second case, where the campaign owner wants to `Collect` the contributions, we must ensure that (1) the transaction slot range is entirely in between the campaign `deadline` and the campaign `collectionDeadline`, (b) the total campaign funds, `campaignTotal`, at least matches the `campaignTarget`, and (c) the transaction is signed by the `campaignOwner`.

As in this example, validator scripts are always predicates that check the conformance of a transaction with the rules

[1] This is purely functional code, so we don't distinguish between code and data.

[2] It is effectively unforgeable due to the use of a collision-resistant, cryptographically strong hashing function—Cardano specifically uses two rounds of SHA256 on a serialised version of the script.

of the contract under contractual parameters, such as the campaign parameters in our example, and contract state specific to the transaction output that the validator guards, such as the contributor's identity here (by proxy of their public key). In other words, on-chain code—in Plutus—does not compute anything or update any blockchain or other state: instead, it is a pure, side-effect free predicate. We argue that this is very powerful. It simplifies reasoning about on-chain code, both concerning functional correctness and resource consumption. This directly addresses two core problems that have plagued Etherum since its inception and led to many vulnerabilities and exploits.

Despite the benefits for reasoning about validator semantics, restricting these scripts to be pure predicates may also seem limiting. However, we recover the seemingly lost expressiveness by combing the on-chain with off-chain code in an extended UTxO ledger model that we detail in the following section.

3 The Extended UTXO Ledger Model

While Bitcoin introduced the graph-based ledger model commonly called a UTxO (unspent transaction output) ledger [Narayanan et al. 2016, Chapter 3], it only provided very limited capabilities for user-defined computation [Bit 2018; Bartoletti and Zunino 2018]. The limitations are twofold:

1. The BitCoin Script language constrains programs to be of a limited size and provides barely any control structures (essentially only conditional statements). The primitive operations that can be used in BitCoin Script are also very limited (for example, the division operation was originally included but was subsequently disabled).
2. The computational context available to a BitCoin Script program is very constrained. For example, it cannot even inspect the transaction that is currently being validated; it does have access to the hash of the transaction, though.

We address the first limitation in Section 5, where we discuss our on-chain code representation, Plutus Core. We address the second by defining an Extended UTxO model which provides on-chain scripts with sufficient context to pass contract state between transactions and to impose invariants (such as contract conditions and obligations) that hold across entire chains of transactions.

3.1 Transactions

Before we dive into scripts, we start by looking at the detailed structure of the transactions that make up the blockchain ledger. The fundamental transaction datatype is

```
data Tx = Tx {
    txInputs     :: Set.Set TxIn,
      — The inputs to this transaction
    txOutputs    :: [TxOut],
      — The outputs of this transaction
    txForge      :: Ada,
      — Currency forged by this transaction
    txFee        :: Ada,
      — The fee for this transaction
    txValidRange :: (Slot, Slot),
      — The validity interval for this transaction
    txSignatures :: Map PubKey Signature
      — Signatures of this transaction
}
```

The inputs and outputs are the connections to preceding and succeeding transactions as already discussed. Forged currency is new currency introduced into the ledger, whereas the fee is the portion of the overall transaction value that is payed to the *slot leader* that integrates the transaction into a block on the blockchain. One new block of the blockchain (containing many transactions) is created by the current slot leader in every `slot`, a fixed time interval defined by the Ouroboros proof-of-stake blockchain protocol. The current slot count provides a notion of time passed since the inception of the blockchain. The validity interval determines the slot range in which the current transaction can be validated: outside this interval the transaction is invalid.

Each transaction is associated with a unique identifier of type`TxID`, which is simply the cryptographic hash of a serialised representation of the transaction without the signatures. The signatures, in fact, sign the transaction hash and not the entire transaction itself.

The outputs of a transaction, `txOutputs`, are a list of

```
data TxOut = TxOut {
  txOutAddress :: TxID,
    — ID of the payment target
  txOutValue   :: Ada,
    — Value of output
  txOutType    :: TxOutType
    — What sort of output is it?
}
```

Outputs always pay a fixed value of a cryptocurrency to an address. In the simplest case, this address identifies a cryptographic key pair contained in a cryptographic wallet. This is generally called a *pubkey payment* and used for direct payments between two users of a blockchain. This case is covered by the first variant of the `TxOutType`:

```
data TxOutType
  = PayToPubKey PubKey       — pubkey payment
  | PayToScript DataScript — script payment
```

We shall discuss the second variant in the next subsection.

