{p.lamela-seijas,s.j.thompson}@kent.ac.uk

Abstract. Blockchains allow the specification of contracts in the form of programs that guarantee their fulfilment. Nevertheless, errors in those programs can cause important, and often irretrievable, monetary loss. General-purpose languages provide a platform on which contracts can be built, but by their very generality they have the potential to exhibit behaviours of an unpredictable kind, and are also not easy to read or comprehend for general users.
An alternative solution is provided by domain-specific languages (DSLs), which are designed to express programs in a particular field. This paper explores the design of one DSL, Marlowe, targeted at the execution of financial contracts in the style of Peyton Jones *et al* on blockchains. We present an executable semantics of Marlowe in Haskell, an example of Marlowe in practice, and describe the Meadow tool that allows users to interact in-browser with simulations of Marlowe contracts.

Keywords: No keywords

Marlowe: financial contracts on blockchain*

Pablo Lamela Seijas[0000−0002−1730−1219]
and Simon Thompson[0000−0002−2350−301X]

School of Computing, University of Kent, Canterbury, UK

1 Introduction

This paper explores the design of a domain specific language, Marlowe,[1][2] targeted at the execution of financial contracts in the style of Peyton Jones, Eber and Seward [16] on blockchains. In doing this, we are required to refine the model of contracts in a number of ways in order to fit with a radically different context.

Consider the following example of an "escrow" contract so that we can explain the motivation more concretely. The aim of this contract, written in functional pseudocode in the style of [16] involves three participants: `alice`, `bob` and `carol`. `alice` is to pay an amount of money to `bob` on receipt of goods from her. `alice` pays the money into escrow controlled by `carol`.

There are two options for the money: if two out of the three participants agree to `pay` it to `bob`, that goes ahead; if, on the other hand, two of the participants opt to `refund` the money to `alice`, that is done instead.

The outer primitive `When` waits until the condition – its first argument – becomes true; in this case, the condition is that either two participants choose `refund` or two participants choose `pay`. The second argument of the `When` is itself another `Contract`, which is performed after the condition of the `When` has been met, and it makes the payment if two participants chose `pay`, otherwise it redeems previous money commitments.

```
(When (Or (two_chose alice bob carol refund)
          (two_chose alice bob carol pay))
      (Choice (two_chose alice bob carol pay)
              (Pay alice bob AvailableMoney)
              redeem_original))
```

We discuss this particular example in more detail in Marlowe in Section 3 below; but it already gives us an example of how traditional contracts are fundamentally different from contracts that are meant to be run on top of the blockchain. In the traditional model, enforcement of the contract is the responsibility of the legal system. If `alice` does not pay the money into escrow, or `carol` chooses to keep it for herself, then they can be sued for the money

* This work is part of the Cardano project and is supported by IOHK, https://iohk.io

[1] Named after Christopher Marlowe, the Elizabethan poet, dramatist and spy, who was born and educated in Canterbury, en.wikipedia.org/wiki/Christopher_Marlowe

[2] Marlowe is available from https://github.com/input-output-hk/scdsl

(and probably damages), thus providing both legal and financial incentives for compliance. On the other hand, in the decentralised blockchain model, where there is no central authority, the contract needs to be enforced *by design*.

This means that we must require participants to *commit* money to cover all possible expenditure *in advance of the contract executing*. In order to make sure that participants continue to engage with a contract, we ensure urgency by imposing *timeouts*: money is committed for a finite period only. We also impose a timeout when waiting for a participant to make a commitment to ensure that the contract does not become stuck even if one of the participants stops interacting with it.

We make the following contributions in this paper.

- Designing a DSL for financial contracts on blockchains: Marlowe.
- Defining an executable, small-step semantics of Marlowe in Haskell.
- Making Marlowe an embedded DSL in Haskell. This extends the expressibility of the language, as we can use all the facilities of Haskell in defining Marlowe contracts; we achieve this by defining Marlowe as a Haskell `data` type.
- Developing the Meadow tool that allows users to interact with and simulate the operation of Marlowe contracts and embedded Marlowe contracts.

Having established this model and its semantics we are able to do a number of other things, which we discuss in the paper. We can explore how the language will be implemented on an existing blockchain, such as Cardano, and our model has shown us that we will need to consider how the evolution of a contract interacts with the blockchain, with miners in the case of a 'proof of work'-based chain, and with users participating in contract execution. We can also perform analyses of Marlowe contracts, based on its formal semantics.

In designing a language like Marlowe we are constrained by the blockchain domain, but within that we do have a range of choices. For example, should we base the language on a system with accounts, like Ethereum, or a UTxO-based model as used by bitcoin? We examine this and other choices after introducing the language and examples of its use.

The paper begins in Section 2 by introducing the Marlowe model, including the assumptions made in designing it, the types of the principal functions and a description of the DSL as an algebraic type, constructor by constructor. Section 3 revisits the escrow example, and shows how it is described using a combination of Marlowe and Haskell constructs: that is, we use Marlowe as an embedded DSL. Section 4 introduces Meadow, our tool for visualising and interacting with Marlowe contracts. Section 5.1 reflects on the design rationale for Marlowe, showing how it can be supported on a variety of blockchains, and Section 5.2 explores how Marlowe can be implemented. Section 6 surveys related work and Section 7 enumerates the next steps for the project after drawing some conclusions.

