

---

# **Plutus Pioneer Program Lecture Notes**

**Transcribed by @PoolChess (Twitter)**

**Sep 12, 2021**





# PLUTUS PIONEER PROGRAM

<b>1</b>	<b>Week 01 - English Auction</b>	<b>3</b>
1.1	Welcome . . . . .	3
1.2	The (E)UTxO Model . . . . .	3
1.3	Running an example auction contract on a local Playground . . . . .	10
1.4	To the Playground . . . . .	17
<b>2</b>	<b>Week 02 - Validation</b>	<b>31</b>
2.1	Before We Start . . . . .	31
2.2	Introduction . . . . .	32
2.3	PlutusTx.Data . . . . .	33
2.4	Plutus Validator . . . . .	35
2.5	Summary . . . . .	56
<b>3</b>	<b>Week 03 - Script Context</b>	<b>57</b>
3.1	Before We Start . . . . .	57
3.2	Recap . . . . .	57
3.3	ScriptContext . . . . .	58
3.4	Example - Vesting . . . . .	63
3.5	Example 2 - Parameterized Contract . . . . .	77
<b>4</b>	<b>Week 04 - Monads</b>	<b>87</b>
4.1	Overview . . . . .	87
4.2	Monads . . . . .	88
4.3	Plutus Monads . . . . .	104
<b>5</b>	<b>Week 05 - Native Tokens</b>	<b>117</b>
5.1	Overview . . . . .	117
5.2	Value . . . . .	117
5.3	Minting Policies . . . . .	120
5.4	Example 1 - Free . . . . .	122
5.5	Example 2 - Signed . . . . .	129
5.6	NFTs . . . . .	133
<b>6</b>	<b>Week 06 - Oracles</b>	<b>141</b>
6.1	Overview . . . . .	141
6.2	Oracle Core . . . . .	146
6.3	Swap Validation . . . . .	158
6.4	Funds Module . . . . .	168
6.5	Testing . . . . .	169
6.6	Plutus Application Backend . . . . .	176

<b>7</b>	<b>Week 07 - State Machines</b>	<b>189</b>
7.1	Introduction . . . . .	189
7.2	Code Example 1 . . . . .	195
7.3	Code Example 2 . . . . .	213
7.4	Conclusion . . . . .	225
<b>8</b>	<b>Week 08 - Property Based Testing</b>	<b>227</b>
8.1	Token Sale . . . . .	227
8.2	Unit Testing . . . . .	239
8.3	Optics and Lenses . . . . .	242
8.4	Property Based Testing . . . . .	247
<b>9</b>	<b>Week 09 - Marlowe</b>	<b>267</b>
9.1	Overview . . . . .	267
9.2	Lecture by Prof. Simon Thompson . . . . .	267
9.3	Lecture by Alex Nemish . . . . .	294
9.4	Playing in the Playground . . . . .	311
<b>10</b>	<b>Week 10 - Uniswap</b>	<b>339</b>
10.1	What is Uniswap . . . . .	339
10.2	Uniswap in Plutus . . . . .	346
10.3	Trying it Out . . . . .	378
<b>11</b>	<b>AWS Node Setup</b>	<b>389</b>
11.1	Setup the IOHK Cache . . . . .	389
11.2	Install Nix . . . . .	389
11.3	Download the Cardano Node . . . . .	390
11.4	Build the node . . . . .	390
11.5	Start the node . . . . .	390
11.6	Setup some environment variables . . . . .	390
<b>12</b>	<b>Wallets and Funds</b>	<b>393</b>
12.1	Some Helper Scripts . . . . .	393
<b>13</b>	<b>AlwaysSucceeds Script</b>	<b>395</b>
13.1	Pay to the Script . . . . .	395
13.2	Unlock the Funds in the Script . . . . .	397
<b>14</b>	<b>AlwaysFails Script</b>	<b>401</b>
14.1	Sending . . . . .	401
14.2	Grabbing . . . . .	402
<b>15</b>	<b>HelloWorld Script</b>	<b>403</b>
15.1	Compiling Plutus Scripts . . . . .	403
<b>16</b>	<b>HelloWorld, ByteStrings and Redeemer</b>	<b>409</b>
<b>17</b>	<b>Minting Tokens</b>	<b>413</b>
17.1	Create Policy . . . . .	413
17.2	Fund a Minting Wallet . . . . .	413
17.3	Mint Tokens . . . . .	414
<b>18</b>	<b>Install the Cardano node</b>	<b>417</b>
18.1	Mount the data volume . . . . .	417
18.2	Setup some environment variables . . . . .	417
18.3	Install some dependencies . . . . .	417

