Smart Contracts

Marlowe

Lars Brünjes

INPUT | OUTPUT

January 10 2020

Parties & Accounts

```
newtype PubKey = PubKey Text
  deriving (Eq,Ord)

type Party = PubKey

type NumAccount = Integer

data AccountId = AccountId NumAccount Party
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A party is a participant in the contract. Parties can perform actions like depositing money into an account. Marlowe also has a concept of accounts to make contract creation easier. Accounts are given by a combination of a number and a party. This party will get all remaining money at the end of the contract. Accounts are local to the contract.

The Contract Type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If  Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

The Contract Type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

`Close` is the simplest contract: It closes the contract and provides refunds to the owners of accounts that contain a positive balance. This is performed one account per step, but all accounts will be refunded in a single transaction.

The Contract Type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A payment contract `Pay a p v cont` will make a payment of value `v` from the account `a` to a payee `p`, which will be one of the contract participants or another account in the contract. Warnings will be generated if the value `v` is negative, or if there is not enough in the account to make the payment in full. In that case a partial payment (of all the money available) is made. The continuation contract is the one given in the contract: `cont`.

The Contract Type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
   deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

The conditional `If obs cont1 cont2` will continue as `cont1` or `cont2`, depending on the Boolean value of the observation `obs` when this construct is executed.

The Contract Type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

`When cases timeout cont` is the most complex constructor for contracts. It is a contract that is triggered on actions, which may or may not happen at any particular slot: What happens when various actions happen is described by the cases in the contract.
The list `cases` contains a collection of cases. Each case has the form `Case ac co` where `ac` is an action and `co` a continuation. When a particular action happens, the contract will continue as the corresponding continuation.
In order to make sure that the contract makes progress eventually, the contract will continue as `cont` once `timeout` is reached.

The Contract Type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If  Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A contract `Let id val cont` allows a contract to name a value using an identifier. In this case, the expression `val` is evaluated, and stored with the name `id`. The contract then continues as `cont`.
As well as allowing us to use abbreviations, this mechanism also means that we can capture and save volatile values that might be changing with time, e.g. the current price of oil, or the current slot number, at a particular point in the execution of the contract, to be used later on in contract execution.

The Observation Type

```
data Observation = AndObs Observation Observation
                 | OrObs Observation Observation
                 | NotObs Observation
                 | ChoseSomething ChoiceId
                 | ValueGE Value Value
                 | ValueGT Value Value
                 | ValueLT Value Value
                 | ValueLE Value Value
                 | ValueEQ Value Value
                 | TrueObs
                 | FalseObs
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Observations are Boolean value that come from combining other observations, from comparing values or — in the case of `ChoseSomething` — if a party made a choice.

The Value Type

```
data Value = AvailableMoney AccountId
           | Constant Integer
           | NegValue Value
           | AddValue Value Value
           | SubValue Value Value
           | ChoiceValue ChoiceId Value
           | SlotIntervalStart
           | SlotIntervalEnd
           | UseValue ValueId
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Values are values that can sometimes change over time — like the money available in an account or the current slot number.

The Payee Type

```
data Payee = Account AccountId
           | Party Party
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Payments can be made to in-contract accounts (constructor `Account`) or to parties (`Party` constructor).

The Case Type

```
data Case = Case Action Contract
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

The Bound Type

```haskell
data Bound = Bound Integer Integer
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice ChoiceId [Bound]
            | Notify Observation
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Marlowe distinguishes between three different types of actions (which are triggered externally, outside of the contract's control).

The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice ChoiceId [Bound]
            | Notify Observation
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A `Deposit n p v` makes a deposit of value `v` into account number `n` belonging to party `p`.

The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice ChoiceId [Bound]
            | Notify Observation
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A choice is made for a particular id with a list of bounds on the values that are acceptable. For example, `[Bound 0 0, Bound 3 5]` offers the choice of one of 0, 3, 4 and 5.

The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice ChoiceId [Bound]
            | Notify Observation
  deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

`Notify obs` notifies the contract of an observation obs that has been made. Typically this would be done by one of the parties, or one of their wallets acting automatically.

Exercises

- Write four Marlowe contracts in which Alice is supposed to first deposit 100 Lovelace into the contract. If she does not do this until Slot 5, nothing happens. If she does,
  - In the first contract, her money is paid to Bob.
  - in the second contract, her money should be paid to Bob and Charlie in equal parts,
  - in the third contract, she gets her money back in Slot 10, and
  - in the fourth contract, Bob can choose whether the money goes to himself or to Charlie. If Bob does not make a choice until Slot 10, the money goes back to Alice.
- Write a Marlowe contract in which Alice can choose an amount between 100 and 200 Lovelace and deposit it into the contract until Slot 3. If she does not do this until Slot 3, nothing happens. If she does, Bob gets the chosen amount.