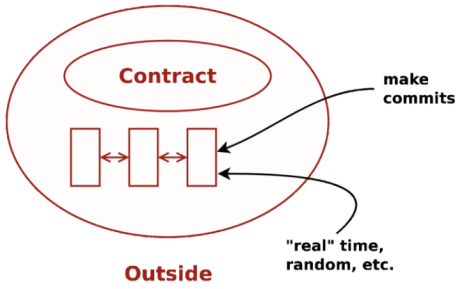

Fig. 1. The context for a contract

2 The Marlowe model

The Marlowe domain-specific language (DSL) is modelled as an algebraic type in Haskell, together with an executable small-step semantics. We start by looking at the different types used by the model, and the assumptions about the infrastructure in which contracts will be run. We then we look at the `Contract` DSL itself, and finally we give its semantics in Haskell. Section 3 revisits the "escrow" example using the embedding of Marlowe as a DSL in Haskell.

2.1 The model types

A running contract interacts with its environment in two ways, as in Figure 1.

Observables First, it will need to observe different kinds of varying quantities including, for example, the current time, the current block number and, random numbers, as well as "real world" quantities like "the price of oil" or "the exchange rate between currencies A and B". As the examples illustrate, observables come both from aspects of the blockchain (e.g. the current block number) and externally. In the latter case, it will be necessary to agree a trusted oracle or beacon giving the value.

Each instance of such an observable will be observed at a particular time and in a particular context. We assume that the system infrastructure ensures that these values are recorded on the blockchain to allow the computation to be repeated for verification purposes.

It is assumed that at each step of the execution of the contract, the values of observables will be available if needed, and these values are (together) given by a value of type `OS` (for "observable set"), where individual observations are described in a "little language" for that purpose: `Observation`. Note that these values are not determined by the participants in the contract, but rather by the external environment in which the contract is run.

Inputs and Commitments On the other hand, at each step there are – potentially, at least – a variety of inputs available from the participants themselves. These

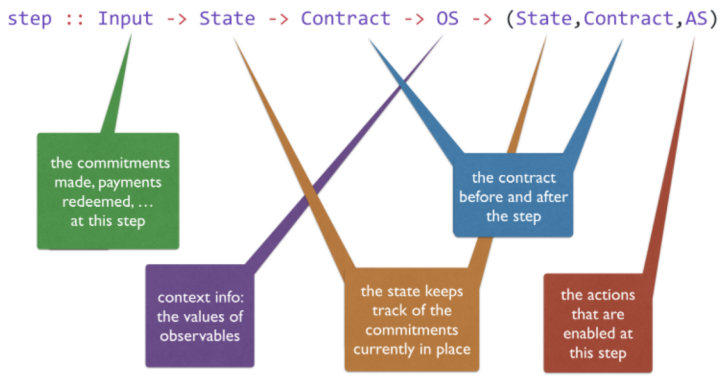

Fig. 2. The `step` function

include commitments of currency (or "cash"), redemption of commitments, and claims of payments by a participant. Moreover, it is also possible for a participant to input an arbitrary value (which we term a "choice"). The particular inputs at a given step are described by a value of type `Input`.

While informally we might see a commitment to something as being indefinite, it is important to realise that, on blockchain, a commitment needs to have a timeout so that progress can be forced in a contract. After the timeout period the cash can be refunded through the user creating a transaction to reclaim the cash. Information about the commitments currently in force forms the `State`, which can be modified at each execution step.

Actions Payments can be granted by using committed money, but they must be manually redeemed by the recipient, in the same way that cash commitments are redeemed when they expire. The effects of the contract in the blockchain are represented by a list `AS` of `Actions` that is derived from the execution of each step of the semantics.

Infrastructure The model makes a number of assumptions about the blockchain infrastructure in which it is run.

- It is assumed that cryptographic functions and operations are provided by a layer external to the system, and so they need not be modelled explicitly.
- We assume that time is "coarse grained" and measured by block number, so that, in particular, timeouts are delimited using block numbers.
- Making a commitment is not something that a contract can perform; rather, it can request that a commitment is made, but that then has to be established externally: hence the input of (a set of) commitments at each step.
- The model manages the release of funds back to the committer when a cash commitment expires (see discussion of the `stepBlock` function below).

Computation Computation is modelled at two different levels.

The `step` function represents a *single computation step* and has this type:

```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

which is also illustrated in Figure 2. The `step` function is total, so that for every contract a result of stepping is defined. However, for some kinds of contracts – commits, redeems or time-shifted contracts – it is possible that performing a step produces the same contract as the result; we call these *quiescent* steps whereas all others *make progress*. We use this distinction in the explanation that follows.

Execution of a contract will involve multiple blocks, with multiple steps in each block. The computation at a single block is given by the `stepBlock` function: this will call the `stepAll` function that calls `step` repeatedly until it is quiescent.

In addition to calling `stepAll`, `stepBlock` will first enable expired cash commitments to be refunded and record, in the state, any choices made at that step. The functions `stepAll` and `stepBlock` have the same type as `step` itself.

2.2 The `Contract` type

The type of contracts is given by the following Haskell data type:

```
data Contract =
   Null |
   CommitCash IdentCC Person Money Timeout Timeout Contract Contract |
   RedeemCC IdentCC Contract |
   Pay IdentPay Person Person Money Timeout Contract |
   Both Contract Contract |
   Choice Observation Contract Contract |
   When Observation Timeout Contract Contract
```

Informally, this type provides a `Null` contract, which does nothing. The next three constructs form contracts that do something, and then continue according to another contract (which is one of the components of the original contract). `CommitCash` will wait for a participant to make a commitment, `RedeemCC` allows for a commitment to be redeemed, and `Pay` for a payment between participants to be claimed by the recipient.

The remaining constructors form composite contracts from simpler components: `Both` has the behaviour of both its components, `Choice` chooses between two contracts on the basis of an observation, and `When` is quiescent until a condition – i.e. an `Observation` – becomes true.

Additionally, many of the contracts have timeouts that also determine their behaviour.

2.3 The `step` function

In this section, we explain the detailed behaviour of contracts by describing how the `step` function operates on each of the constructors of the `Contract` type.

– `Null` is the null contract; it will always be quiescent:

```
step _ st Null _ = (st, Null, [])
```

– `CommitCash ident person val start_timeout end_timeout con1 con2`
For this contract to make progress,
  - either, before the timeout `start_timeout`, the user `person` makes a cash commitment of value `val` and timeout `end_timeout` with the identifier `ident`,
  - or the timeout `start_timeout` is exceed:

```
step
  commits
  st
  c@(CommitCash ident person val start_timeout end_timeout con1 con2)
  os
  | cexe || cexs = (st {sc = ust}, con2, [])
  | Set.member (CC ident person cval end_timeout) (cc commits)
        = (st {sc = ust}, con1, [SuccessfulCommit ident person cval])
  | otherwise = (st, c, [])
  where ccs = sc st
        cexs = expired (blockNumber os) start_timeout
        cexe = expired (blockNumber os) end_timeout
        cns = (person, if cexe || cexs
                       then ManuallyRedeemed
                       else NotRedeemed cval end_timeout)
        ust = Map.insert ident cns ccs
        cval = evalMoney st val
```

In the first case, a `SuccessfulCommit` action is generated and the contract continues as `con1`; in the second case no action is generated and the contract continues as `con2`. While neither case holds, the contract is quiescent, waiting for the cash to be committed.
If the cash is committed successfully and the timeout `end_timeout` is reached, then it is impossible to further spend the committed cash, and any unspent funds can be reclaimed by `person`. This is enforced by the `stepBlock` function, as noted above.

– `RedeemCC ident con` (`CC` stands for cash commitment.) For this contract to make progress, the creator of the cash commitment with identifier `ident` is allowed to redeem the unspent funds in that commitment; the contract then continues as `con`, and the action `CommitRedeemed` is produced.

```
step commits st c@(RedeemCC ident con) _ =
    case Map.lookup ident ccs of
      Just (person, NotRedeemed val _) ->
        let newstate =
                st {sc = Map.insert ident (person, ManuallyRedeemed) ccs} in
        if Set.member (RC ident person val) (rc commits)
        then (newstate, con, [CommitRedeemed ident person val])
```

```
        else (st, c, [])
      Just (person, ManuallyRedeemed) ->
        (st, con, [DuplicateRedeem ident person])
      Nothing -> (st,c,[])
  where
      ccs = sc st
```

Committed cash can only be redeemed once, and an attempt to redeem it a second time will produce a `DuplicateRedeem` action, continuing as `con`.
If the cash commitment with identifier `ident` has expired, it becomes possible for the remaining funds to be redeemed by the committer; this can be done by the `stepBlock` function processing the appropriate `Input`, and an `ExpiredCommitRedeemed` action will be produced.
Once the commitment `ident` has expired and is redeemed, a `RedeemCC ident con` contract will immediately evolve to `con`.

– `Pay idpay from to val expi con` makes it possible, assuming that sufficient funds are available, for `to` to claim a payment with id `idpay` of `val` from `from` before the timeout `expi`. The contract continues as `con`.

```
step inp st c@(Pay idpay from to val expi con) os
  | expired (blockNumber os) expi = (st, con, [ExpiredPay idpay from to cval])
  | right_claim =
    if ((committed st from bn >= cval) && (cval >= 0))
      then (newstate, con, [SuccessfulPay idpay from to cval])
      else (st, con, [FailedPay idpay from to cval])
  | otherwise = (st, c, [])
  where
    cval = evalMoney st val
    newstate = stateUpdate st from to bn cval
    bn = blockNumber os
    right_claim =
      case Map.lookup (idpay, to) (rp inp) of
        Just claimed_val -> claimed_val == cval
        Nothing -> False
```

By 'available' we mean that sufficient commitments have been made and not yet expired to cover the payment; in this case, the payment uses the currency allocated by the cash commitments made by `from` that expire the earliest.
This contract will result in a `FailedPay` action if the funds are not available; otherwise a `SuccessfulPay` action is generated.

– `Both con1 con2` enforces the behaviour of both contracts `con1` and `con2`.

```
step comms st (Both con1 con2) os =
    (st2, result, ac1 ++ ac2)
    where
        result | res1 == Null = res2
               | res2 == Null = res1
               | otherwise = Both res1 res2
```

```
        (st1,res1,ac1) = step comms st con1 os
        (st2,res2,ac2) = step comms st1 con2 os
```

Because the model is stateful and produces output actions, to make a step, it is necessary to execute a single step of each of the contracts `con1` and `con2` in sequence: first `con1` then `con2`.

– `Choice obs conT conF` behaves as either `conT` or `conF` depending on the (Boolean) result of `obs` at the time that the observation is made, `conT` if it is `True` and `conF` if `False`.

```
step _ st (Choice obs conT conF) os =
    if interpretObs st obs os
        then (st,conT,[])
        else (st,conF,[])
```

– `When obs expi con con2` This contract will not progress until `obs` is `True` or until the current block number is greater than or equal to the one specified by timeout `expi`. In case the timeout applies, the contract will continue as `con2`, if the timeout does not apply and `obs` is `True`, then the contract continues as `con`. Otherwise the contract is quiescent.