Outputs of one transaction get consumed by the inputs of subsequent transactions. To this end, each input specification `TxIn` contains an output reference of type `TxOutRef`:

```
data TxIn = TxIn {
    txInRef  :: TxOutRef,
    txInType :: TxInType
}
```

```
data TxOutRef = TxOutRef {
  id  :: TxID,
  — ID of previous transaction
  index :: Int
  — Index into the referenced transaction's outputs
  }
```

An output reference of type `TxOutRef` uniquely identifies an output within the existing ledger by a combination of the transaction identifier containing the output and an index into the list of outputs of that transaction. Moreover, transaction inputs also contain a component `txInType`, which needs to line up with the `TxOutType` of the consumed output.

```
data TxInType
  = ConsumePublicKeyAddress PubKey
  | ConsumeScriptAddress ValidatorScript
                         RedeemerScript
```

When a transaction input `txIn :: TxIn` refers in its `txInRef` to a transaction output `txOut :: TxOut` by specifying the identifier `id` of that output's transaction and the `index` of the `txOut` in `id`'s list of outputs, we say that `txIn` *spends* `txOut`. Given the a blockchain as a list of transactions, we can determine the set of all outputs which appear in a transaction but are not spent by an input of any other transaction. This set of outputs is called the *unspent transaction output* (*UTxO*) *set.* It determines all the value (funds) that can still be spent and is thus the primary data structure representing the current state of a UTxO ledger.

3.2 Validation

A crucial aspect of adding new transactions to an existing chain is *transaction validation.* The purpose of transaction validation for pubkey payments is to ensure the monetary integrity of the cryptocurrency processed in those transactions. The central validation conditions are the following:

1. Each output of every transaction may be spent at most once (by an input of another transaction). This is often called the *no-double-spend rule.*
2. Each input of every transaction must contain a transaction identifier `id` (in its `txInRef`) that refers to a transaction that actually exists in the chain. Moreover, the output `index` associated with the `id` in the `TxOutRef` value must exist in the referenced transaction `id`.
3. If a transaction `id` contains an input that spends an output `txOut`, then the transaction must contain a signature in its `txSignatures` field that was created with the private key matching the public key in `txOut`'s `txOutType`. In other words, owning the private key matching the public key of an output amounts to owning the value `txOutValue` in that output, as it confers the ability to spend it.
4. For any transaction, the sum of the values consumed (from other transaction's outputs) by all its inputs plus `txForge` must be equal the sum of all values produced by all of the transaction's outputs (i.e., sum of all `txOutValue` fields) plus the transaction fee `txFee`. This essentially means that we neither lose nor spuriously create value.

For more details and a precise mathematical specification of the standard UTxO ledger rules, see [Zahnentferner 2018].

3.3 Transactions with scripts

Pubkey payment outputs are sufficient for a simple payment system. If we want to go beyond that, we need a more sophisticated decision procedure to decide whether a given input is allowed to spend the output it refers to — i.e., we need to make Condition (3) of the enumeration in the previous subsection more general. The general idea here is to replace the combination of public key (the lock) and transaction signature (the key) with a general computation. Instead of the public key we have a validator script `validator`, and instead of the transaction signature we have a redeemer script `redeemer :: Redeemer`, with the assumption that `validator :: Redeemer -> Bool`. To validate a connection, we simply evaluate `validator redeemer` and require that it yields `True`.

This is exactly the situation with Bitcoin, where both the validator and the redeemer script are implemented in BitCoin Script (a simple stack-based language, whose most complex control structure is a conditional); the redeemer script is essentially all that an output's validator knows about the transaction (and the input) that is attempting to spend it.

In the Extended UTxO model we extend the context information considerably. The type of the validator is now

```
validator :: DataScript -> Redeemer -> PendingTx
          -> Bool
```

The `DataScript` is part of the same output as the validator. If we look at the definition of `TxOutType` again, we see that the `PayToScript` variant supplies exactly that data script.

But where is the validator script itself? It is actually not part of the transaction output. All that is included in the output is the hash of the validator script; to be precise, the hash of the validator script is the address, `txOutAddress`, of a pay-to-script output. As the collision-resistant cryptographic hash of a script (for all practical purposes) uniquely identifies that script, it is sufficient to fully determine the required validation computation.[3]

Now, if we look at the definition of `TxInType` again, we see that its `ConsumeScriptAddress` variant provides the missing validator script together with the input's redeemer. It is important to realise that the person who submits the spending transaction cannot cheat at this point. If they provide the wrong validator script, its hash won't match the output address and the transaction will be deemed invalid.

[3]The actual validator script is not required until validation time, so in order to reduce on-chain storage requirements it may be stored off-chain until it is needed, or an already-existing validator may be re-used.

The third argument, `PendingTx`, to the extended validator signature contains the entire transaction that is currently being validated. We know this setup already: the `validator` definition in the where clause of `contributionValidator` in Section 2, after partial application to the `Campaign` parameters, has the same structure (with contract-specific types for `DataScript` and `Redeemer`).

This extension of the context available to the validator greatly boosts the expressiveness of the contract system, to a level where we conjecture that it is comparable to that of Ethereum. We already go beyond Bitcoin Script with simple examples like our crowdfunding code, and leave it far behind with complex examples such as the Marlowe financial contracts described in Section 6.

4 Staged programming

Just like web applications, *decentralised applications* (*dapps*) built on blockchains comprise two separate components which are deployed in separate execution environments:

1. *on-chain* code stored on the blockchain and executed during the inclusion of new transactions into new blocks that are being added to the chain (similar to the server component of a web application), and
2. *off-chain* code typically deployed through a website and executed on the client machine of a blockchain user with access to the user's cryptographic wallet, much like the client portion of a web application running in a user's web browser; in fact, dapp off-chain code does often execute in a web browser as well.

