```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DefaultSignatures #-}
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE StrictData #-}
{-# LANGUAGE TemplateHaskell #-}
{-# OPTIONS -fplugin=Language.Plutus.CoreToPLC.Plugin -fplugin-opt Language.Plutus.CoreToPLC.P
{-# OPTIONS_GHC − Wno − incomplete − uni − patterns#−}
```

```
module Language.Marlowe.Compiler where
import Control.Applicative            (Applicative (..))
import Control.Monad                  (Monad (..))
import Control.Monad.Error.Class      (MonadError (..))
import GHC.Generics                   (Generic)
import qualified Data.List as List
import qualified Data.Set                           as Set
import Data.Set (Set)
import qualified Data.Map.Strict as Map
import Data.Map.Strict (Map)
import qualified Language.Plutus.CoreToPLC.Builtins as Builtins
import Language.Plutus.Runtime
import Language.Plutus.TH                           (plutus)
import Wallet.API                     (EventTrigger (..), Range (..), WalletAPI (..), WalletAPIError, otherE
                                        pubKey, signAndSubmit)
import Wallet.UTXO                                  (Address′, DataScript (..), TxOutRef′, Validator (..), script′
                                                      scriptTxOut, applyScript, emptyValidator, unitData)
import qualified Wallet.UTXO                        as UTXO
import qualified Language.Plutus.Runtime.TH as TH
import Language.Plutus.Lift           (LiftPlc (..), TypeablePlc (..))
import Prelude                                      (Int, Bool (..), Num (..), Show (..), Read (..), Ord (..), Eq (
          fromIntegral, succ, sum, ($), (< $ >), (++), otherwise, Maybe (..))
```

1 Marlowe

Apparently, Plutus doesn't support complex recursive data types yet.

```
data Contract = Null
   | CommitCash IdentCC PubKey Value Timeout Timeout {-Contract Contract -}
   | Pay IdentPay Person Person Value Timeout {-Contract -}
     deriving (Eq, Generic)
```

Assumptions

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

```
example = CommitCash (IdentCC 1) (PubKey 1) (Value 100) (Block 200) (Block 256)
  (Pay (IdentPay 1) (PubKey 1) (PubKey 2) (Value 100) (RedeemCC (IdentCC 1) Null))
  Null
```

2 Questions

Q: Should we put together the first CommitCash with the Contract setup? Contract setup would still require some money.

Q: Should we be able to return excess money in the contract (money not accounted for). To whom? We could use excess money to ensure a contract has money on it, and then return to the creator of the contract when it becomes Null.

Q: There is a risk someone will put a continuation of a Marlowe contract without getting the previous continuation as input. Can we detect this and allow for refund?

Q: What happens on a FailedPay? Should we still pay what we can?

Q: What is signed in a transaction?

Q: How to distinguish different instances of contracts? Is it a thing? Maybe we need to add a sort of identifier of a contract.

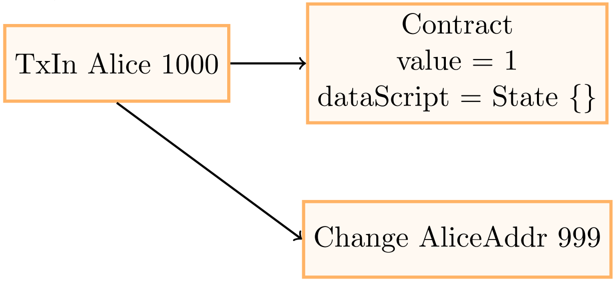
- Whole validator script (read Contract script) on every spending tx.
- No offchain messages (‘internal messages‘ in Ethereum)? How to call a function? Answer: currently only via transaction

3 Contract Initialization

This can be done in 2 ways.

