1 Introduction

This file contains implementation of Marlowe Validation Script... Extended UTXO model...
TODO: write decent intro...

2 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
- Currently the contracts are not secure, because we do not validate that provided continuation contract is indeed same as expected. This will be addressed when required mechanisms are implemented in Plutus.

2.1 Imports

```
  {-# LANGUAGE DataKinds #-}
  {-# LANGUAGE DefaultSignatures #-}
  {-# LANGUAGE DeriveAnyClass #-}
  {-# LANGUAGE DeriveGeneric #-}
  {-# LANGUAGE DerivingStrategies #-}
  {-# LANGUAGE OverloadedStrings #-}
  {-# LANGUAGE RecordWildCards #-}
  {-# 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 GHC.Generics                 (Generic)
import qualified Data.Set                     as Set
import Language.Plutus.Runtime      hiding (Value)
import qualified Language.Plutus.Runtime      as Plutus
import Language.PlutusTx.TH         (plutus)
import Wallet.API                   (EventTrigger (..)
                                    , Range (..)
                                    , WalletAPI (..)
                                    , WalletAPIError
                                    , otherError
                                    , pubKey
                                    , signAndSubmit
                                    , payToPubKey
                                    , ownPubKeyTxOut
                                    )
import Wallet.UTXO                  (Address′
                                    , DataScript (..)
                                    , TxOutRef′
                                    , TxOut′
                                    , TxOut (..)
                                    , Validator (..)
                                    , scriptTxIn
                                    , scriptTxOut
                                    , applyScript
                                    , emptyValidator
                                    , unitData
                                    , txOutValue
```

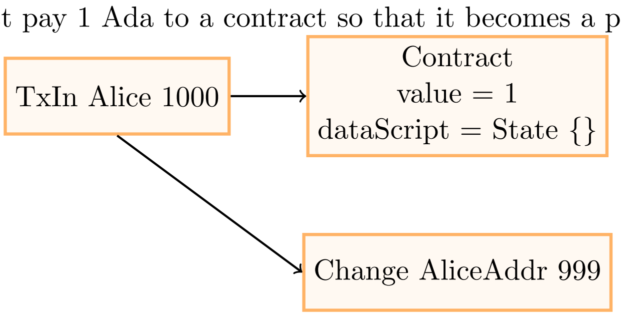
```
                                   )
import qualified Wallet.UTXO                  as UTXO

import qualified Language.Plutus.Runtime.TH as TH
import qualified Language.PlutusTx.Builtins   as Builtins
import Language.PlutusTx.Lift          (makeLift)
```

3 Contract Initialization

This can be done in 2 ways.

3.1 Initialization by depositing Ada to a new contract

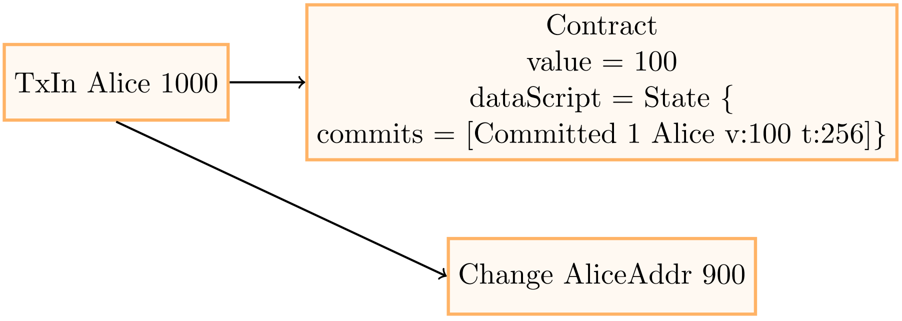
Just pay 1 Ada to a contract so that it becomes a part of UTXO.



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

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 *Data Script*, and actions/inputs are passed as a *Redeemer Script* and TxIns/TxOuts

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

Redeemer Script = input, i.e. *Commit*, *Redeem*, *Pay*, and *SpendDeposit*

Data Script = *Remaining Contract* + *State*

State = Set of *Commits* + Set of *Choices*

This implies that *Remaining Contract* and its *State* are publicly visible.