Why is this decomposition necessary? The on-chain code contains the dapp's contractual components. It needs to enforce that only transactions that meet the contractual obligations are successfully validated and added to the chain. In other words, the integrity of a smart contract 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 (the servers adding new blocks to the chain in the proof-of-stake Ouroboros protocol, corresponding to the *miners* of a proof-of-work protocol as employed in Bitcoin) need to execute on-chain code—specifically, the validation scripts from Section 3—to guarantee that only transactions that abide by all relevant contractual obligations are accepted into the chain.

Conversely, the off-chain code, which submits new transaction to slot leaders for validation and inclusion into the chain, necessarily needs to run in close association with a contract user's cryptocurrency wallet. After all, each transaction needs to be paid for by inclusion of a small transaction fee, and a cryptographic wallet is the only place where the necessary cryptographic 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 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, which has led to the proposal of *tierless* web programming [Cooper et al. 2007]. 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 overcome these problems by using Haskell for both on-chain and off-chain code. This enables us to build on the existing, vibrant Haskell ecosystem and to seamlessly share datatypes and code between the two. As an added bonus, the Haskell typechecker helps us to avoid mistakes where the two connect.

The purely functional nature of Haskell helps us to keep the on-chain and off-chain code separate, but we still need a language mechanism to distinguish between on-chain and off-chain code. Given that off-chain code conceptually embeds on-chain code, as the former submits the latter in the form of scripts accompanying transactions, one option would be to use an *embedded language*, similarly to how the Accelerate library [Chakravarty et al. 2011] embeds array code to custom-compile to off-load to accelerators, such as GPUs. Unfortunately, embedded languages tend to lead to complex types (again, Accelerate is a good example) and one of our design goals was to make Plutus easy to use for developers who are new to Haskell. Secondly, embedded languages favour runtime compilation of the embedded language (as, once more, becomes obvious when looking at Accelerate).

In summary, we require a two-level language, but we want the embedded language to be compiled at host language compile time. This is exactly what compile-time metaprogramming, as realised by Template Haskell [Sheard and Jones 2002], provides.

4.1 Template Haskell for embedded code

While Template Haskell fits our requirement of compile-time metaprogramming, it doesn't directly support our need to generate code in our own intermediate language, Plutus Core, for storage and execution on the blockchain. Let's extract the on-chain validator code from the `contributionValidator` function of the crowdfunding contract and have a look at it:

```
val = [||
  λCampaign{..} (contributor :: PubKey)
    (action :: CampaignAction) (tx :: PendingTx) ->
      let
       PendingTx inputs _ _ _ slots _ _ = tx
       in
      case action of
```

```
      Refund  -> . . .
      Collect -> . . .
  ||])
```

The type of `val` is

```
Q (TExp (Campaign -> PubKey
          -> CampaignAction -> PendingTx -> Bool))
```

If we were to splice the quoted Haskell code bound to `val`, it would be compiled through GHC's standard pipeline and would end up being inlined into the embedding off-chain code. Alternatively, we could use Template Haskell's abstract syntax for Haskell (defined by `TExp`) and write a custom compiler to Plutus Core. However, the latter seems wasteful. Plutus Core is lower-level than, but strongly related to GHC's Core intermediate language [Sulzmann et al. 2007] (both are variants of System *F*). GHC itself already includes all the machinery to typecheck and desugar Haskell to GHC Core. It seems inadvisable to duplicate that functionality.

In principle, we could use the GHC API (i.e., GHC as a library from within Template Haskell). However, this is awkward for two pragmatic reasons: (1) the abstract Haskell syntax provided by Template Haskell is frustratingly different from that used inside GHC, and (2) Template Haskell doesn't distinguish between package dependencies of the meta program and the object program. In other words, if we use the GHC API during compiling a Plutus program in Template Haskell, we also need to ship GHC as a library with the Plutus off-chain code to every user wallet (even though it is never used at runtime). This also seems very wasteful.

Luckily, there is an alternative: GHC plugins [ghc 2019, section 13.3].

4.2 Plugins for custom compilation

GHC core-to-core plugins enable us to inject our own custom Plutus Core compiler code into the GHC pipeline. Our custom compiler,

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 with 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? (GHC Core is a typed intermediate language; hence, any code transformation needs to be type-preserving.) 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 [Domínguez and Boespflug 2017]. The idea is to wrap the quoted AST (e.g., `validator` above) into a splice of a Template Haskell function that inserts a marker around that AST fragment; specifically, we use `$$(L.compileScript val)`, where

```
compileScript :: Q (TExp a)
              -> Q (TExp (CompiledCode a))
```

Now, `compileScript`—despite its name—does not actually compile the AST of the quoted program fragment. Instead, it inserts a marker, `plc`, that is picked up by our custom Plutus Core compiler injected with the plugin. Specifically, we have

```
compileScript e = [|| plc $(e) ||]
```

with `plc :: a -> CompiledCode a`.[4] Now, `CompiledCode` is the type of our serialised Plutus Core representation, so the types line up, too. Voilà!