18.4	Install Cabal . . . . .	418
18.5	Install GHC . . . . .	418
18.6	Install libsodium . . . . .	418
18.7	Build the Cardano node . . . . .	418
18.8	Copy the binaries . . . . .	419
18.9	Get the config files . . . . .	419



These are my notes from the [series of videos by Lars Brünjes](#) for the Plutus Pioneers Program. I have also documented my journey through the Alonzo White testnet with instructions on how to run a node and how to compile and deploy Plutus code.

---

**Note:** If you find these notes useful and would like give support, please consider delegating some ADA to [CHESS Pool](#).

---

The source code for the lectures can be found [here](#).

To stay up-to-date with changes to these notes, you can follow me on Twitter @PoolChess. They are currently being updated for the second iteration of the program.



## WEEK 01 - ENGLISH AUCTION

---

**Note:** This is a written version of [Lecture #1 - Iteration #2](#).

It covers an introduction to Plutus, the (E)UTxO model (and how it compares to other models), and concludes with an example English Auction managed with a Plutus smart contract running on the Plutus Playground.

These notes have been updated to reflect the changes in iteration two of the program.

The Plutus commit used in these notes is `ea0ca4e9f9821a9dbfc5255fa0f42b6f2b3887c4`.

---

### 1.1 Welcome

Learning Plutus isn't easy, and there are a number of reasons for that.

1. Plutus uses the (E)UTxO model. This is different and less intuitive than the Ethereum method for creating smart contracts. It has a lot of advantages, but it requires a new way of thinking about smart contracts. And that's before we even start with the language itself.
2. Plutus is new and still under rapid development.
3. Tooling is not ideal, yet. So, experienced Haskell developers will notice that the experience with Plutus is not as pleasant, for example, when trying to access documentation or get syntax hints from the REPL. It can also be a challenge to build Plutus in the first place. The easiest way currently is to use Nix. The Plutus team is working on providing a Docker image, which will help.
4. Plutus is Haskell, more or less, which can have a tough learning curve for those coming from an imperative programming background.
5. Plutus is brand new and this means that there are not many help resources available, such as StackOverflow posts.

### 1.2 The (E)UTxO Model

#### 1.2.1 Overview

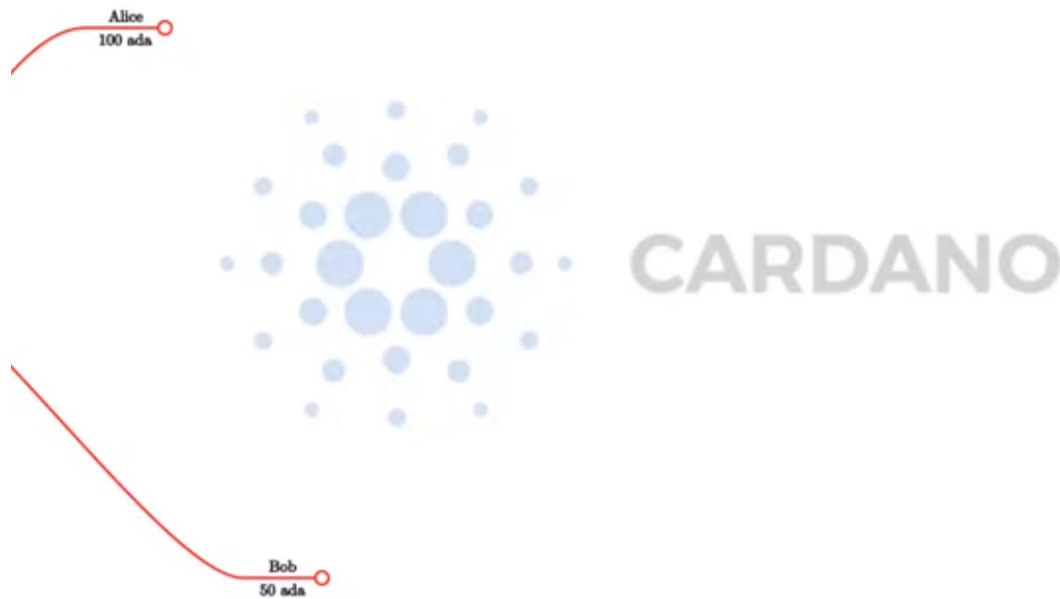
One of the most important things you need to understand in order to write Plutus smart contracts is the accounting model that Cardano uses, the Extended Unspent Transaction Output model.

