Marlowe: financial contracts on Cardano Computation Layer

Alexander Nemish

December 10, 2018

1 Introduction

Here we present a reference implementation of Marlowe, domain-specific language targeted at the execution of financial contracts in the style of Peyton Jones *et al* on Cardano Computation Layer.

The implementation is based on semantics described in paper *Marlowe: financial contracts on blockchain* by Simon Thompson and Pablo Lamela Seijas

We use PlutuxTx compiler, that compiles Haskell code into serialized *Plutus Core* code, to create a Cardano *Validator Script* that secures value.

This *Marlowe Validator Script* implements Marlowe interpreter, described in the paper.

2 Extended UTXO model

The *extended UTxO model* brings a significant portion of the expressiveness of Ethereum's account-based scripting model to the UTxO-based Cardano blockchain. The extension has two components: (1) an extension to the data carried by UTxO outputs and processed by associated validator scripts together with (2) an extension to the wallet backend to facilitate off-chain code that coordinates the execution of on-chain computations.

2.1 Extension to transaction outputs

In the classic UTxO model (Cardano SL in Byron and Shelley), a transaction output locked by a script carries two pieces of information:

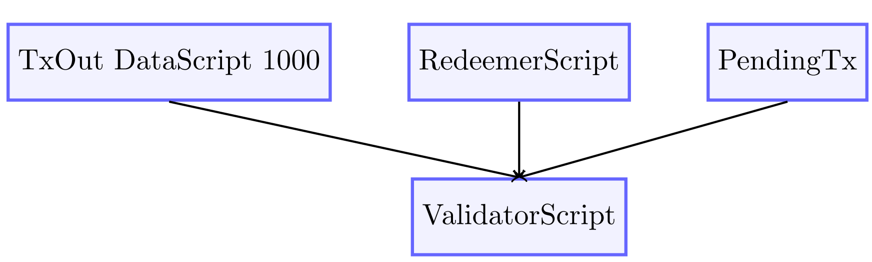
- it's value and
- (the hash of) a validator script.

We extend this to include a second script, which we call the *Data Script*. This second script is a *Plutus Core* expression, just like the *Validator Script*. However, the requirements on its type are different. The type of the data script can be any monomorphic type.

2.2 Extension to validator scripts

An extended validator script expects four arguments:

- the redeemer expression,
- the data script (from above),
- the output's value, and
- parts of the validated transaction and related blockchain state. (More detail is in the next section.)



We consider a validator script to have executed successful if it does not terminate in the *Plutus Core error* state.

2.3 Blockchain state available to validator scripts

Validator scripts receive, at a minimum, the following information from the validated transaction and the rest of the blockchain:

- the current block height (not including the currently validated transaction),
- the hash of the currently validated transaction,
- for every input of the validated transaction, its value and the hashes of its validator, data, and redeemer scripts,
- for every output of the validated transaction, its value and the hash of its validator and data script, and
- the sum of the values of all unspent outputs (of the current blockchain without the currently validated transaction) locked by the currently executed validator script.

3 Assumptions

- Fees are payed by transaction issues. For simplicity, assume zero fees.
- Every contract is created by contract owner by issuing a transaction with the contract in TxOut

4 Semantics

Marlowe Contract execution is a chain of transactions, where remaining contract and its state is passed through *Data Script*, and actions/inputs (i.e. *Choices* and *Oracle Values*) are passed as *Redeemer Script*

Validation Script is always the same Marlowe interpreter implementation, available below.

Both *Redeemer Script* and *Data Script* have the same structure:

$$(Input, MarloweData)$$

where

- *Input* contains contract actions (i.e. *Pay*, *Redeem*), *Choices* and *Oracle Values*,
- *MarloweData* contains remaining *Contract* and its *State*
- *State* is a set of *Commits* plus set of made *Choices*

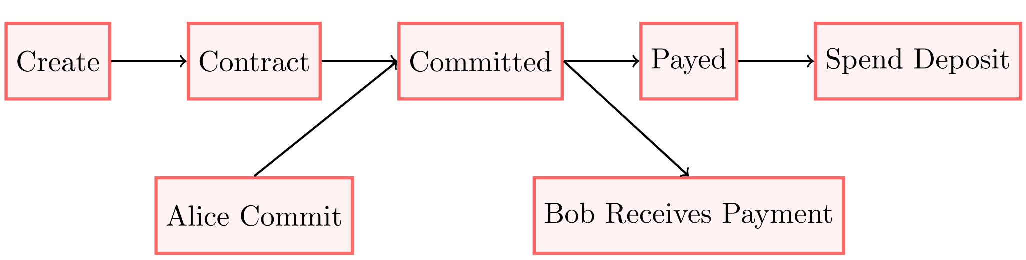
To spend TxOut secured by Marlowe Validator Script, a user must provide *Redeemer Script* that is a tuple of an *Input* and expected output of Marlowe Contract interpretation for the given *Input*, i.e. *Contract* and *State*.