4.3 Compiling GHC Core to Plutus Core

Both GHC Core and Plutus Core are extensions of System *F*, the polymorphic lambda calculus. GHC Core is much the more generous extension. It adds mutually-recursive binding groups, algebraic data types, case expressions, coercions, and more. In contrast, Plutus Core, for reasons that we explain in Section 5, stays much closer to the mathematical calculus.

Now, folklore has it that all the fancy constructs of GHC Core can be desugared into the pure calculus as long as it includes a simple facility for recursion, such as a fixed-point combinator. In practice, it appears as if, so far, nobody has worked out all the details required to handle full Haskell algebraic data types. As this is an interesting topic in itself, we cover it in a companion paper [Kireev et al. 2019].

4.3.1 Lifting values at runtime

There is one last thing! We are going to great lengths to compile on-chain validator scripts at off-chain code compile time. However, the data scripts and often also the redeemer scripts are values (not programs) determined at compile time. For example, in `contribute` (Section 2), we use `ownPubKey` to obtain a public key `ownPK` from the wallet at off-chain code runtime, in order to include it into the data script of the transaction to be submitted. Data scripts are also included in the form of Plutus Core; hence we need to compile the value of `ownPK`. This is quite simple as public keys are simple byte strings. However, in other situations we need to translate more complex Haskell data structures to Plutus Core. Including the entire Plutus compiler toolchain into every off-chain program is not practical, so we need a more lightweight solution.

We take inspiration from how Template Haskell injects runtime values into quoted code. It has a `Lift` type class with a `lift` method that constructs the Template Haskell AST representing the argument passed to `lift`. We similarly reuse part of the Plutus Core compiler along with typeclasses to generate instances of the following classes (Term and

[4] In reality, the definition is slightly more involved to help us generate good error messages in the plugin compiler.