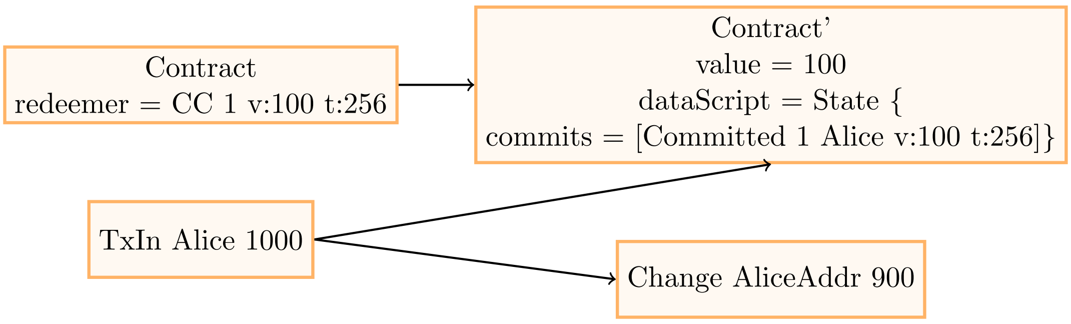
4.1 Null

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

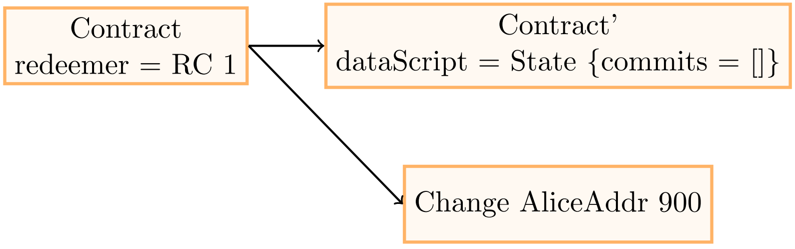
4.2 CommitCash

Alice has 1000 Ada in AliceUTXO.



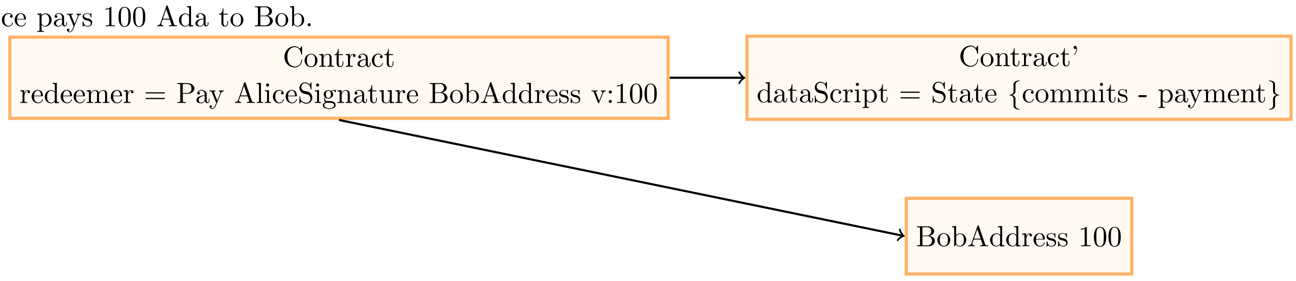
4.3 RedeemCC

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 = Int
type Cash = Int
type Person = PubKey
```

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

5.2 Input

Input is passed in *Redeemer Script*

```
data InputCommand = Commit IdentCC
    | Payment IdentPay
    | Redeem IdentCC
    | SpendDeposit
  deriving (Generic)
makeLift '' InputCommand

data Input = Input InputCommand [OracleValue Int] [Choice]
  deriving Generic
makeLift '' Input
```

5.3 Contract State

```
data State = State {
  stateCommitted :: [Commit],
    -- ^commitsMUSTbesortedbyexpirationtime, ascending
  stateChoices :: [Choice]
  } deriving (Eq, Ord, Generic)
makeLift '' State

emptyState :: State
emptyState = State {stateCommitted = [], stateChoices = []}
```

5.4 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
    deriving (Eq, Generic)
makeLift '' Value
```