To ensure that user provides valid remainig *Contract* and *State Marlowe Validator Script* compares evaluated contract and state with provided by user, and rejects a transaction if those don't match.

To ensure that remaining contract's *Data Script* has the same *Contract* and *State* as was passed with *Redeemer Script*, we check that *Data Script* hash is the same as *Redeemer Script*. That's why those are of the same structure

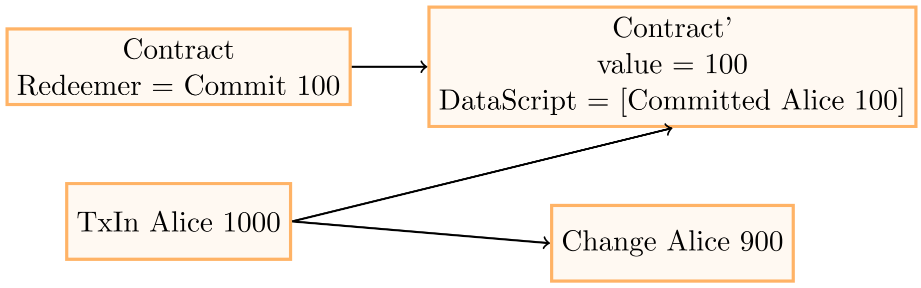
$$(Input, MarloweData)$$

.



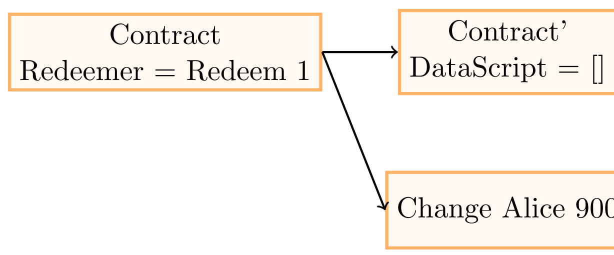
4.1 Commit

Alice has 1000 Ada.



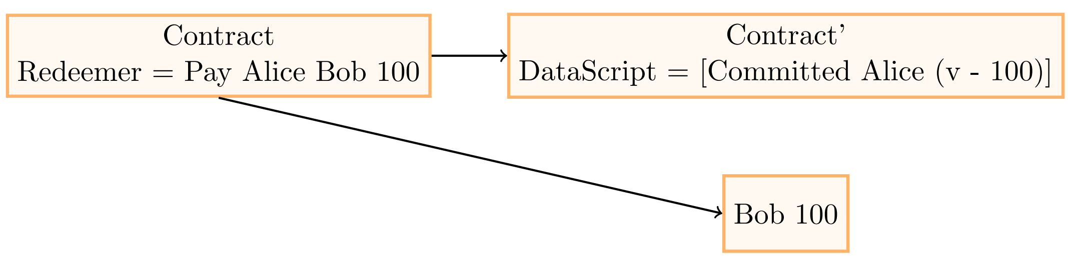
4.2 Redeem

Redeem a previously make CommitCash if valid. Alice committed 100 Ada with CC 1, timeout 256.



4.3 Pay

Alice pays 100 Ada to Bob.



4.4 SpendDeposit

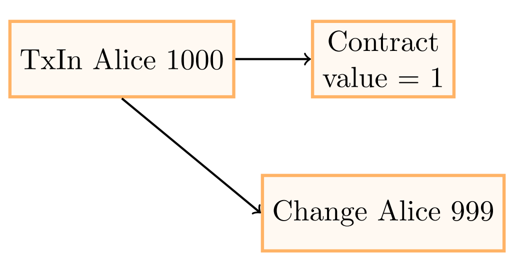
See 5

5 Contract Initialization

This can be done in 2 ways.

5.1 Initialization by depositing Ada to a new contract

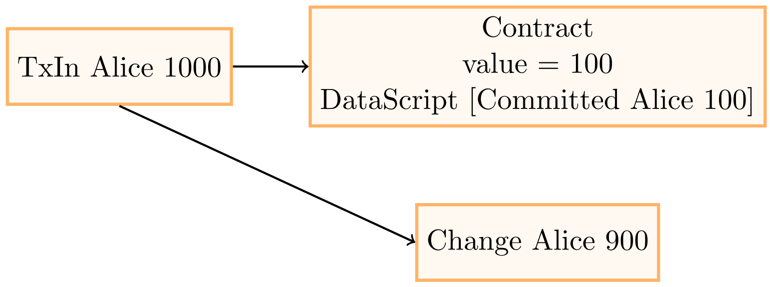
Just pay 1 Ada to a contract so that it becomes a part of *eUTXO*.



Considerations Someone need to spend this 1 Ada, otherwise all Marlowe contracts will be in UTXO. Current implementation allows anyone to spend this value.