Type are the datatypes for Plutus Core 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 ())
```

5 Plutus Core

A key function of Plutus is to generate on-chain validator scripts whose execution during transaction validation ensure contract integrity and security properties. Our compiled representation for on-chain scripts is Plutus Core. Once submitted as part of a blockchain transaction, scripts are immutable. One must have absolute certainty as to what the code will do, so a complete specification of Plutus Core is essential.

The stakes for a smart contract can be high; billions of dollars are currently invested in smart contracts on Ethereum [Wood 2014]. Changing deployed contract scripts is only possible if a majority of block producing nodes agree to an undesirable update called a *hard fork*. As a result, it is important to design a language without hidden flaws, and that can remain stable for a long time.

5.1 A minimal core language

What language should serve as Plutus Core? We need a small, purely functional intermediate language to simplify the process of precisely specifying its semantics and mechanising its meta-theory. We are not the first to decide that System F [Girard 1972] is a good basis for a typed intermediate language (a choice, for example, also made by GHC). However, we deviate from the standard choices in two important ways: (1) we are basing our design on System F_ω and (b) we don't have explicit datatypes and case expressions in Plutus Core.

System F_ω directly supports parameterised types such as *List A*, where *List* has the higher order kind, $\mathit{Type} \to \mathit{Type}$. Explicit datatypes and case expressions are usually included in intermediate languages as they facilitate code optimisations and efficient machine code generation. However, they are typically the language construct with the most complex semantics. Plutus Core is never compiled to machine code. It gets interpreted in a sandbox during validation of transactions. Moreover, a large part of the computational costs of transaction validation is in the crypotographic operations; in comparison, the computational overhead of desugaring datatypes seems minor. The alternative to including explicit datatypes and case expressions is simulate them using Church encoding or Scott encoding.

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

5.2 Isorecursive vs. equirecursive vs. ifix

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 [Pierce 2002, chapter 21]. In the equirecursive approach one views a recursive type as an abbreviation for an infinite tree, and considers $\mu\alpha.\, A[\alpha]$ and $A[\mu\alpha.\, A[\alpha]]$ to be the same type. In the isorecursive approach, one considers $\mu\alpha.\, A[\alpha]$ to be a type in its own right, and introduces two term forms, *fold* to convert $A[\mu\alpha.\, A[\alpha]]$ to $\mu\alpha.\, A[\alpha]$, and *unfold* to convert the other way.

Above, we've assumed that α has kind Type, but at higher kinds things become more complicated. Strictly speaking we should write $\mu\alpha : K.\, A[\alpha]$, where α has kind K, and kinds are given by the grammar: $J, K ::= \mathit{Type} \mid J \to K$. While equirecursive types in System F are known to be decidable, it is not known whether or not equirecursion is decidable at higher kinds [Cai et al. 2016]. Accordingly, we picked the more conservative design: isorecursive types.

Here, also, at higher kinds there is a twist. Terms must have a type, so one cannot have terms that directly correspond to fold and unfold at higher kind. The trick is to realise that every kind K must have the form $K_1 \to \cdots \to K_n \to \mathit{Type}$.

Hence, a term involving recursion at higher-kind must have the type $M : (\mu\alpha : K.\, A[\alpha])\ A_1\ \cdots\ A_n$ where $A_1 : K_1, \ldots, A_n : K_n$. Unfolding then yields the term

$$\mathit{unfold}_{A_1,\ldots,A_n}\ M : A[\mu\alpha : K.\, A[\alpha]]\ A_1\ \cdots\ A_n$$

Similarly for *fold*. No way is known to infer $A_1, \ldots, A_n$, so they must be explicitly present in the *fold* and *unfold* terms. This is called the *spine* formulation, and is found in the PhD thesis of a recent Milner award winner [Dreyer 2005].

A different formulation turns out to have equivalent power while being slightly less messy. We replace $\mu\alpha : K.\, A[\alpha]$ by

$$\mathit{ifix}_K : ((K \to \mathit{Type}) \to (K \to \mathit{Type})) \to (K \to \mathit{Type})$$

Given a term of type $M : (\mathit{ifix}_K\ A)\ B$, where $A : K \to K$ and $B : K$, we then have $\mathit{unfold}_{A,B}\ M : A\ (\mathit{ifix}_K\ A)\ B$, which avoids the need to list a whole spine of constructs. We settled on this construct, after [Brown and Palsberg 2017], who use the same syntax for isorecursive fixed points, but do not support fixed points at arbitrary kinds.

When we first hit on modelling Plutus Core after System F_ω we were pleased at being able to base our design on such a canonical approach. It was a disappointment to discover the complications above. While we can still rely on standard solutions, they are not quite so widely known in the theory community as we might have hoped.

5.3 Recursion on values

On the other hand, there was one pleasant surprise along the way. Our original design included both fixpoints at the type level, as above, and a fixpoint at the value level to define recursive functions. We were pleased to discover that the latter was redundant. It is well known that one can define recursion over values in the untyped lambda calculus, and it is well known that one can model the untyped lambda calculus with the recursive type $\mu\alpha.\, \alpha \to \alpha$.

Hence, it should be straightforward to define recursion at the value level in terms of recursion at the type level, which is confirmed by [Harper 2012, chapter 20.3]. Accordingly, we deleted recursion on values from our core calculus.

5.4 Mutual recursion

The construct $\mu\alpha.\, A[\alpha]$ only supports a singly-recursive type. Is that sufficient, or do we need to add a construct to support mutually-recursive types?

One possibility is to apply Bekić's Theorem [see Winskel 1993]. Consider mutually recursive types defined by the following two equations. $\alpha = A[\alpha, \beta]; \beta = B[\alpha, \beta]$. These can be solved by setting: $\alpha = \mu\alpha.\, A[\alpha, \mu\beta.\, B[\alpha, \beta]]; \beta = \mu\beta.\, B[\mu\alpha.\, A[\alpha, \beta], \beta]$. The size of such terms grows rapidly. Ten mutually recursive types, where each type depends upon the other nine, can be written as equations in 180 symbols. Expansion with Bekić's theorem is explosive: it requires 190 million symbols. Fortunately, it is extraordinarily rare to find such a type.

But then we discovered a better approach than Bekić's Theorem, which exploited higher-order kinds to encode mutually recursive families of types with only a constant factor increase compared to a set of mutually recursive equations. That approach is described in [Kireev et al. 2019].

5.5 Formal model in Agda

One pleasant development is that the cryptocurrency community respects the value of formal methods. The proposal for a new scripting language for Bitcoin, called *Simplicity*, was accompanied by a complete formal description in Coq [O'Connor 2017], and the smart contract language Michelson has also been formalised in Coq [mic 2018]. Accordingly, we developed the meta theory of Plutus Core in Agda: the development is described in the companion paper [Anonymous 2019].

6 Marlowe

Marlowe [Lamela Seijas and Thompson 2018] is a domain-specific language targeted at the execution of financial contracts in the style of [Peyton Jones et al. 2000] on a blockchain.

To support execution on blockchain, Marlowe adds notions of *commitments* and *timeouts* to the model of [Peyton Jones et al. 2000]. A commitment calls for a participant to commit currency to a contract, e.g. to ensure that a payment will not fail. Timeouts are used to ensure that contracts make timely progress during execution. For example,

