Smart Contracts

Marlowe

Lars Brünjes

INPUT | OUTPUT

January 22 2021

Parties & Accounts

```
data Party = PK PubKeyHash | Role TokenName

type AccountId = Party
```

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 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 Token (Value Observation) Contract
              | If  Observation Contract Contract
              | When [Case Contract] Timeout Contract
              | Let ValueId (Value Observation) Contract
              | Assert Observation Contract
```

The Contract Type

```
data Contract = Close
              | Pay AccountId Payee Token (Value Observation) Contract
              | If  Observation Contract Contract
              | When [Case Contract] Timeout Contract
              | Let ValueId (Value Observation) Contract
              | Assert Observation Contract
```

`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 Token (Value Observation) Contract
              | If Observation Contract Contract
              | When [Case Contract] Timeout Contract
              | Let ValueId (Value Observation) Contract
              | Assert Observation Contract
```

A payment contract `Pay a p v t cont` will make a payment of value `v` in token `t` 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 Token (Value Observation) Contract
              | If  Observation Contract Contract
              | When [Case Contract] Timeout Contract
              | Let ValueId (Value Observation) Contract
              | Assert Observation Contract
```

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 Token (Value Observation) Contract
              | If Observation Contract Contract
              | When [Case Contract] Timeout Contract
              | Let ValueId (Value Observation) Contract
              | Assert Observation Contract
```

`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 Token (Value Observation) Contract
              | If Observation Contract Contract
              | When [Case Contract] Timeout Contract
              | Let ValueId (Value Observation) Contract
              | Assert Observation Contract
```

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 Contract Type

```
data Contract = Close
              | Pay AccountId Payee Token (Value Observation) Contract
              | If  Observation Contract Contract
              | When [Case Contract] Timeout Contract
              | Let ValueId (Value Observation) Contract
              | Assert Observation Contract
```

A contract `Assert obs cont` will behave like its continuation `cont`, but it will issue a warning if observation `obs` is false. It can be used to ensure that a property holds in any given point of the contract, since static analysis will fail if any execution causes an `Assert` to be false.

The Observation Type

```
data Observation = AndObs Observation Observation
                 | OrObs Observation Observation
                 | NotObs Observation
                 | ChoseSomething ChoiceId
                 | ValueGE (Value Observation) (Value Observation)
                 | ValueGT (Value Observation) (Value Observation)
                 | ValueLT (Value Observation) (Value Observation)
                 | ValueLE (Value Observation) (Value Observation)
                 | ValueEQ (Value Observation) (Value Observation)
                 | TrueObs
                 | FalseObs
```

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 a = AvailableMoney AccountId Token
             | Constant Integer
             | NegValue (Value a)
             | AddValue (Value a) (Value a)
             | SubValue (Value a) (Value a)
             | MulValue (Value a) (Value a)
             | Scale Rational (Value a)
             | ChoiceValue ChoiceId
             | SlotIntervalStart
             | SlotIntervalEnd
             | UseValue ValueId
             | Cond a (Value a) (Value a)
```

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
```

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

The Case Type

```
data Case a = Case Action a
```

The Bound Type

```
data Bound = Bound Integer Integer
```

The Action Type

```
data Action = Deposit AccountId Party Token (Value Observation)
            | Choice ChoiceId [Bound]
            | Notify Observation
```

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 Token (Value Observation)
            | Choice ChoiceId [Bound]
            | Notify Observation
```

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

The Action Type

```
data Action = Deposit AccountId Party Token (Value Observation)
            | Choice ChoiceId [Bound]
            | Notify Observation
```

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 Token (Value Observation)
            | Choice ChoiceId [Bound]
            | Notify Observation
```

`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 Examples.Crowd
    ( crowd
    ) where

import Data.List        ( foldl ')
import Language.Marlowe

crowd :: Integer   −− ^ campaign goal
      −> Integer   −− ^ individual contribution
      −> Party     −− ^ campaign owner
      −> [Party]   −− ^ contributors
      −> Slot      −− ^ deadline
      −> Contract
crowd goal c owner contributors deadline
    = go [] contributors
```

```
  where
    go :: [Party] −> [Party] −> Contract
    go ys ns =
        When
            [Case (Deposit n n ada $ Constant c) $
                go (n : ys) $ filter (/= n) ns | n <− ns]
            deadline $ settle ys

    settle :: [Party] −> Contract
    settle ys
        | fromIntegral (length ys) ∗ c >= goal = foldl ' pay Close ys
        | otherwise                           = Close

    pay :: Contract −> Party −> Contract
```

Example: Simple Crowd Sourcing

```
{-# LANGUAGE OverloadedStrings #-}

module Examples.Crowd
    ( crowd
    ) where

import Data.List          ( foldl')
import Language.Marlowe

crowd :: Integer   -- ^ campaign goal
      -> Integer   -- ^ individual contribution
      -> Party     -- ^ campaign owner
      -> [Party]   -- ^ contributors
      -> Slot      -- ^ deadline
      -> Contract
crowd goal c owner contributors deadline
    = go [] contributors
  where
    go :: [Party] -> [Party] -> Contract
    go ys ns =
        When
            [Case (Deposit n n ada $ Constant c) $
                go (n : ys) $ filter (/= n) ns | n <- ns]
            deadline $ settle ys

    settle :: [Party] -> Contract
    settle ys
        | fromIntegral (length ys) * c >= goal = foldl' pay Close ys
        | otherwise                            = Close

    pay :: Contract -> Party -> Contract
```

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: For a number of rounds 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: For a number of rounds bidders can increase their bids until the highest bidder wins.
- Remarks
  - The auctioned item should be an arbitrary token, which the seller has to deposit into the contract in the beginning. When a bidder wins the auction, the bid goes to the seller, and the winner gets the token.

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: For a number of rounds bidders can increase their bids until the highest bidder wins.
- Remarks
  - The auctioned item should be an arbitrary token, which the seller has to deposit into the contract in the beginning. When a bidder wins the auction, the bid goes to the seller, and the winner gets the token.
  - 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: For a number of rounds bidders can increase their bids until the highest bidder wins.
- Remarks
  - The auctioned item should be an arbitrary token, which the seller has to deposit into the contract in the beginning. When a bidder wins the auction, the bid goes to the seller, and the winner gets the token.
  - 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: For a number of rounds bidders can increase their bids until the highest bidder wins.
- Remarks
  - The auctioned item should be an arbitrary token, which the seller has to deposit into the contract in the beginning. When a bidder wins the auction, the bid goes to the seller, and the winner gets the token.
  - 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. In particular, the seller must get his token back if the auction fails.

INPUT | OUTPUT