```
step _ st (When obs expi con con2) os
  | expired (blockNumber os) expi = (st,con2,[])
  | interpretObs st obs os = (st,con,[])
  | otherwise = (st, When obs expi con con2, [])
```

We look next at an example of Marlowe in action.

3 Marlowe as an embedded DSL

In this section, we revisit the escrow example that we discussed briefly in the introduction, and show how we can make Marlowe contracts that are easier to write, read, and understand, by embedding them into Haskell code, that is, taking advantage of the fact that Marlowe contracts are implemented as Haskell terms to write Haskell programs that generate Marlowe code, instead of writing Marlowe directly.

We used Haskell because it is the language in which Marlowe is implemented, but it would be easy to embed Marlowe in any other language. It would only be necessary to translate its primitives into a data type in that language. In Meadow we use Fay [5], a subset of Haskell that we discuss in more detail in Section 4).

The example we use through this section implements an escrow contract, as first introduced in Section 1. The escrow mechanism allows `alice` to deposit the money into a contract, in a way that the money will only be released when two out of three participants agree on whether `bob` has indeed given `alice` the item.

The escrow participant (`carol`) is supposed to be a neutral third party that will decide in case of dispute. This way, if participants `alice` and `bob` are honest, they will just agree on the result of the transaction and `carol` will not need to

do anything. If `alice` and `bob` disagree, `carol` will be able to choose whether the money must go to `alice` or to `bob`.

In our implementation we make things more specific: the money paid for the item is 450 ADA, and it must be committed by `alice` before block `10`; it will be refunded to `alice` if there is no consensus before block `90`.

We start by defining some Haskell functions. We can encapsulate identifiers in functions to make the contract more readable. That way we can generate an identifier for the cash commitment:

```
iCC1 :: IdentCC
iCC1 = IdentCC 1
```

An identifier for the payment:

```
iP1 :: IdentPay
iP1 = IdentPay 1
```

And we can create identifiers for all the participants:

```
alice, bob, carol :: Person
alice = 1
bob   = 2
carol = 3
```

We can also create a sub-contract that allows the money from the commitment with identifier `IdentCC 1` to be redeemed:

```
redeem_original :: Contract
redeem_original = RedeemCC iCC1 Null
```

Once redeemed, the contract continues as `Null` since we expect this to be the last thing that is done. Each participant has a say on who deserves the money: either `alice` deserves a refund, which we represent with the number 0; or `bob` deserves a payment, which we represent with the number 1. Because there is only one choice to make per participant, we use the same `IdentChoice` as their participant id. We can define a function that returns an observation that is true if and only if `per` has chosen the number `c` for the choice `IdentChoice per` as follows:

```
chose :: Int -> ConcreteChoice -> Observation
chose per c = PersonChoseThis (IdentChoice per) per c
```

Then, we can easily write a function that returns an observation that is `True` if and only if at least one of the participants `per` and `per'` has chosen the number `val` as follows:

```
one_chose :: Person -> Person -> ConcreteChoice -> Observation
one_chose per per' val = (OrObs (chose per val) (chose per' val))
```

Building on that, we can now write a function that returns an observation that is `True` if and only if at least two out of the three participants `p1`, `p2`, and `p3` have agreed in choosing the number `c` as follows:

```
two_chose :: Person -> Person -> Person -> ConcreteChoice -> Observation
two_chose p1 p2 p3 c = OrObs (AndObs (chose p1 c) (one_chose p2 p3 c))
                             (AndObs (chose p2 c) (chose p3 c))
```

Finally, we can write the escrow contract, thus:

```
escrow :: Contract
escrow = CommitCash iCC1 1 (ConstMoney 450) 10 100
                    (When (OrObs (two_chose alice bob carol 0)
                                 (two_chose alice bob carol 1))
                          90
                          (Choice (two_chose alice bob carol 1)
                                  (Pay iP1 alice bob (AvailableMoney iCC1) 100
                                       redeem_original)
                                  redeem_original)
                          redeem_original)
                    Null
```

The outermost primitive `CommitCash` allows the `alice` to commit `450` ADA before block `10`, with the promise that money will be released on block `100` if they are not claimed before that.

The next primitive, `When`, waits for one of three things to happen:

1. The observation became `True` because two out of three have chosen `0`.
2. The observation became `True` because two out of three have chosen `1`.
3. The observation remained `False` until `90` was published in the blockchain.

If the third option happens, the money is refunded. Otherwise, there is a `Choice` that will immediately refund the money to `alice` unless two out of three chose option `1`; in the later case, `Pay` will give `bob` the opportunity to claim the funds available in the commitment with identifier `iCC1` before block `100` (by using the identifier `iP1` for the claim). If the funds are not claimed before block `100` they will also be refunded to `alice`.

Reflecting on the example, we can see that using Haskell definitions has made the contract substantially more comprehensible. While our current implementation does not do this, it is possible to modify the embedding to support more efficient operation, by preserving *sharing* in the host language. For example, if we were to replace the repeated expression `two_chose alice bob carol 1` by a `where` clause,