3.1 Initialization by depositing Ada to a new contract

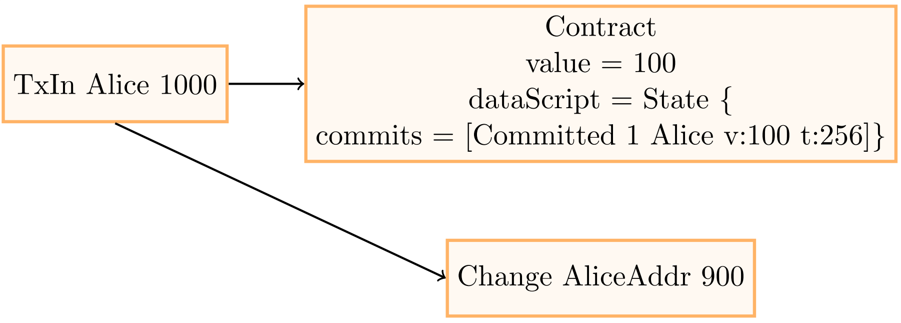
Just pay 1 Ada to a contract so that it becomes a part of UTXO.



Considerations Someone need to spend this 1 Ada, otherwise all Marlowe contracts will be in UTXO. We can allow anyone to spend this value, so it’ll become a part of a block reward. ???

3.2 Initialization by CommitCash

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



4 Semantics

Contract execution is a chain of transactions, where contract state is passed through *dataScript*, and actions/inputs are passed as a *redeemer* script and TxIns/TxOuts

Validation Script = marlowe interpreter + possibly encoded address of a contract owner for initial deposit refund

This would change script address for every contract owner. This could be a desired or not desired property. Discuss.

redeemer script = action/input, i.e. CommitCash val timeout, Choice 1, OracleValue ”oil” 20

pendingTx

dataScript = Contract + State

This implies that remaining Contract and its State are publicly visible. Discuss.

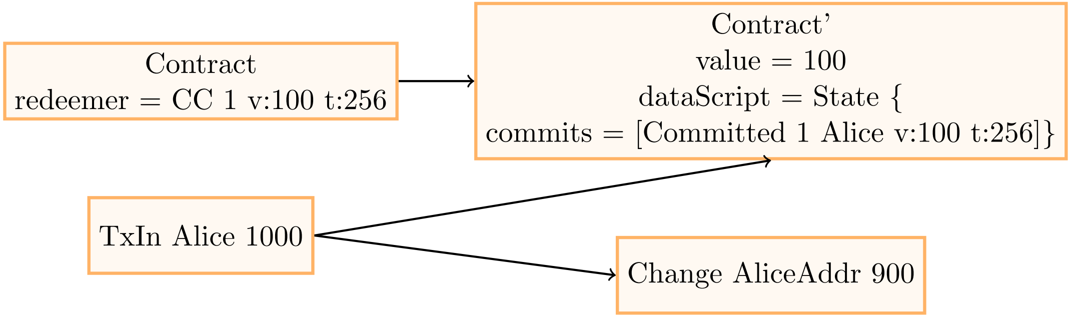
4.1 Null

Possibly allow redeem of cash spent by mistake on this address? How?

If we have all chain of txs of a contract we could allow redeems of mistakenly put money, and that would allow a contract creator to withdraw the contract initialization payment. 3

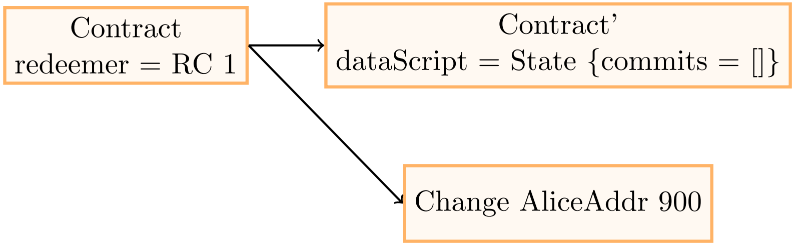
4.2 CommitCash

Alice has 1000 ADA in AliceUTXO.