```
Pay i alice bob val t k
```

describes a contract that enables a payment of `val` from `alice` to `bob` (identified by `i` and timing out at time `t`). Once the payment is successfully claimed by `bob`, or if the timeout happens before `bob` makes a claim, the contract continues as `k`, itself a Marlowe contract.

The behaviour of Marlowe contracts is encapsulated in a small-step semantics. At each step this takes any available inputs: commitments, values of oracles, payment claims; the contract state, that keeps track of commitments; and the contract itself. It will return any actions generated, together with the updated state and remaining contract.

We implement Marlowe contracts with Plutus and its Extended UTxO model by effectively encoding the transition step function of the Marlowe small-step semantics as Plutus on-chain code. As described in the previous sections, this is compiled into a Plutus Core validator script that ensures that funds locked by the contract can only be spent in accordance with the Marlowe semantics: we call this the *Marlowe validator*. The off-chain component of a Marlowe contract chains multiple transactions together, effectively implementing the transitive closure over the transition step function. The remaining contract and its state are encoded in the data scripts accompanying the Marlowe validator. Finally, actions and inputs (i.e., *choices* and *oracle values*) of a Marlowe contract are passed as redeemer scripts. Overall, each step in Marlowe contract execution encoded in Plutus is a transaction which spends an output locked by the Marlowe validator by providing a valid input in a redeemer script, and produces a transaction output with a Marlowe contract as a continuation (the remaining contract).

6.1 Design space

The implementation outlined above effectively implements a Marlowe interpreter (in the form of its small-step semantics) in Plutus. An alternative would have been to implement a Marlowe to Plutus compiler that, given a Marlowe contract, generates contract-specific Plutus code, effectively specialising the Marlowe validator to the next language construct in the remaining Marlowe contract (contained in the accompanying data script). The interpreter approach has a number of advantages:

- It is simple: we implement a single Plutus script that can be used for all Marlowe contracts, thus making it easier to implement, review, and test what we have done.
- It is close to the semantics of Marlowe described in [Lamela Seijas and Thompson 2018], making it easier to audit.
- It means that the same implementation can be used for both on- and off-chain (wallet) execution of Marlowe code.
- It allows client-side contract evaluation, where we reuse the same code to do contract execution emulation (e.g., in an IDE or in a web-based development environment for Marlowe).
- Having a single interpreter for all (or a particular group of) Marlowe contracts allows one to monitor the blockchain for these kinds of contract, if desired.

Finally, as we retain the remaining contract in data scripts accompanying the Marlowe validator, we make it accessible to everyone, simplifying contract reflection and introspection.

6.2 Contract lifecycle on extended UTxO model

Given our implementation of Marlowe by way of the Marlowe validator implementing the Marlowe operational semantics, we can divide the execution of a Marlowe contract into three phases: *creation*, *execution*, and *completion*.

6.2.1 Creation

Marlowe contract creation is realised as a *creation transaction* with at least one script output locked by the Marlowe validator and with a given Marlowe contract in the data script; this output must contain a non-zero amount of money, a *contract deposit*, which can be spent during the completion phase. Note that we do not place any restriction on the transaction inputs, which could use any other transaction outputs, including ones locked by scripts. As part of the creation transaction, we can initialise a contract with a particular state containing a number of commitments, as shown in Figure 2.

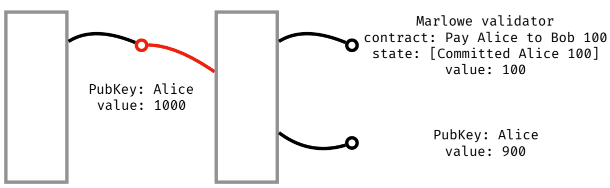

Figure 2. Marlowe contract initialisation by committing money

6.2.2 Execution

A Marlowe contract executes by way of the stepwise submission of *execution transactions* by the parties involved. These transactions form a chain where the remaining contract and the current contract state are captured in the data script of the *continuation output* of the associated execution transaction; the continuation output is always the one that is also locked by the Marlowe validator. Moreover, contract actions and inputs, the choices and oracle values, are represented as redeemer scripts on inputs spending from Marlowe validators.

We illustrate this chain in Figure 3. The black outputs in the chain are all locked by the Marlowe validator. They are spent by connecting redeemer scripts (red lines) that represent the actions and inputs. The Marlowe validator, encoding the Marlowe operational semantics, first validates the current contract and state, given in its accompanying data script. That is, it checks that the contract correctly uses identifiers, and holds at least what it should, namely the deposit and the outstanding commitments.

The validator then evaluates the continuation contract and its state, using the `eval` function, i.e., the transition step

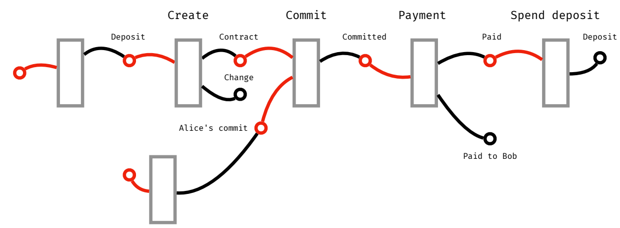

Figure 3. Simple Marlowe contract phases

function defined in [Lamela Seijas and Thompson 2018] with the following signature:

```
eval :: Input
     -> Slot -> Ada -> Ada -> State -> Contract
     -> (State, Contract, Bool)
```

Here, `Input` is a combination of contract participant *actions* (`Commit`, `Payment`, `Redeem`), oracle values, and choices made by the participants. The two `Ada` parameters are the current contract value and the result contract value. So, for example, if the contract is to perform a 20 Ada `Payment` and the input current amount is 100 Ada, then the result value will be 80 Ada. The `Contract` and `State` values are the current contract and its State, taken from the data script.

On the basis of these arguments, the `eval` function, taking into account the transaction's slot range, checks that all inputs are within defined bounds and that payments are within committed bounds. In case of a valid input, it returns the new `State` and `Contract` and the Boolean `True`; otherwise, it returns the current `State` and `Contract`, unchanged, together with the Boolean `False`.

It is important to keep in mind that on-chain code cannot generate transaction outputs, but can only validate whatever values the off-chain code provides in a transaction. The values for every step during contract evaluation are created by off-chain code (or manually by a user) and submitted to the blockchain for validation as part of a transaction. In contrast to committed on-chain code, off-chain code can be arbitrarily manipulated by a user, and so cannot be trusted by other participants in the contract. Consequently, the on-chain validator must carefully check all values provided to it, including any `Contract` and `State` values. The only piece of information that the Marlowe validator can trust is the data script located in the same transaction output as itself (after all, that data script was provided by the same entity as the validator).

Take the following contract:

```
Commit id Alice 100 (Both (Pay Alice to Bob 30 Ada)
                          (Pay Alice to Charlie 70 Ada))
```

After Alice has made her commitment, the contract becomes

```
Both (Pay Alice to Bob 30 Ada)
     (Pay Alice to Charlie 70 Ada)
```

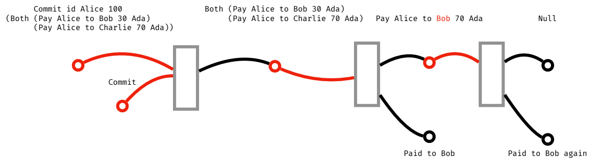

Figure 4. Malicious Marlowe contract execution

Bob can now issue a transaction with a `Payment` input in the redeemer script and a script output with 30 Ada less, protected by the Marlowe validator script, together with a data script containing the evaluated continuation contract

```
Pay Alice to Charlie 70 Ada
```

Charlie can then issue a similar transaction to receive the remaining 70 Ada.

6.2.3 Ensuring execution validity

Looking again at this example, suppose that Bob chooses, maliciously, to issue a transaction with the continuation `Pay Alice to Bob 70 Ada` to try to take all the money, as in Figure 4, much to the disappointment of Charlie. To avoid this, we must ensure that the continuation contract resulting from `eval` is equal to the one in the data script of its continuation transaction output.

Performing this equality check is tricky, though. From Section 3, we know that the transaction information passed to the validator only contains the cryptographic hashes of the data scripts of each transaction output. We might hope to be able to compute the hash of the continuation contract and the new contract state in the on-chain validator itself. However, this is very fragile since many denotationally equal Plutus Core expressions have different serialised forms, and hence different hashes. (In other words, α-, β-, and η-conversion all preserve denotational equality, but they change the hash of the serialised terms.)

To work around this we require the input redeemer script and the output data script to be identical; more precisely, we require that they have the same hash, which we can easily check as both hashes are included in the transaction. At validation time, a validator gets the spending input's redeemer passed as an argument (the actual term, not just the hash), and hence we can use it to check for equality with the result of running `eval`.

The spending redeemer and the data script of the continuation output both have the same type: `(Input, MarloweData)` where (1) the `Input` contains contract actions (i.e., `Payment`, `Redeem`), `Choices`, and `Oracle Values`; (2) `MarloweData` contains the remaining `Contract` and its `State`; and (3) the `State` here is a set of `Commits` plus a set of `Choices` made.

To spend a transaction output locked by the Marlowe validator script, the off-chain code must provide a redeemer script which is a pair of an `Input` and the expected output of interpreting a Marlowe contract for that given `Input`; i.e., a `Contract/State` pair. The expected contract and state can be precisely evaluated beforehand off-chain using the same `eval` function as is contained the Marlowe validator.

To ensure that the off-chain code provides valid remaining `Contract` and `State` values, the Marlowe validator script will compare the evaluated contract and resulting state with those provided within the redeemer value, and will reject all transactions where those do not match.

To ensure that the remaining contract's data script has the same `Contract` and `State` values as those that the validator got in the redeemer script, we check that the data script hash is the same as that of the redeemer script.

6.2.4 Completion

When a contract evaluates to `Null` and all expired `Commits` are redeemed, the initial contract deposit can be spent, closing the contract.

7 Related Work

Smart contract systems. There is a large number of proposals for blockchain smart contract systems. Most are only described in informal "white papers" and have not been implemented. It is beyond the scope of this paper to review them all, so we focus on the most pertinent.

The two most widely used low-level smart contract languages are Bitcoin Script [Bit 2018] and the Ethereum Virtual Machine (EVM) bytecode [Wood 2014]. These are both comparatively *ad hoc* languages with semantics given by a stack machine. Bitcoin Script is very limited in expressiveness and restricted to constant-time programs with very limited control structures (essentially only branches). There is also no "official" high-level language compiling to Bitcoin Script, although several proposals have been put forward by a variety of groups. At the other extremity, the EVM supports a fully-fledged instruction set and admits Turing complete programs (whose execution is dynamically limited by a contract user's need to pay a cost proportional to the used resources); moreover, there is a de facto standard high-level language: the statically typed, object-oriented Solidity [Sol 2019].

Closer to our Plutus Core is the language Simplicity [O'Connor 2017], which is a combinator language together with an abstract machine giving its operational semantics. Like Plutus Core, Simplicity has been formalised in a proof assistant, and is designed to facilitate reasoning about the resource usage of programs. Unlike Plutus Core, Simplicity is not Turing complete. Further, while it is straightforward to adapt sophisticated functional programming techniques for use in Plutus Core due to its basis in System F_ω, the same techniques are not so readily usable with Simplicity.