The UTxO model, without the (E) is the one that was introduced by Bitcoin, but there are other models. Ethereum for example, uses the so-called account-based model, which is what you are used to from normal banking, where everybody has an account, each account has a balance and if you transfer money from one account to another then the balances get updated accordingly.

That is not how the UTxO model works.

Unspent transaction outputs are exactly what the name says. They are transaction outputs from previous transactions that have happened on the blockchain that have not yet been spent.

Let's look at an example where we have two such UTxOs. One belonging to Alice of 100 Ada, and another belonging to Bob of 50 Ada.



Alice wants to send 10 ADA to Bob, so she needs to create a transaction.

---

**Note:** A transaction is something that contains an arbitrary number of inputs and an arbitrary number of outputs. The effect of a transaction is to consume inputs and produce new outputs.

---

The important thing is that you can only ever use complete UTxOs as input. Alice cannot simply split her existing 100 ADA into a 90 and a 10, she has to use the full 100 ADA as the input to a transaction.

Once consumed by the transaction, Alice's input is no longer a UTxO (an unspent transaction). It will have been spent as an input to Tx 1. So, she needs to create outputs for her transaction.

She wants to pay 10 ADA to Bob, so one output will be 10 ADA (to Bob). She then wants her change back, so she creates a second output of 90 ADA (to herself). The full UTxO of 100 ADA has been spent, with Bob receiving a new transaction of 10 ADA, and Alice receiving the change of 90 ADA.