```
escrow = ...        (When (OrObs (two_chose alice bob carol 0)
                                 chose_refund)
                          90
                          (Choice chose_refund ...
                           ...
          where chose_refund = two_chose alice bob carol 1
```

then the repeated computation of the expression could be avoided.

4 Visualising and interacting with Marlowe contracts

For Marlowe to be usable in practice, users need to be able to understand how contracts will behave once deployed to the blockchain, but without doing the deployment. We can do that by simulating their behaviour off-chain, interactively stepping through the evaluation of a contract in a browser. We do this in two stages, first transforming an embedded contract (using features of Haskell) to a pure Marlowe contract, and then interactively stepping through that contract.

To achieve this, and to aid Marlowe's take-up usage by people that does not know or are not familiar with its syntax, we have developed Meadow, a web tool that supports the interactive construction, revision, and simulation of smart-contracts written in Marlowe. The tool is publicly available in the url: `https://input-output-hk.github.io/scdsl/`. In Figure 3, we provide a screenshot of Meadow in the middle of simulating the execution of the "deposit incentive" contract available in the GitHub repository (in the file: `src/DepositIncentive.hs`) and in the examples section of Meadow (on the bottom right part).

Meadow has been mainly written in Haskell and compiled to JavaScript partially by the Haste compiler [6] and partially by GHCJS [13]; Meadow also relies on the Blockly library [8] for providing a visual editor for smart contracts written in Marlowe. Embedding support in Meadow is provided through a pruned and bundled version of the Fay compiler [5], compiled to JavaScript using GHCJS. All text fields that edit and present Meadow and Fay code are instances of CodeMirror text editor [9].

Blockly's editor allows the user to visualise and edit smart-contracts as interlocking blocks that can be dragged and dropped like pieces of a jigsaw puzzle. Meadow also provides functionality to generate syntactically correct and formatted code (that is displayed in the upper right corner of the application), and to convert the code back to its Blockly representation.

Additionally, Meadow allows the user to use Meadow's Fay embedding from within the browser.

The reason that we chose Fay instead of Haskell for Meadow is a technical one: we wanted Meadow to be run completely inside the browser, because that makes it easy to deploy, since the server is only required to send the page and not to run any Marlowe-related computation. There exist compilers from Haskell to JavaScript but most of them are not easy to bootstrap into JavaScript. Among other reasons, they often rely on the API of the host OS to read and write files.

The Fay compiler, on the other hand, is mostly written in pure Haskell, and it is not hard to compile to JavaScript by using GHCJS. Nevertheless, it does not include a type checker, it relies in GHC's type-checker; and it also tries to read its `Prelude` from disk. We have worked around these issues by disabling type-checking within Meadow embedded editor, and by embedding all the required modules as constant strings in a modified version of the Fay compiler.

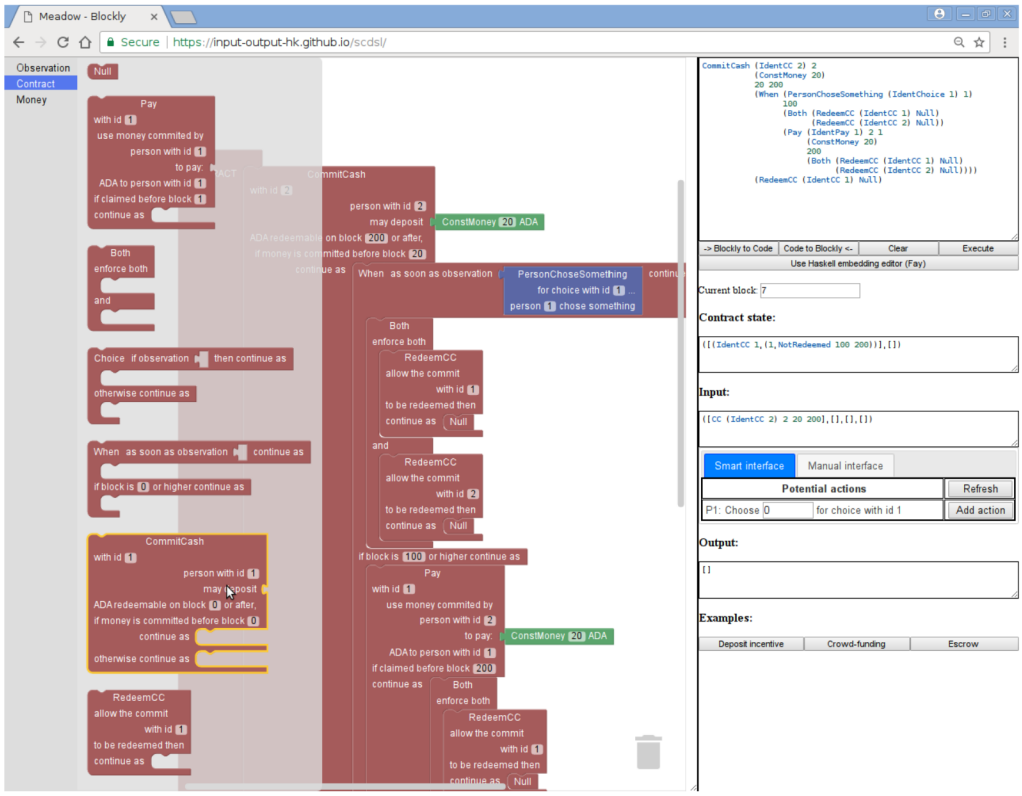

Fig. 3. The Meadow tool simulating the "deposit incentive" contract.

In particular, Meadow includes a pruned copy of the Fay compiler, bundled with its Prelude module, its foreign interface, and a module with the definitions of the Marlowe primitives and a function to pretty-print Marlowe contracts. This allows the user to compile and execute, inside Meadow, Fay code that generates and prints a Marlowe contract. When the user opens the embedded editor, the current Marlowe contract is embedded into a template of a Fay program that prints it. The user is able to modify this code in the left panel of the editor and use the advanced functionalities provided by Fay like, for example, bindings, list-comprehensions, turing-complete functions definitions, etc.

When the user clicks the execute button, the Fay code is compiled to JavaScript and evaluated; this causes the Marlowe contract generated by the execution of the code to be written to the panel on the right. Generated contracts can then be sent back to the main screen of Meadow, translated to Blockly, and their execution can be simulated. The embedded editor also allows users to save the Fay code to a file in their computers, to load code from a file in their computers, and to temporarily hide the editor while keeping its contents in memory, even while the compilation and execution process is being carried out in the background.

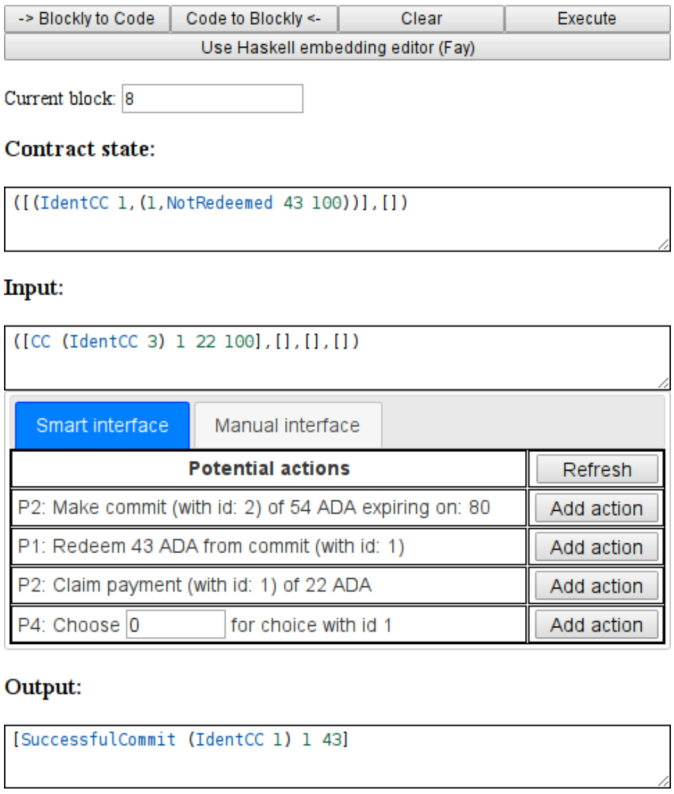

Fig. 4. Detail of the interface for contract execution simulation.

The execution of complete contracts can be simulated block by block by using the panel on the right (see Figure 4), which includes text fields to view and edit the current block number, the state of the contract, the current inputs, and the outputs from last block.

Additionally, to facilitate the introduction of inputs, Meadow provides two different interfaces:

- The manual interface, provides a template for each of the four possible types of input: commits, redeems, payment claims, and choices.
- The smart interface (shown in Figure 4), calculates the possible operations that would make sense given the current inputs, state of the contract, block number, and remaining contract; and it provides them in a table with most of the parameters already filled in.