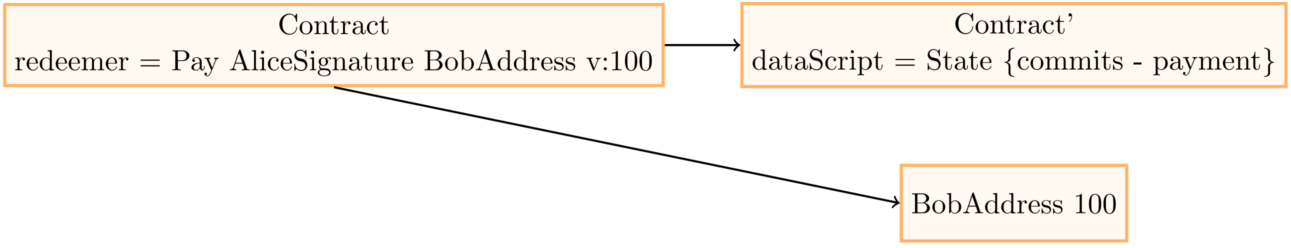
4.3 RedeemCC

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



4.4 Pay

Alice pays 100 ADA to Bob.


Contract
redeemer = Pay AliceSignature BobAddress v:100
Contract'
dataScript = State {commits - payment}
BobAddress 100

5 Types and Data Representation

```
type Timeout = Height
type Cash = Value
type Person = PubKey
  -- contractPlcCode = (plutus[CommitCash (IdentCC 1) (PubKey 1) 123 100 200 Null Null])
  -- 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, Generic)
instance LiftPlc IdentCC
instance TypeablePlc IdentCC
newtype IdentChoice = IdentChoice {unIdentChoice :: Int}
  deriving (Eq, Ord, Generic)
instance LiftPlc IdentChoice
instance TypeablePlc IdentChoice
newtype IdentPay = IdentPay Int
  deriving (Eq, Ord, Generic)
instance LiftPlc IdentPay
instance TypeablePlc IdentPay
  -- A cash commitment is made by a person, for a particular amount and timeout.
data CC = CC IdentCC Person Cash Timeout
  deriving (Eq, Ord, Generic)
instance LiftPlc CC
instance TypeablePlc CC
  -- A cash redemption is made by a person, for a particular amount.
data RC = RC IdentCC Person Cash
  deriving (Eq, Ord, Generic)
instance LiftPlc RC
instance TypeablePlc RC
data Input = Input {
  cc :: [CC],
  rc :: [RC]
    -- rp :: Map.Map (IdentPay, Person) Cash
  } deriving (Generic)
instance LiftPlc Input
instance TypeablePlc Input
emptyInput :: Input
  -- emptyInput = Input Set.empty Set.empty Map.empty
emptyInput = Input [] []
data State = State {
  stateCommitted :: [(IdentCC, CCStatus)]
  } deriving (Eq, Ord, Generic)
instance LiftPlc State
instance TypeablePlc State
emptyState :: State
emptyState = State {stateCommitted = []}
data MarloweData = MarloweData {
  marloweState :: State,
  marloweContract :: Contract
  } deriving (Generic)
instance LiftPlc MarloweData
instance TypeablePlc MarloweData
type ConcreteChoice = Int
type CCStatus = (Person, CCRedeemStatus)
data CCRedeemStatus = NotRedeemed Cash Timeout | ManuallyRedeemed
  deriving (Eq, Ord, Generic)
instance LiftPlc CCRedeemStatus
instance TypeablePlc CCRedeemStatus
instance LiftPlc Contract
instance TypeablePlc Contract
```

6 Marlowe Interpreter and Helpers

```
marloweValidator = Validator result where
  result = UTXO.fromPlcCode $ (plutus [| λ(redeemer :: ()) MarloweData {..} (pendingTx :: PendingTx Va
      True
     |])
createContract :: (
  MonadError WalletAPIError m,
  WalletAPI m)
  ⇒ Contract
  → Value
  → m ()
createContract contract value = do
  _ ← if value ⩽ 0 then otherError "Must contribute a positive value" else pure ()
  let ds = DataScript $ UTXO.lifted (MarloweData {marloweContract = contract, marloweState = emptyS
  let v′ = UTXO.Value $ fromIntegral value
  (payment, change) ← createPaymentWithChange v′
  let o = scriptTxOut v′ marloweValidator ds

  signAndSubmit payment [o, change]
endContract :: (Monad m, WalletAPI m) ⇒ Contract → TxOutRef′ → UTXO.Value → m ()
endContract contract ref val = do
  oo ← payToPublicKey val
  let scr = marloweValidator
    i = scriptTxIn ref scr UTXO.unitRedeemer
  signAndSubmit (Set.singleton i) [oo]
```