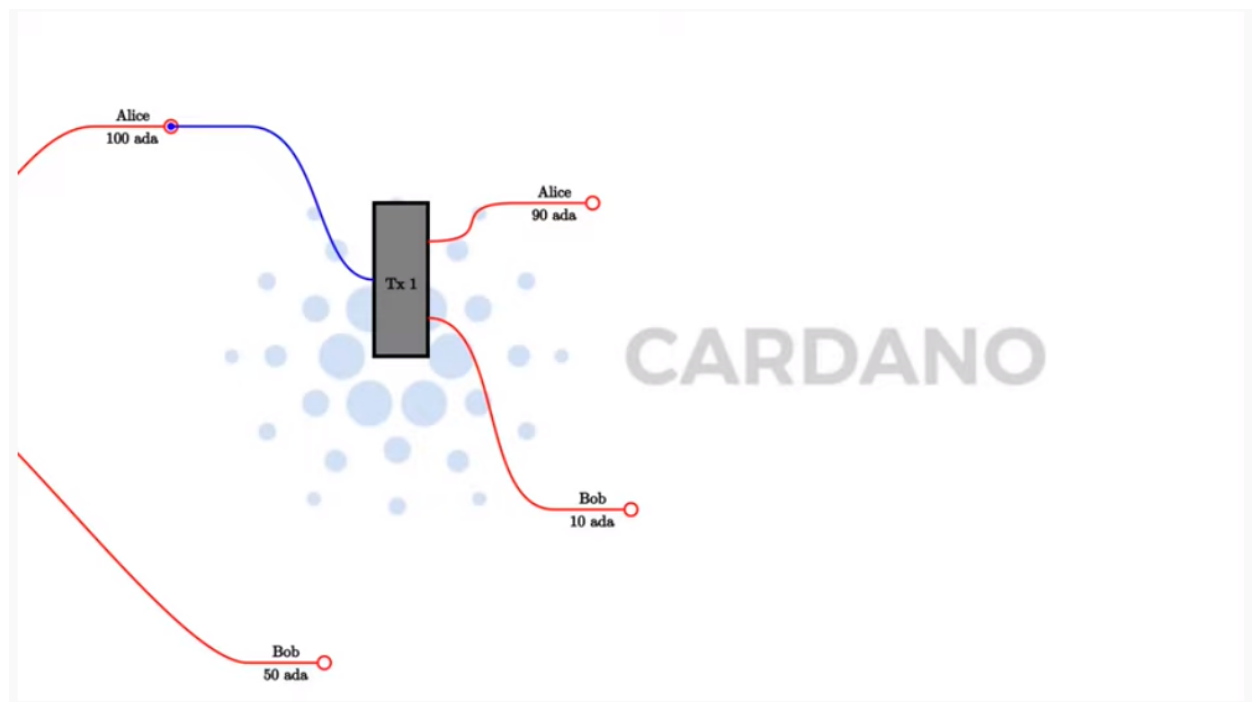
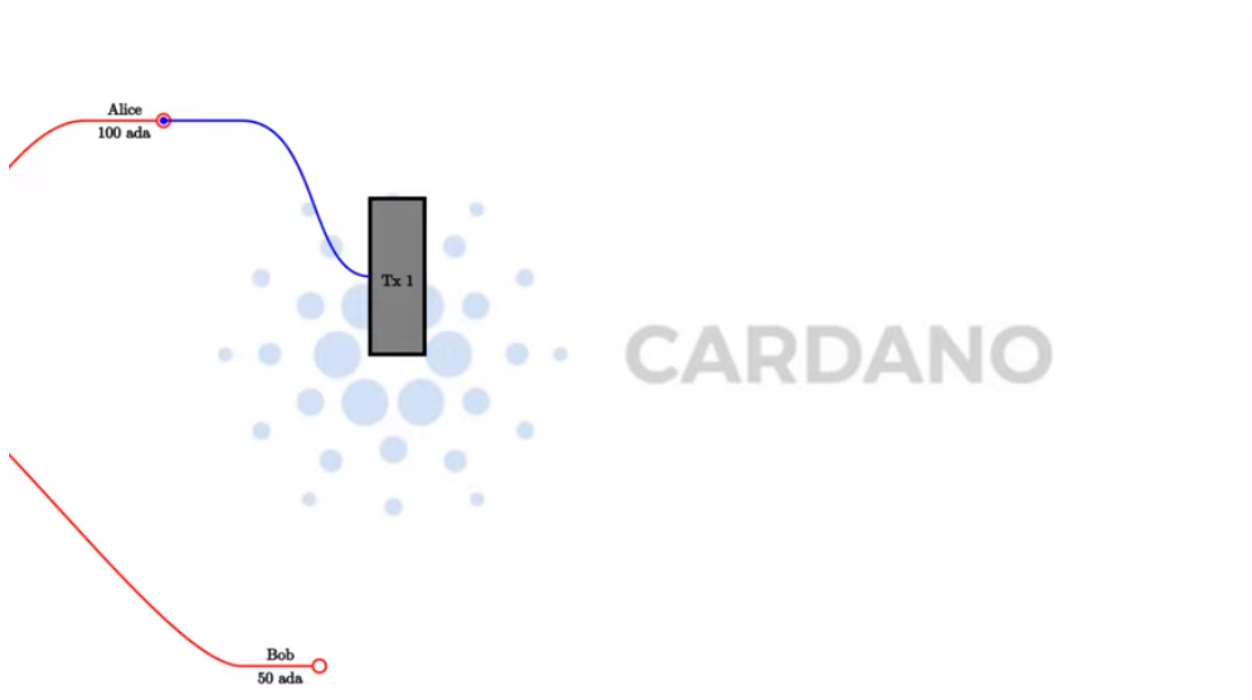
In any transaction, the sum of the output values must match the sum of the input values. Although, strictly speaking, this is not true. There are two exceptions.