The smart interface is usually more convenient to use than the manual interface since the later provides between 3 and 4 fields for each operation whereas the former "guesses" the possible intentions of the user and usually can input new operations with a single click (except for choices, which still may require the user to input a number).

In the next section, we re-examine the model and the Marlowe language, looking at possible extensions, alternative design decisions and ways in which we can formally analyse Marlowe contracts.

5 Design and Implementation

Marlowe is defined by its executable semantics, but to be deployed on blockchain it will have to be implemented on an existing distributed infrastructure. We intend to deploy it on IOHK's Cardano infrastructure, but it can be implemented in other systems too, as we explain now.

5.1 Design Rationale

Marlowe abstracts away from a number of concrete details of how blockchains operate. In particular, it is agnostic between UTxO-based systems, such as Bitcoin and Cardano SL, and account-based models such as Ethereum; it can be implemented on "push" or "pull"-based systems, and can be executed on or off-chain.

UTxO and accounts Transactions on Bitcoin are made by spending the (as yet) unspent outputs of previous transactions ('unspent transaction outputs' or UTxOs). The chain need not maintain any state, such as the current value owned by any particular participant: such *account* information is implicit and external to the Bitcoin blockchain. On the other hand, the Ethereum model explicitly keeps track of account values, and this information needs to be kept on chain. Of the two, the UTxO model is simpler and requires less support from the implementation of the chain itself.

Interaction modality Contracts can be conceived of as acting in two ways. In a *push* model, contracts are seen to make things happen. In the case of blockchain, this would include making payments or transactions take place. Alternatively, in a *pull* model, contracts enable certain things, such as transactions, to happen but the transactions need to be effected by an external actor. The pull model makes fewer demands on the blockchain implementation than the push model.

Layered design and sidechains Some chains have a layered design, in particular Cardano [11]. The Cardano Settlement Layer (SL) is to support settlement of transactions but nothing more complicated, whereas the Computation Layer (or CL) supports more powerful general computations over the chain. Moreover, the Cardano roadmap [3] envisages the possibility of computation taking place, in part at least, on a sidechain rather than on the main chain itself, which may support only the SL. Other off-chain approaches include the Lightning network, overlaid on blockchains (usually Bitcoin) and the channels of æternity [1].

5.2 Compilation

We have designed Marlowe assuming that each contract would be compiled into

- one or more on-chain programs and
- a client side program or interface that has a limited number of choices to be made at each point in time (including an interface for the observables).

The contracts implemented in the Marlowe language have an implicit interface defined for what would correspond to the client side of a distributed application, as shown by the interactive interface. Of course, we probably do not want to give a user a programmatic interface but an actual application with user interface.

A contract written in Marlowe could also be potentially compiled to a smart-contract written in Ethereum. This could be done in two ways: it could be compiled so that a smart contract represents a single instance of the Marlowe contract with a fixed set of participants; alternatively it would be possible to create a generic contract with an extra "instantiate" operation that sets up a new Marlowe contract instance with a given set of participants (set of public keys).

Transactions In principle, we can directly translate each operation in the model (i.e: cash commitment, payment claim, choice, or redemption) into a separate transaction in the blockchain. In the Ethereum model this would work in the same way that is simulated by Meadow: several operations can be issued in parallel and the blockchain subsystem will automatically apply them in some order and integrate them into the next block.

Another approach would be to use a UTxO model with continuations in which a UTxO represents a contract. In this model, an operation would correspond to a transaction that spends a UTxO and creates (at least) a new UTxO that represents the contract after the operation.

Usually, in the UTxO model, it is necessary that a UTxO is in the blockchain in order for another transaction to be able to spend it. This could potentially limit the number of operations that can be applied in a single block to one, which would make infeasible the approach of using the same UTxO to represent several instances of the same contract (since it would not scale). On the other hand, it would prevent non-determinism and, with it, it would remove the possibility of race-conditions. A change to the state of a contract before a transaction (and its corresponding operation) is accepted in the blockchain would invalidate the transaction and require the user to issue it again.

Even if we only allow one transaction per block, it would still be possible to combine several transactions off-chain and to issue a transaction that contains several operations. However, that would require an offline protocol independent from the blockchain and it would require participants to collaborate.

Nevertheless, there already exists in Bitcoin a mechanism that allows the combination of several chained transactions within a single block while using the UTxO model, even though it is used to solve a different problem and it is not widely spread yet. This is the case of "child pays for parent", which allows the recipient of an unpublished transaction to issue another transaction that