5.2 Initialization by CommitCash

Any contract that starts with *CommitCash* can be initialized with actuall *CommitCash*



6 Implementation

6.1 Imports

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DefaultSignatures #-}
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TemplateHaskell #-}
{-# OPTIONS -fplugin=Language.PlutusTx.Plugin -fplugin-opt Language.PlutusTx.Plugin:dont-typechec
{-# OPTIONS_GHC -Wno-incomplete-uni-patterns -Wno-name-shadowing #-}
```

```
module Language.Marlowe.Compiler where
import Control.Applicative                    (Applicative (..))
import Control.Monad                          (Monad (..)
                                              , void
                                              )
import Control.Monad.Error.Class              (MonadError (..))
import Data.Maybe                             (maybeToList)
import qualified Data.Set                     as Set
import qualified Language.PlutusTx            as PlutusTx
import Wallet                                 (WalletAPI (..)
                                              , WalletAPIError
                                              , throwOtherError
                                              , signAndSubmit
                                              , ownPubKeyTxOut
                                              )
import Ledger                                 (DataScript (..)
                                              , Height (..)
                                              , PubKey (..)
                                              , TxOutRef'
                                              , TxOut'
                                              , TxIn'
                                              , TxOut (..)
                                              , ValidatorScript (..)
                                              , scriptTxIn
                                              , scriptTxOut
                                              )
import qualified Ledger                       as Ledger
import Ledger.Validation
import qualified Ledger.Validation as Validation
import qualified Language.PlutusTx.Builtins as Builtins
import Language.PlutusTx.Lift                 (makeLift)
```

6.2 Types and Data Representation

```
type Timeout = Int
type Cash = Int
```

```
type Person = PubKey
```

6.3 Identifiers

Commitments, choices and payments are all identified by identifiers. Their types are given here. In a more sophisticated model these would be generated automatically (and so uniquely); here we simply assume that they are unique.

```
newtype IdentCC = IdentCC Int
  deriving (Eq, Ord)
makeLift ''IdentCC
newtype IdentChoice = IdentChoice Int
  deriving (Eq, Ord)
makeLift ''IdentChoice
newtype IdentPay = IdentPay Int
  deriving (Eq, Ord)
makeLift ''IdentPay
type ConcreteChoice = Int
type CCStatus = (Person, CCRedeemStatus)
data CCRedeemStatus = NotRedeemed Cash Timeout
  deriving (Eq, Ord)
makeLift ''CCRedeemStatus
type Choice = ((IdentChoice, Person), ConcreteChoice)
type Commit = (IdentCC, CCStatus)
```

6.4 Input

Input is passed in *Redeemer Script*

```
data InputCommand = Commit IdentCC
  | Payment IdentPay
  | Redeem IdentCC
  | SpendDeposit
makeLift ''InputCommand
data Input = Input InputCommand [OracleValue Int] [Choice]
makeLift ''Input
```

6.5 Contract State

Commits MUST be sorted by expiration time, ascending.

```
data State = State {
  stateCommitted :: [Commit],
  stateChoices :: [Choice]
  } deriving (Eq, Ord)
makeLift ''State
emptyState :: State
emptyState = State {stateCommitted = [], stateChoices = []}
```

6.6 Value

Value is a set of contract primitives that represent constants, functions, and variables that can be evaluated as an amount of money.

```
data Value = Committed IdentCC
           | Value Int
           | AddValue Value Value
           | MulValue Value Value
           | DivValue Value Value Value   -- divident, divisor, default value (when divisor evaluates to 0)
           | ValueFromChoice IdentChoice Person Value
           | ValueFromOracle PubKey Value   -- Oracle PubKey, default value when no Oracle Value prov
             deriving (Eq)
makeLift ''Value
```

6.7 Observation

Representation of observations over observables and the state. Rendered into predicates by interpretObs.

```
data Observation = BelowTimeout Int   -- are we still on time for something that expires on Timeout?
    | AndObs Observation Observation
    | OrObs Observation Observation
    | NotObs Observation
    | PersonChoseThis IdentChoice Person ConcreteChoice
    | PersonChoseSomething IdentChoice Person
    | ValueGE Value Value   -- is first amount is greater or equal than the second?
    | TrueObs
    | FalseObs
  deriving (Eq)
makeLift '' Observation
```

6.8 Marlowe Contract Data Type

```
data Contract = Null
    | CommitCash IdentCC PubKey Value Timeout Timeout Contract Contract
    | RedeemCC IdentCC Contract
    | Pay IdentPay Person Person Value Timeout Contract
    | Both Contract Contract
    | Choice Observation Contract Contract
    | When Observation Timeout Contract Contract
      deriving (Eq)
makeLift '' Contract
```

6.9 Marlowe Data Script

This data type is a content of a contract *Data Script*

```
data MarloweData = MarloweData {
    marloweState :: State,
    marloweContract :: Contract
    }
makeLift '' MarloweData
data ValidatorState = ValidatorState {
    ccIds :: [IdentCC],
    payIds :: [IdentPay]
    }
makeLift '' ValidatorState
```

7 Marlowe Validator Script

Validator Script is a serialized Plutus Core generated by Plutus Compiler via Template Haskell.