The Tezos system introduces Michelson as a bytecode-based, low-level compiler target, which might be characterised as a statically typed crossover between Forth and

Lisp. Michelson has been formalised in Coq [mic 2018]. Several higher-level languages have been proposed, but none appear to be available at the time of writing.

All of the above languages cover only the on-chain component of distributed apps. The integrated treatment of on-chain and off-chain components in Plutus is, to the best of our knowledge, a unique feature among general-purpose contract languages. Moreover, instead of inventing yet another language, we stick to System F_ω and Haskell and inherit the rich amount of existing work around these.

Implementations of System F^{μ}_{ω}. Two recent implementations of System F_ω with recursive types are System $F^{\mu i}_{\omega}$ [Brown and Palsberg 2017] and System $F^{\mu *}_{\omega}$ [Cai et al. 2016]. $F^{\mu i}_{\omega}$ is more similar to Plutus Core, as it also uses isorecursive types. The primary difference is that the fixed point operator in Plutus Core is available at arbitrary kinds, while in $F^{\mu i}_{\omega}$ the fixed point operator is restricted to kind $(* \to *)$. Additionally, $F^{\mu i}_{\omega}$ extends F_ω with a type operator `Typecase` that allows syntactic inspection of types. In contrast, $F^{\mu *}_{\omega}$ uses equirecursive types, and the fixed point operator is restricted to kind $*$. Additionally, $F^{\mu *}_{\omega}$ supports algebraic datatypes through record and variant syntax.

References

2013–2018. Bitcoin Script reference guide. https://en.bitcoin.it/wiki/Script.

2016–2019. Solidity documentation. https://solidity.readthedocs.io/.

2018. Michelson in Coq. GitRepo. https://framagit.org/rafoo/michelson-coq

2019. GHC User's Guide. https://downloads.haskell.org/~ghc/8.6.3/docs/html/users_guide/index.html. Accessed: 2019-02-20.

Anonymous. 2019. System F in Agda, for fun and profit. Under submission.

Massimo Bartoletti and Roberto Zunino. 2018. BitML: A Calculus for Bitcoin Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, 83–100.

Matt Brown and Jens Palsberg. 2017. Typed Self-Evaluation via Intensional Type Functions. In *Proceedings of the 44th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. 415–428.

Yufei Cai, Paolo G. Giarusso, and Klaus Ostermann. 2016. System F-omega with Equirecursive Types for Datatype-Generic Programming. In *ACM SIGPLAN Symposium on Principles of Programming Languages*.

Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (FMCO'06)*. Springer-Verlag, Berlin, Heidelberg, 266–296.

Facundo Domínguez and Mathieu Boespflug. 2017. GHC compiler plugins in the wild: typing Java.

Derek Dreyer. 2005. *Understanding and Evolving the ML Module System*. PhD Thesis. Carnegie Mellon University, School of Computer Science.

Samuel Falkon. 2017. The Story of the DAO – Its History and Consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee. medium.com.

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État. Université Paris 7.

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

Alex Hern. 2018. Bitcoin's energy usage is huge – we can't afford to ignore it. https://www.theguardian.com/technology/2018/jan/17/bitcoin-electricity-usage-huge-climate-cryptocurrency. The Guardian.

Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Advances in Cryptology - CRYPTO 2017*. 357–388.

Roman Kireev, Chad Nester, Michael Peyton Jones, Philip Wadler, Vasilis Gkoumas, and Kenneth MacKenzie. 2019. Unraveling recursion: compiling an IR with recursion to System F. Under submission.

Pablo Lamela Seijas and Simon Thompson. 2018. Marlowe: Financial Contracts on Blockchain. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. ISoLA 2018. (LNCS)*, Vol. 11247.

Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press.

Russel O'Connor. 2017. Simplicity: A New Language for Blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*.

Simon Peyton Jones et al. 2000. Composing Contracts: An Adventure in Financial Engineering (Functional Pearl). In *ICFP*. ACM.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Plutus Team. 2019a. The Plutus language implementation and tools. https://github.com/input-output-hk/plutus.

Plutus Team. 2019b. Plutus Playground. https://testnet.iohkdev.io/plutus/.

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *2002 ACM SIGPLAN Workshop on Haskell*. ACM, 1–16.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *ACM SIGPLAN Int. Workshop on Types in Languages Design and Impl.*

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

Gavin Wood. 2014. Ethereum: A Secure Decentralized Generalised Transaction Ledger. (2014). https://gavwood.com/paper.pdf

Joachim Zahnentferner. 2018. Chimeric Ledgers: Translating and Unifying UTxO-based and Account-based Cryptocurrencies. *IACR Cryptology ePrint Archive* 2018 (2018), 262. http://eprint.iacr.org/2018/262