5.5 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, Generic)
makeLift '' Observation
```

5.6 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, Generic)
makeLift '' Contract
```

5.7 Marlowe Data Script

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

```
data MarloweData = MarloweData {
  marloweState :: State,
  marloweContract :: Contract
  } deriving (Generic)
makeLift '' MarloweData
```

6 Marlowe Validator Script

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

```
marloweValidator :: Validator
marloweValidator = Validator result where
  result = UTXO.fromPlcCode $$ (plutus [∨ λ
    (Input inputCommand inputOracles inputChoices :: Input)
    (MarloweData {..} :: MarloweData)
    (pendingTx@PendingTx {pendingTxBlockHeight} :: PendingTx ValidatorHash) → let
```

6.1 Marlowe Validator Prelude

```
eqPk :: PubKey → PubKey → Bool
eqPk = $$(TH.eqPubKey)
eqIdentCC :: IdentCC → IdentCC → Bool
eqIdentCC (IdentCC a) (IdentCC b) = a ≡ b
eqValidator :: ValidatorHash → ValidatorHash → Bool
eqValidator = $$(TH.eqValidator)
¬ :: Bool → Bool
¬ = $$(TH.¬)
infixr 3 ∧
(∧) :: Bool → Bool → Bool
(∧) = $$(TH.and)
infixr 3 ∨
(∨) :: Bool → Bool → Bool
(∨) = $$(TH.or)
signedBy :: PubKey → Bool
signedBy = $$(TH.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 = $$(TH.isJust)

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

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

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@(Plutus.Height blockNumber) oracles = case oracles of
    OracleValue (Signed (pk, (Plutus.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)
```

6.2 Contract Validation

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

```
validateContract :: [IdentCC] → [IdentPay] → Contract → Bool
validateContract ccIds payIds contract = case contract of
    Null → True
    CommitCash ident _ _ _ _ c1 c2 → notInCCs ident ∧
      let ids = ident : ccIds
      in validateContract ids payIds c1 ∧ validateContract ids payIds c2
    RedeemCC ident c → notInCCs ident ∧ validateContract (ident : ccIds) payIds c
    Pay ident _ _ _ _ c → notInPays ident ∧ validateContract ccIds (ident : payIds) c
    Both c1 c2 → validateContract ccIds payIds c1 ∧ validateContract ccIds payIds c2
    Choice _ c1 c2 → validateContract ccIds payIds c1 ∧ validateContract ccIds payIds c2
    When _ _ c1 c2 → validateContract ccIds payIds c1 ∧ validateContract ccIds payIds c2
  where
    notInCCs :: IdentCC → Bool
    notInCCs = notElem eqIdentCC ccIds
    notInPays :: IdentPay → Bool
    notInPays = notElem (λ(IdentPay a) (IdentPay b) → a ≡ b) payIds
```

6.3 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
```

6.4 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 Plutus.Height blockNumber = pendingTxBlockHeight in blockNumber
```