```
marloweValidator :: ValidatorScript
marloweValidator = ValidatorScript result where
  result = Ledger.fromCompiledCode $$ (PlutusTx.compile [∨ λ
    (Input inputCommand inputOracles inputChoices :: Input, MarloweData expectedState expectedContract
    (_ :: Input, MarloweData {..} :: MarloweData)
    (pendingTx@PendingTx {pendingTxBlockHeight} :: PendingTx ValidatorHash) → let
```

7.1 Marlowe Validator Prelude

```
eqPk :: PubKey → PubKey → Bool
eqPk = $$(Validation.eqPubKey)
eqIdentCC :: IdentCC → IdentCC → Bool
```

```
eqIdentCC (IdentCC a) (IdentCC b) = a ≡ b

eqValidator :: ValidatorHash → ValidatorHash → Bool
eqValidator = $$(Validation.eqValidator)

¬ :: Bool → Bool
¬ = $$(PlutusTx.¬)

infixr 3 ∧
(∧) :: Bool → Bool → Bool
(∧) = $$(PlutusTx.and)

infixr 3 ∨
(∨) :: Bool → Bool → Bool
(∨) = $$(PlutusTx.or)

signedBy :: PubKey → Bool
signedBy = $$(Validation.txSignedBy) pendingTx

null :: [a] → Bool
null [] = True
null _  = False

reverse :: [a] → [a]
reverse l = rev l [] where
        rev [] a = a
        rev (x : xs) a = rev xs (x : a)

  -- it's quadratic, I know. We'll have Sets later
mergeChoices :: [Choice] → [Choice] → [Choice]
mergeChoices input choices = case input of
     choice : rest | notElem eqChoice choices choice → mergeChoices rest (choice : choices)
                   | otherwise → mergeChoices rest choices
     [] → choices
  where
     eqChoice :: Choice → Choice → Bool
     eqChoice ((IdentChoice id1, p1), _) ((IdentChoice id2, p2), _) = id1 ≡ id2 ∧ p1 `eqPk` p2

isJust :: Maybe a → Bool
isJust = $$(PlutusTx.isJust)

maybe :: r → (a → r) → Maybe a → r
maybe = $$(PlutusTx.maybe)

nullContract :: Contract → Bool
nullContract Null = True
nullContract _ = False

eqValue :: Value → Value → Bool
eqValue l r = case (l, r) of
     (Committed idl, Committed idr) → idl `eqIdentCC` idr
     (Value vl, Value vr) → vl ≡ vr
     (AddValue v1l v2l, AddValue v1r v2r) → v1l `eqValue` v1r ∧ v2l `eqValue` v2r
     (MulValue v1l v2l, MulValue v1r v2r) → v1l `eqValue` v1r ∧ v2l `eqValue` v2r
     (DivValue v1l v2l v3l, DivValue v1r v2r v3r) →
        v1l `eqValue` v1r
        ∧ v2l `eqValue` v2r
        ∧ v3l `eqValue` v3r
     (ValueFromChoice (IdentChoice idl) pkl vl, ValueFromChoice (IdentChoice idr) pkr vr) →
        idl ≡ idr
        ∧ pkl `eqPk` pkr
        ∧ vl `eqValue` vr
     (ValueFromOracle pkl vl, ValueFromOracle pkr vr) → pkl `eqPk` pkr ∧ vl `eqValue` vr
     _ → False

eqObservation :: Observation → Observation → Bool
eqObservation l r = case (l, r) of
     (BelowTimeout tl, BelowTimeout tr) → tl ≡ tr
     (AndObs o1l o2l, AndObs o1r o2r) → o1l `eqObservation` o1r ∧ o2l `eqObservation` o2r
     (OrObs o1l o2l, OrObs o1r o2r) → o1l `eqObservation` o1r ∧ o2l `eqObservation` o2r
     (NotObs ol, NotObs or) → ol `eqObservation` or
     (PersonChoseThis (IdentChoice idl) pkl cl, PersonChoseThis (IdentChoice idr) pkr cr) →
        idl ≡ idr ∧ pkl `eqPk` pkr ∧ cl ≡ cr
```