Example: Simple Crowd Sourcing

```
-{-# LANGUAGE OverloadedStrings #-}
module Main where

import Language.Marlowe

main :: IO ()
main = print . pretty $ contract
    "Alice" 300 ["Bob", "Charlie", "Dora", "Eve"] 100 10

contract :: Party       -- campaign owner
         -> Integer -- funding target
         -> [Party] -- contributors
         -> Integer -- contribution
         -> Slot    -- deadline
         -> Contract
contract owner target contributors contribution deadline
    | not (enough contributors) = Close
    | otherwise                 = go [] contributors
  where
    go :: [Party] -> [Party] -> Contract
    go ps [] = check ps
    go ps qs =
        When
            [mkCase ps qs q | q <- qs]
            deadline
```

```
--mkCase :: [Party] -> [Party] -> Party -> Case
mkCase ps qs q =
    Case
        (Deposit (account q) q $ Constant contribution) $
        go (q : ps) $ filter (/= q) qs

account :: Party -> AccountId
account = AccountId 1

enough :: [Party] -> Bool
enough ps = contribution * fromIntegral (length ps) >= target

check :: [Party] -> Contract
check ps
    | enough ps = pay ps
    | otherwise = Close

pay :: [Party] -> Contract
pay []       = Close
pay (p : ps) = Pay (account p) (Party owner) (Constant contribution) $ pay ps
```

Example: Simple Crowd Sourcing

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Language.Marlowe

main :: IO ()
main = print . pretty $ contract
    "Alice" 300 ["Bob", "Charlie", "Dora", "Eve"] 100 10

contract :: Party        -- campaign owner
         -> Integer      -- funding target
         -> [Party]      -- contributors
         -> Integer      -- contribution
         -> Slot         -- deadline
         -> Contract
contract owner target contributors contribution deadline
    | not (enough contributors) = Close
    | otherwise                 = go [] contributors
  where
    go :: [Party] -> [Party] -> Contract
    go ps [] = check ps
    go ps qs =
        When
            [mkCase ps qs q | q <- qs]
            deadline

    --mkCase :: [Party] -> [Party] -> Party -> Case
    mkCase ps qs q =
        Case
            (Deposit (account q) q $ Constant contribution) $
            go (q : ps) $ filter (/= q) qs

    account :: Party -> AccountId
    account = AccountId 1

    enough :: [Party] -> Bool
    enough ps = contribution * fromIntegral (length ps) >= target

    check :: [Party] -> Contract
    check ps
        | enough ps = pay ps
        | otherwise = Close

    pay :: [Party] -> Contract
    pay []       = Close
    pay (p : ps) = Pay (account p) (Party owner) (Constant contribution) $ pay ps
```

Remark

For more complex contracts, Blockly becomes infeasible, and using the full power of Haskell makes things much more concise.

Projects

- Tasks
  - Write a Marlowe contract that simulates a Dutch Auction: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.

Projects

- Tasks
    - Write a Marlowe contract that simulates a Dutch Auction: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
    - Write a Marlowe contract that simulates a First Price Sealed Bid Auction: There is one round of (normally secret bidding, but we cannot do this in Marlowe), and the highest bidder wins.

Projects

- Tasks
  - Write a Marlowe contract that simulates a Dutch Auction: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
  - Write a Marlowe contract that simulates a First Price Sealed Bid Auction: There is one round of (normally secret bidding, but we cannot do this in Marlowe), and the highest bidder wins.
  - Write a Marlowe contract that simulates an English Auction: There is one several rounds in which bidders can increase their bids until the highest bidder wins.

Projects

- Tasks
  - Write a Marlowe contract that simulates a Dutch Auction: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
  - Write a Marlowe contract that simulates a First Price Sealed Bid Auction: There is one round of (normally secret bidding, but we cannot do this in Marlowe), and the highest bidder wins.
  - Write a Marlowe contract that simulates an English Auction: There is one several rounds in which bidders can increase their bids until the highest bidder wins.
- Remarks
  - You can model "winning" as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.

Projects

- Tasks
  - Write a Marlowe contract that simulates a Dutch Auction: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
  - Write a Marlowe contract that simulates a First Price Sealed Bid Auction: There is one round of (normally secret bidding, but we cannot do this in Marlowe), and the highest bidder wins.
  - Write a Marlowe contract that simulates an English Auction: There is one several rounds in which bidders can increase their bids until the highest bidder wins.
- Remarks
  - You can model "winning" as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.
  - Ideally, you would parameterize your solution over the list of bidders, but for simplicity, you can use just two bidders.

Projects

- Tasks
  - Write a Marlowe contract that simulates a Dutch Auction: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
  - Write a Marlowe contract that simulates a First Price Sealed Bid Auction: There is one round of (normally secret bidding, but we cannot do this in Marlowe), and the highest bidder wins.
  - Write a Marlowe contract that simulates an English Auction: There is one several rounds in which bidders can increase their bids until the highest bidder wins.
- Remarks
  - You can model "winning" as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.
  - Ideally, you would parameterize your solution over the list of bidders, but for simplicity, you can use just two bidders.
  - Make sure that no bidder can make a bid without being forced to actually pay if he wins the auction.

Projects

- Tasks
  - Write a Marlowe contract that simulates a Dutch Auction: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
  - Write a Marlowe contract that simulates a First Price Sealed Bid Auction: There is one round of (normally secret bidding, but we cannot do this in Marlowe), and the highest bidder wins.
  - Write a Marlowe contract that simulates an English Auction: There is one several rounds in which bidders can increase their bids until the highest bidder wins.
- Remarks
  - You can model "winning" as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.
  - Ideally, you would parameterize your solution over the list of bidders, but for simplicity, you can use just two bidders.
  - Make sure that no bidder can make a bid without being forced to actually pay if he wins the auction.
  - Make also sure that everybody else gets back their money in the end.

INPUT | OUTPUT