6.5 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 (Plutus.Value committed) (Just (validatorHash, _)) DataTxOut, _]
    _ _ _ _ thisScriptHash = pendingTx
  (PendingTxIn _ _ (Plutus.Value scriptValue),
    PendingTxIn _ _ (Plutus.Value commitValue)) = orderTxIns in1 in2
  vv = evalValue state value
  isValid = vv > 0
     ∧ committed ≡ vv + scriptValue
     ∧ signedBy pubKey
     ∧ validatorHash ‘eqValidator‘ thisScriptHash
  in if isValid then let
       cns = (pubKey, NotRedeemed commitValue 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 _ _ (Plutus.Value scriptValue)]
    [PendingTxOut (Plutus.Value change) (Just (validatorHash, _)) DataTxOut, _]
    _ _ _ _ thisScriptHash = pendingTx
  pv = evalValue state payValue
  isValid = pid ≡ contractIdentPay
     ∧ pv > 0
     ∧ change ≡ scriptValue − pv
     ∧ signedBy to
     ∧ validatorHash ‘eqValidator‘ thisScriptHash
  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 _ _ (Plutus.Value scriptValue)]
            (PendingTxOut (Plutus.Value change) (Just (validatorHash, _)) 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
        in if isValid
        then (updatedState, con, True)
        else (state, contract, False)
    (_, Redeem identCC) → let
            PendingTx [PendingTxIn _ _ (Plutus.Value scriptValue)]
                (PendingTxOut (Plutus.Value change) (Just (validatorHash, _)) 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
            in if isValid
            then (updatedState, contract, True)
            else (state, contract, False)
    (Null, SpendDeposit) | null commits → (state, Null, True)
    _ → (state, Null, False)
contractIsValid = validateContract [ ] [ ] marloweContract
    -- record Choices from Input into State
stateWithChoices = let
    State commits choices = marloweState
    in State commits (mergeChoices inputChoices choices)
(_ :: State, _ :: Contract, allowTransaction) = eval inputCommand stateWithChoices marloweContract
    -- if a contract is not valid we allow a contract creator to spend its initial deposit
    -- otherwise, if the contract IS valid we check the contract allows this transaction
in if ¬ contractIsValid ∨ allowTransaction then ()
else Builtins.error ()
```

```
∨])
```

6.6 Helpers for creating Transactions on Mockchain

```
createContract :: (
  MonadError WalletAPIError m,
  WalletAPI m)
  ⇒ Contract
  → Int
  → 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 = emptyState}
  let v′ = UTXO.Value value
  (payment, change) ← createPaymentWithChange v′
  let o = scriptTxOut v′ marloweValidator ds
  void $ signAndSubmit payment [o, change]
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 otherError "Must commit a positive value" else pure ()
  let (TxOut _ (UTXO.Value contractValue) _, ref) = txOut
  let input = Input (Commit identCC) oracles choices
  let i = scriptTxIn ref marloweValidator $ UTXO.Redeemer (UTXO.lifted input)
  let ds = DataScript $ UTXO.lifted MarloweData {
    marloweContract = expectedCont,
    marloweState = expectedState
    }
  (payment, change) ← createPaymentWithChange (UTXO.Value value)
  let o = scriptTxOut (UTXO.Value $ value + contractValue) marloweValidator ds
  void $ signAndSubmit (Set.insert i payment) [o, 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 otherError "Must commit a positive value" else pure ()
  let (TxOut _ (UTXO.Value contractValue) _, ref) = txOut
  let input = Input (Payment identPay) oracles choices
  let i = scriptTxIn ref marloweValidator (UTXO.Redeemer $ UTXO.lifted input)
  let ds = DataScript $ UTXO.lifted MarloweData {
    marloweContract = expectedCont,
    marloweState = expectedState
```

```
        }
  let o = scriptTxOut (UTXO.Value $ contractValue − value) marloweValidator ds
  oo ← ownPubKeyTxOut (UTXO.Value value)
  void $ signAndSubmit (Set.singleton i) [o, 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 otherError "Must commit a positive value" else pure ()
  let (TxOut _ (UTXO.Value contractValue) _, ref) = txOut
  let input = Input (Redeem identCC) oracles choices
  let i = scriptTxIn ref marloweValidator (UTXO.Redeemer $ UTXO.lifted input)
  let ds = DataScript $ UTXO.lifted MarloweData {
     marloweContract = expectedCont,
     marloweState = expectedState
        }
  let o = scriptTxOut (UTXO.Value $ contractValue − value) marloweValidator ds
  oo ← ownPubKeyTxOut (UTXO.Value value)
  void $ signAndSubmit (Set.singleton i) [o, oo]
endContract :: (Monad m, WalletAPI m) ⇒ (TxOut′, TxOutRef′) → m ()
endContract (TxOut _ val _, ref) = do
  oo ← ownPubKeyTxOut val
  let scr = marloweValidator
  let input = Input SpendDeposit [] []
     i   = scriptTxIn ref scr $ UTXO.Redeemer $ UTXO.lifted input
  void $ signAndSubmit (Set.singleton i) [oo]
```