```
    (PersonChoseSomething (IdentChoice idl) pkl, PersonChoseSomething (IdentChoice idr) pkr) →
      idl ≡ idr ∧ pkl `eqPk` pkr
    (ValueGE v1l v2l, ValueGE v1r v2r) → v1l `eqValue` v1r ∧ v2l `eqValue` v2r
    (TrueObs, TrueObs) → True
    (FalseObs, FalseObs) → True
    _ → False
eqContract :: Contract → Contract → Bool
eqContract l r = case (l, r) of
    (Null, Null) → True
    (CommitCash idl pkl vl t1l t2l c1l c2l, CommitCash idr pkr vr t1r t2r c1r c2r) →
      idl `eqIdentCC` idr
      ∧ pkl `eqPk` pkr
      ∧ vl `eqValue` vr
      ∧ t1l ≡ t1r ∧ t2l ≡ t2r
      ∧ eqContract c1l c1r ∧ eqContract c2l c2r
    (RedeemCC idl c1l, RedeemCC idr c1r) → idl `eqIdentCC` idr ∧ eqContract c1l c1r
    (Pay (IdentPay idl) pk1l pk2l vl tl cl, Pay (IdentPay idr) pk1r pk2r vr tr cr) →
      idl ≡ idr
      ∧ pk1l `eqPk` pk1r
      ∧ pk2l `eqPk` pk2r
      ∧ vl `eqValue` vr
      ∧ tl ≡ tr
      ∧ eqContract cl cr
    (Both c1l c2l, Both c1r c2r) → eqContract c1l c1r ∧ eqContract c2l c2r
    (Choice ol c1l c2l, Choice or c1r c2r) →
      ol `eqObservation` or
      ∧ eqContract c1l c1r
      ∧ eqContract c2l c2r
    (When ol tl c1l c2l, When or tr c1r c2r) →
      ol `eqObservation` or
      ∧ tl ≡ tr
      ∧ eqContract c1l c1r
      ∧ eqContract c2l c2r
    _ → False
all :: () → forall a ∘ (a → a → Bool) → [a] → [a] → Bool
all _ = go where
    go _ [] [] = True
    go eq (a : as) (b : bs) = eq a b ∧ all () eq as bs
    go _ _ _ = False
eqCommit :: Commit → Commit → Bool
eqCommit (id1, (pk1, (NotRedeemed val1 t1))) (id2, (pk2, (NotRedeemed val2 t2))) =
    id1 `eqIdentCC` id2 ∧ pk1 `eqPk` pk2 ∧ val1 ≡ val2 ∧ t1 ≡ t2
eqChoice :: Choice → Choice → Bool
eqChoice ((IdentChoice id1, pk1), c1) ((IdentChoice id2, pk2), c2) =
    id1 ≡ id2 ∧ c1 ≡ c2 ∧ pk1 `eqPk` pk2
eqState :: State → State → Bool
eqState (State commits1 choices1) (State commits2 choices2) =
    all () eqCommit commits1 commits2 ∧ all () eqChoice choices1 choices2
findCommit :: IdentCC → [(IdentCC, CCStatus)] → Maybe CCStatus
findCommit i@(IdentCC searchId) commits = case commits of
    (IdentCC id, status) : _ | id ≡ searchId → Just status
    _ : xs → findCommit i xs
    _ → Nothing
fromOracle :: PubKey → Height → [OracleValue Int] → Maybe Int
fromOracle pubKey h@(Height blockNumber) oracles = case oracles of
    OracleValue pk (Height bn) value : _
      | pk `eqPk` pubKey ∧ bn ≡ blockNumber → Just value
    _ : rest → fromOracle pubKey h rest
    _ → Nothing
fromChoices :: IdentChoice → PubKey → [Choice] → Maybe ConcreteChoice
fromChoices identChoice@(IdentChoice id) pubKey choices = case choices of
    ((IdentChoice i, party), value) : _ | id ≡ i ∧ party `eqPk` pubKey → Just value
```

```
    _ : rest → fromChoices identChoice pubKey rest
    _ → Nothing
elem :: (a → a → Bool) → [a] → a → Bool
elem = realElem
  where
    realElem eq (e : ls) a = a `eq` e ∨ realElem eq ls a
    realElem _ [] _ = False
notElem :: (a → a → Bool) → [a] → a → Bool
notElem eq as a = ¬ (elem eq as a)
```

7.2 Contract Validation

Here we check that *IdentCC* and *IdentPay* identifiers are unique.

```
validateContract :: ValidatorState → Contract → (ValidatorState, Bool)
validateContract state@(ValidatorState ccIds payIds) contract = case contract of
    Null → (state, True)
    CommitCash ident _ _ _ _ c1 c2 →
      if notElem eqIdentCC ccIds ident
      then checkBoth (ValidatorState (ident : ccIds) payIds) c1 c2
      else (state, False)
    RedeemCC _ c → validateContract state c
    Pay ident _ _ _ _ c →
      if notElem (λ(IdentPay a) (IdentPay b) → a ≡ b) payIds ident
      then validateContract (ValidatorState ccIds (ident : payIds)) c
      else (state, False)
    Both c1 c2 → checkBoth state c1 c2
    Choice _ c1 c2 → checkBoth state c1 c2
    When _ _ c1 c2 → checkBoth state c1 c2
  where
    checkBoth :: ValidatorState → Contract → Contract → (ValidatorState, Bool)
    checkBoth state c1 c2 = let
      (us, valid) = validateContract state c1
      in if valid then validateContract us c2
      else (state, False)
```

7.3 Value Evaluation