spends the former, with the aim of including enough fees for both transactions to have enough incentive to be included by a miner in a block (in which case both transactions would be included in the same block).

The same mechanism could be used for allowing miners to aggregate several transactions that act as a chain in the same block. The clients would only need to monitor the transaction pool in order to send transactions that can be chained with the existing ones.

Cost The Cardano platform has a notion of cost, and Marlowe contracts will potentially incur costs of two kinds: The cost of a single transaction execution (e.g. a single Plutus validator), and the cost of the whole contract, which may consist of many transactions.

Even in Ethereum this distinction is important because single contract execution is limited by gas (and is thus finite), whereas the life of the whole contract may be unbounded and consist of an indefinite number of transactions. So far we are assuming contracts defined in Marlowe to be finite in both aspects.

Recording information on chain Inputs and values of observables need to be recorded somehow, and this raises the potential issue about bloating the chain with data. In general, it will only be necessary to store a signed hash value of the observable, and this will keep data usage bounded; but the full information may still need to be posted in case of dispute.

General computations If we have contracts that involve, for example, the oil price then we may need to convert a published price in USD into ADA, using a value for the prevailing USD/ADA exchange rate. This will require a computation: in this case, a multiplication. If Marlowe is to be stand-alone then we need to extend it with arithmetic and other operations, but we expect Marlowe to be embedded in a suitable language that provides these facilities.

6 Related work

Blockchains are executed in a replicated form by parties who cannot be guaranteed not to be hostile, either by directly trying to change the contents of the chain, or through trying to affect other properties of the chain by indirect means (such as swamping honest parties with work). Programming on the blockchain therefore needs to be constrained in some ways, since it has to be amenable to replication or verification within a reasonable time if the security and integrity of the chain are to be preserved. We look at representatives of the main approaches now; we discuss some of these in more detail in an earlier paper [12].

Split contracts An early work on smart contracts raises the issue that practical contractual situations may well not be amenable to complete formalisation, hence giving rise to a *split* [14] between an automated part and a non-automated part that mediates, for example, real assets. We can foresee that this approach will be needed for Marlowe too, if it achieves read-world adoption.

Bitcoin script One approach to this is to choose mechanisms such as bitcoin script [17] which are manifestly non-Turing complete. A bitcoin script, written in a Forth-like language, is essentially linear: it can branch, but the language contains neither looping constructs nor recursion. It is therefore straightforward not only to see that scripts will terminate, but also to give an accurate estimate of the time taken to execute a script.

Ethereum On the other hand, the Ethereum system [19] provides a Turing-complete language for the EVM virtual machine, and a higher-level programming language, Solidity, that compiles into EVM code. However, EVM and Solidity programs are constrained *post hoc* by two mechanisms: program execution must be paid for using 'gas' proportional to the effort expended, and a set of *ad hoc* limits on program execution, e.g. on stack size.

Nxt In Nxt [15], programmability of the system is provided through a "fat" high-level API, which is accessible from Nxt clients through a REST interface. The API provides functionality supporting various kinds of transactions. The core software itself does not support any form of scripting language; rather, users are expected to work with the built in transaction types and transactions that support some 250 primitive operations; these can be "scripted" in a client (only) using a binding to the API, which is available, for instance, in JavaScript.

Multiple languages A common feature of many blockchain platforms is that they provide multiple scripting languages. As we noted earlier, Ethereum can be programmed at the EVM level as well as by using the high-level Solidity language. Tezos [7] supports the stack-based, strongly-typed, functional language Michelson, but also provides a high-level language, Liquidity, that compiles into Michelson. Liquidity is also functional and strongly typed, but provides more constructs in a more familiar syntax, namely that of a subset of OCaml.

The æternity system [1] provides multiple languages and VMs [18]: the functional language Sophia (akin to Reason) and the Functional Typed Warded Virtual Machine (FTWVM) for safe "system level programming", the language Varna and the HLM for simple contracts, and a (port of) Solidity and the EVM for compatibility with Ethereum. In a similar way, Cardano provides support for IELE [10], a rational reconstruction of the EVM, and thus for Solidity too.

Domain-specific languages A domain-specific language or DSL is a high-level language designed to work in a specific field or domain. The intention is that because the users will know about the field, the constructs of the language can be designed to be meaningful to them, and also that, because of its nature, the DSL need not include all the features of a general purpose language. Removing this clutter and having the remaining operations directly reflect the application area is intended to make the language more accessible to domain experts who do not necessarily see themselves as programmers.