```
evalValue :: State → Value → Int
evalValue state@(State committed choices) value = case value of
  Committed ident → case findCommit ident committed of
    Just (_, NotRedeemed c _) → c
    _ → 0
  Value v → v
  AddValue lhs rhs → evalValue state lhs + evalValue state rhs
  MulValue lhs rhs → evalValue state lhs ∗ evalValue state rhs
  DivValue lhs rhs def → do
    let divident = evalValue state lhs
    let divisor = evalValue state rhs
    let defVal  = evalValue state def
    if divisor ≡ 0 then defVal else divident `div` divisor
  ValueFromChoice ident pubKey def → case fromChoices ident pubKey choices of
    Just v → v
    _ → evalValue state def
  ValueFromOracle pubKey def → case fromOracle pubKey pendingTxBlockHeight inputOracles of
    Just v → v
    _ → evalValue state def
```

7.4 Observation Evaluation

```
interpretObs :: Int → [OracleValue Int] → State → Observation → Bool
interpretObs blockNumber oracles state@(State _ choices) obs = case obs of
```

```
  BelowTimeout n → blockNumber ⩽ n
  AndObs obs1 obs2 → go obs1 ∧ go obs2
  OrObs obs1 obs2 → go obs1 ∨ go obs2
  NotObs obs → ¬ (go obs)
  PersonChoseThis choice_id person reference_choice →
    maybe False (≡ reference_choice) (find choice_id person choices)
  PersonChoseSomething choice_id person → isJust (find choice_id person choices)
  ValueGE a b → evalValue state a ⩾ evalValue state b
  TrueObs → True
  FalseObs → False
  where
    go = interpretObs blockNumber oracles state
    find choiceId@(IdentChoice cid) person choices = case choices of
      (((IdentChoice id, party), choice) : _)
        | cid ≡ id ∧ party `eqPk` person → Just choice
      (_ : cs) → find choiceId person cs
      _ → Nothing
orderTxIns :: PendingTxIn → PendingTxIn → (PendingTxIn, PendingTxIn)
orderTxIns t1 t2 = case t1 of
  PendingTxIn _ (Just _ :: Maybe (ValidatorHash, RedeemerHash)) _ → (t1, t2)
  _ → (t2, t1)
currentBlockNumber :: Int
currentBlockNumber = let Height blockNumber = pendingTxBlockHeight in blockNumber
```

7.5 Contract Evaluation

```
eval :: InputCommand → State → Contract → (State, Contract, Bool)
eval input state@(State commits oracles) contract = case (contract, input) of
  (When obs timeout con con2, _)
     | currentBlockNumber > timeout → eval input state con2
     | interpretObs currentBlockNumber inputOracles state obs → eval input state con
  (Choice obs conT conF, _) → if interpretObs currentBlockNumber inputOracles state obs
    then eval input state conT
    else eval input state conF
  (Both con1 con2, _) → (st2, result, isValid1 ∨ isValid2)
    where
      result | nullContract res1 = res2
             | nullContract res2 = res1
             | True = Both res1 res2
      (st1, res1, isValid1) = eval input state con1
      (st2, res2, isValid2) = eval input st1 con2
    -- expired CommitCash
  (CommitCash _ _ _ startTimeout endTimeout _ con2, _)
     | currentBlockNumber > startTimeout ∨ currentBlockNumber > endTimeout → eval input state con2
  (CommitCash id1 pubKey value _ endTimeout con1 _, Commit id2) | id1 `eqIdentCC` id2 → let
    PendingTx [in1, in2]
      (PendingTxOut (Ledger.Value committed)
        (Just (validatorHash, DataScriptHash dataScriptHash)) DataTxOut : _)
      _ _ _ _ thisScriptHash = pendingTx
    (PendingTxIn _ (Just (_, RedeemerHash redeemerHash)) (Ledger.Value scriptValue), _) =
      orderTxIns in1 in2
    vv = evalValue state value
    isValid = vv > 0
      ∧ committed ≡ vv + scriptValue
      ∧ signedBy pubKey
      ∧ validatorHash `eqValidator` thisScriptHash
      ∧ Builtins.equalsByteString dataScriptHash redeemerHash
    in if isValid then let
        cns = (pubKey, NotRedeemed vv endTimeout)
        insertCommit :: Commit → [Commit] → [Commit]
```

```
            insertCommit commit@(_, (pubKey, NotRedeemed _ endTimeout)) commits =
                case commits of
                    [] → [commit]
                    (_, (pk, NotRedeemed _ t)) : _
                        | pk `eqPk` pubKey ∧ endTimeout < t → commit : commits
                    c : cs → c : insertCommit commit cs
            updatedState = let State committed choices = state
                in State (insertCommit (id1, cns) committed) choices
            in (updatedState, con1, True)
        else (state, contract, False)
(Pay _ _ _ _ timeout con, _)
    | currentBlockNumber > timeout → eval input state con
(Pay (IdentPay contractIdentPay) from to payValue _ con, Payment (IdentPay pid)) → let
    PendingTx [PendingTxIn _ (Just (_, RedeemerHash redeemerHash)) (Ledger.Value scriptValue)]
        (PendingTxOut (Ledger.Value change)
            (Just (validatorHash, DataScriptHash dataScriptHash)) DataTxOut : _)
            _ _ _ _ thisScriptHash = pendingTx
    pv = evalValue state payValue
    isValid = pid ≡ contractIdentPay
        ∧ pv > 0
        ∧ change ≡ scriptValue − pv
        ∧ signedBy to
        ∧ validatorHash `eqValidator` thisScriptHash
        ∧ Builtins.equalsByteString dataScriptHash redeemerHash
    in if isValid then let
            -- Discounts the Cash from an initial segment of the list of pairs.
        discountFromPairList ::
            [(IdentCC, CCStatus)]
            → Int
            → [(IdentCC, CCStatus)]
            → Maybe [(IdentCC, CCStatus)]
        discountFromPairList acc value commits = case commits of
            (ident, (party, NotRedeemed available expire)) : rest
                | currentBlockNumber ⩽ expire ∧ from `eqPk` party →
                if available > value then let
                    change = available − value
                    updatedCommit = (ident, (party, NotRedeemed change expire))
                    in discountFromPairList (updatedCommit : acc) 0 rest
                else discountFromPairList acc (value − available) rest
            commit : rest → discountFromPairList (commit : acc) value rest
            [] → if value ≡ 0 then Just acc else Nothing
        in case discountFromPairList [] pv commits of
            Just updatedCommits → let
                updatedState = State (reverse updatedCommits) oracles
                in (updatedState, con, True)
            Nothing → (state, contract, False)
    else (state, contract, False)
(RedeemCC id1 con, Redeem id2) | id1 `eqIdentCC` id2 → let
    PendingTx [PendingTxIn _ (Just (_, RedeemerHash redeemerHash)) (Ledger.Value scriptValue)]
        (PendingTxOut (Ledger.Value change)
            (Just (validatorHash, DataScriptHash dataScriptHash)) DataTxOut : _)
            _ _ _ _ thisScriptHash = pendingTx
    findAndRemove :: [(IdentCC, CCStatus)] → [(IdentCC, CCStatus)] → (Bool, State) → (Bool, State)
    findAndRemove ls resultCommits result = case ls of
        (i, (_, NotRedeemed val _)) : ls | i `eqIdentCC` id1 ∧ change ≡ scriptValue − val →
            findAndRemove ls resultCommits (True, state)
        e : ls → findAndRemove ls (e : resultCommits) result
        [] → let
            (isValid, State _ choices) = result
            in (isValid, State (reverse resultCommits) choices)
    (ok, updatedState) = findAndRemove commits [] (False, state)
```

```
    isValid = ok
        ∧ validatorHash `eqValidator` thisScriptHash
        ∧ Builtins.equalsByteString dataScriptHash redeemerHash
    in if isValid
    then (updatedState, con, True)
    else (state, contract, False)
  (_, Redeem identCC) → let
      PendingTx [PendingTxIn _ (Just (_, RedeemerHash redeemerHash)) (Ledger.Value scriptValue)]
        (PendingTxOut (Ledger.Value change)
          (Just (validatorHash, DataScriptHash dataScriptHash)) DataTxOut : _)
          _ _ _ _ thisScriptHash = pendingTx
      findAndRemoveExpired ::
        [(IdentCC, CCStatus)]
        → [(IdentCC, CCStatus)]
        → (Bool, State)
        → (Bool, State)
      findAndRemoveExpired ls resultCommits result = case ls of
        (i, (_, NotRedeemed val expire)) : ls |
          i `eqIdentCC` identCC ∧ change ≡ scriptValue − val ∧ currentBlockNumber > expire →
            findAndRemoveExpired ls resultCommits (True, state)
        e : ls → findAndRemoveExpired ls (e : resultCommits) result
        [] → let
          (isValid, State _ choices) = result
          in (isValid, State (reverse resultCommits) choices)
      (ok, updatedState) = findAndRemoveExpired commits [] (False, state)
      isValid = ok
        ∧ validatorHash `eqValidator` thisScriptHash
        ∧ Builtins.equalsByteString dataScriptHash redeemerHash
      in if isValid
      then (updatedState, contract, True)
      else (state, contract, False)
  (Null, SpendDeposit) | null commits → (state, Null, True)
  _ → (state, Null, False)
(_, contractIsValid) = validateContract (ValidatorState [] []) marloweContract
State currentCommits currentChoices = marloweState
in if contractIsValid then let
    -- record Choices from Input into State
  mergedChoices = mergeChoices (reverse inputChoices) currentChoices
  stateWithChoices = State currentCommits mergedChoices
  (newState :: State, newCont :: Contract, validated) =
    eval inputCommand stateWithChoices marloweContract
  allowTransaction = validated
    ∧ newCont `eqContract` expectedContract
    ∧ newState `eqState` expectedState
  in if allowTransaction then () else Builtins.error ()
else if null currentCommits then () else Builtins.error ()
 ∨])
```

7.6 Marlowe Wallet API