Stand-alone DSLs have the advantage of providing appropriate, domain-level error messages when things go wrong, but suffer the disadvantage of having

to be implemented from scratch. An alternative to this are *embedded* DSLs (EDSLs), which provide a "little language" for the particular domain embedded within a general-purpose host language. This means that parts of the host – such as arithmetical expressions, or list idioms – can be used to extend the expressibility of the DSL. A notable example of this is the financial contracts language described by Peyton Jones and his collaborators [16], which is embedded in Haskell. Marlowe is similar to the DSL in [16] in the use of Haskell embedding, in that the functionality provided is similar, in the use of composable combinators, and in the declarative style that allows users to describe what needs to be enforced and not how. However the two approaches differ in other aspects like the lack of Marlowe's reliance on the legal system for enforcement, its support for multiple party contracts, its explicitness of choices, and its use of a pull model.

Findel The Findel project [2] examines financial contracts on the Ethereum platform, based on the seminal [16], and the authors note that payments need to be bounded; this is made concrete in our account by our notion of commitments. They take no account of commitments or timeouts as our approach does, but it should be noted that the Ethereum platform is more powerful than the one that we target in this paper.

7 Conclusions and Future Work

In this paper, we have presented Marlowe, a DSL for financial contracts on blockchains, based on earlier work on contracts [16], together with examples of its use. We have seen that to make this operational on blockchain we need to add commitments and timeouts, and to design a semantics that reflects these. As we saw in Section 5.1, Marlowe has been designed to make as few demands as possible on the underlying blockchain: it can be implemented on UTxO or account-based blockchains, for example.

We have also presented Meadow, that allows users to interact with and simulate the operation of Marlowe contracts, contributing to the potential adoptability of the system. We have also described the design rationale for Marlowe, and sketched ways in which it can be implemented.

We plan to continue the work with Marlowe in a number of directions. We will continue to develop the core language, for example considering the automation of generation of identifiers for commitments and others. We will then implement this version of Marlowe in Cardano, compiling from Marlowe contracts to on chain contracts and users' wallets, and deploy it in a test network to observe and measure its behaviour in practice.

Building on the operational semantics given here, we will develop analyses of Marlowe contracts – such as to show that contracts cannot generate `FailedPay` actions in certain circumstances. We will also develop QuickCheck-style property-based testing [4], and properties developed here can become candidates for fully fledged verification in a formalisation of the semantics of Marlowe.

We would like to thank IOHK for supporting us: not only has this led us to work on Marlowe, but we have benefited hugely from collaboration with colleagues including Manuel Chakravarty, Duncan Coutts, Bernardo David, Charles Hoskinson, Aggelos Kiayias, Bruno Woltzenlogel Paleo, Rebecca Valentine and Phil Wadler. We are very grateful to Thomas Arts and colleagues from æternity for their convivial discussions on blockchain and contracts in Göteborg, and to colleagues from the Universities of Kent and Leicester for their comments too.

References

1. æternity: æternity. `https://aeternity.com` (2018)
2. Biryukov, A., Khovratovich, D., Tikhomirov, S.: Findel: Secure derivative contracts for ethereum. In: Brenner, M., et al. (eds.) Financial Cryptography and Data Security. pp. 453–467. Springer International Publishing (2017)
3. Cardano: Why we are building Cardano. `https://whycardano.com` (2017)
4. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP '00. pp. 268–279. ACM (2000)
5. Done, C.: Fay. `https://github.com/faylang/fay/wiki` (2012), [last accessed 14-05-2018]
6. Ekblad, A.: Haste. `https://haste-lang.org/` (2012), [last accessed 26-03-2018]
7. Goodman, L.: Tezos – a self-amending crypto-ledger. `https://www.tezos.com/static/papers/white_paper.pdf` (2014)
8. Google: Blockly. `https://developers.google.com/blockly/` (2011), [last accessed 26-03-2018]
9. Haverbeke, M., et al.: CodeMirror text editor. `https://codemirror.net/` (2011), [last accessed 14-05-2018]
10. IELE Semantics. `https://github.com/runtimeverification/iele-semantics` (2016), [last accessed 26-03-2018]
11. IOHK: The Cardano Project. `https://iohk.io/projects/cardano/` (2017)
12. Lamela Seijas, P., Thompson, S., McAdams, D.: Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156 (2016), `https://eprint.iacr.org/2016/1156`
13. Mackenzie, H., Nazarov, V., Stegeman, L.: GHCJS. `https://github.com/ghcjs/ghcjs` (2010), [last accessed 14-05-2018]
14. Miller, M.: The Digital Path: Smart Contracts and the Third World. In: Birner, J., Garrouste, P. (eds.) Markets, Information and Communication: Austrian Perspectives on the Internet Economy. Taylor and Francis (2004)
15. Nxt. `https://nxtplatform.org/` (2013), [last accessed 26-03-2018]
16. Peyton Jones, S., et al.: Composing contracts: An adventure in financial engineering (functional pearl). In: Proceedings of the Fifth ACM SIGPLAN ICFP. ACM (2000)
17. Script – Bitcoin Wiki. `https://en.bitcoin.it` (2010), [last accessed 26-03-2018]
18. Stenman, E.: The æternity system. CODE BEAM SF, `https://www.youtube.com/watch?v=VXsqvfPIdWg` (2018 March)
19. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)