```
createContract :: (
  MonadError WalletAPIError m,
  WalletAPI m)
  ⇒ Contract
  → Int
  → m ()
createContract contract value = do
  _ ← if value ⩽ 0 then throwOtherError "Must contribute a positive value" else pure ()
  let ds = DataScript $ Ledger.lifted (Input SpendDeposit [] [], MarloweData {
    marloweContract = contract,
```

```
    marloweState = emptyState})
  let v′ = Ledger.Value value
  (payment, change) ← createPaymentWithChange v′
  let o = scriptTxOut v′ marloweValidator ds
  void $ signAndSubmit payment (o : maybeToList change)
marloweTx ::
  (Input, MarloweData)
  → (TxOut′, TxOutRef′)
  → (TxIn′ → (Int → TxOut′) → Int → m ())
  → m ()
marloweTx inputState txOut f = do
  let (TxOut _ (Ledger.Value contractValue) _, ref) = txOut
  let lifted = Ledger.lifted inputState
  let scriptIn = scriptTxIn ref marloweValidator $ Ledger.RedeemerScript lifted
  let dataScript = DataScript lifted
  let scritOut v = scriptTxOut (Ledger.Value v) marloweValidator dataScript
  f scriptIn scritOut contractValue
createRedeemer
  :: InputCommand → [OracleValue Int] → [Choice] → State → Contract → (Input, MarloweData)
createRedeemer inputCommand oracles choices expectedState expectedCont =
  let input = Input inputCommand oracles choices
      mdata = MarloweData {marloweContract = expectedCont, marloweState = expectedState}
  in (input, mdata)
commit :: (
  MonadError WalletAPIError m,
  WalletAPI m)
  ⇒ (TxOut′, TxOutRef′)
  → [OracleValue Int]
  → [Choice]
  → IdentCC
  → Int
  → State
  → Contract
  → m ()
commit txOut oracles choices identCC value expectedState expectedCont = do
  _ ← if value ⩽ 0 then throwOtherError "Must commit a positive value" else pure ()
  let redeemer = createRedeemer (Commit identCC) oracles choices expectedState expectedCont
  marloweTx redeemer txOut $ λi getOut v → do
      (payment, change) ← createPaymentWithChange (Ledger.Value value)
      void $ signAndSubmit (Set.insert i payment) (getOut (v + value) : maybeToList change)
receivePayment :: (
  MonadError WalletAPIError m,
  WalletAPI m)
  ⇒ (TxOut′, TxOutRef′)
  → [OracleValue Int]
  → [Choice]
  → IdentPay
  → Int
  → State
  → Contract
  → m ()
receivePayment txOut oracles choices identPay value expectedState expectedCont = do
  _ ← if value ⩽ 0 then throwOtherError "Must commit a positive value" else pure ()
  let redeemer = createRedeemer (Payment identPay) oracles choices expectedState expectedCont
  marloweTx redeemer txOut $ λi getOut v → do
      let out = getOut (v − value)
      oo ← ownPubKeyTxOut (Ledger.Value value)
      void $ signAndSubmit (Set.singleton i) [out, oo]
redeem :: (
  MonadError WalletAPIError m,
  WalletAPI m)
  ⇒ (TxOut′, TxOutRef′)
```

```
    → [OracleValue Int]
    → [Choice]
    → IdentCC
    → Int
    → State
    → Contract
    → m ()
redeem txOut oracles choices identCC value expectedState expectedCont = do
  _ ← if value ⩽ 0 then throwOtherError "Must commit a positive value" else pure ()
  let redeemer = createRedeemer (Redeem identCC) oracles choices expectedState expectedCont
  marloweTx redeemer txOut $ λi getOut v → do
    let out = getOut (v − value)
    oo ← ownPubKeyTxOut (Ledger.Value value)
    void $ signAndSubmit (Set.singleton i) [out, oo]
endContract :: (Monad m, WalletAPI m) ⇒ (TxOut′, TxOutRef′) → State → m ()
endContract txOut state = do
  let redeemer = createRedeemer SpendDeposit [] [] state Null
  marloweTx redeemer txOut $ λi _ v → do
    oo ← ownPubKeyTxOut (Ledger.Value v)
    void $ signAndSubmit (Set.singleton i) [oo]
```