1. Transaction fees. In a real blockchain, you have to pay fees for each transactions.
2. Native Tokens. It's possible for transactions to create new tokens, or to burn tokens, in which case the inputs will be lower or higher than the outputs, depending on the scenario.

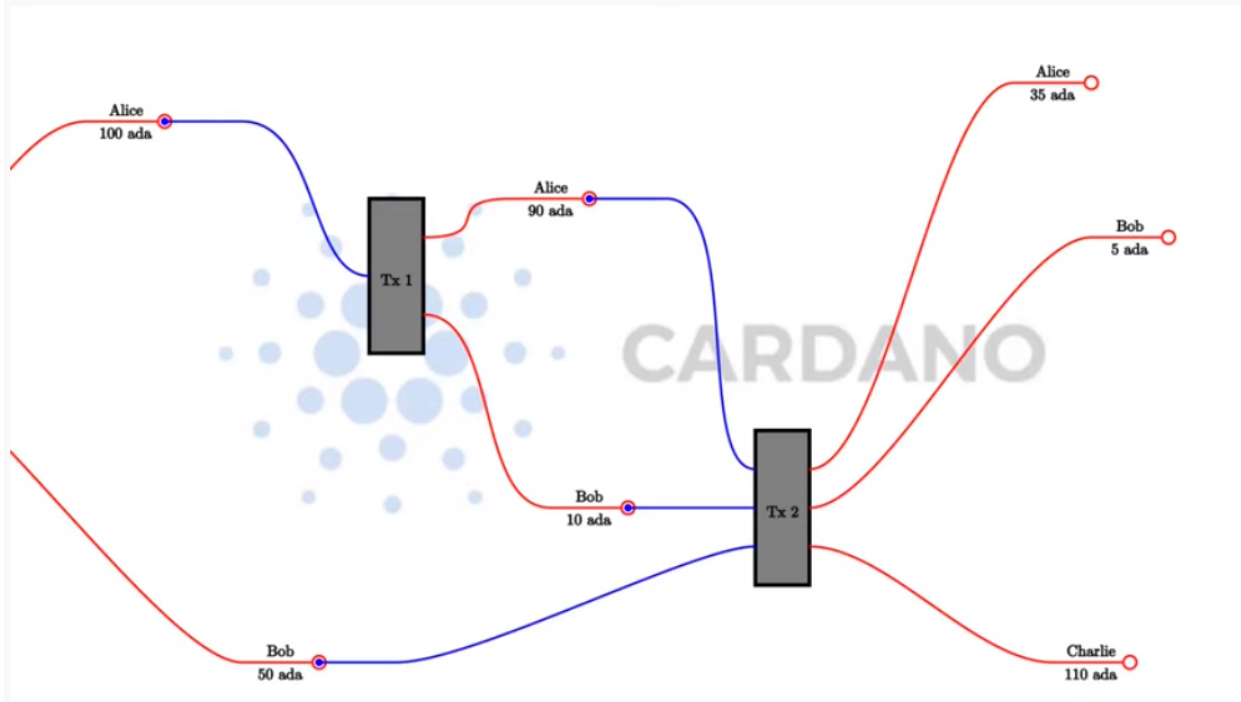
Let's take a look at a slightly more complicated example.

Alice and Bob want to transfer 55 ADA each to Charlie. Alice has no choice, as she only has one UTxO. Bob also has no choice as neither of his two UTxOs is large enough to cover the 55 ADA he wishes to send to Charlie. Bob will





have to use both his UTxOs as input.



### 1.2.2 When Is Spending Allowed?

Obviously it wouldn't be a good idea if any transaction could spend arbitrary UTxOs. If that was the case then Bob could spend Alice's money without her consent.

The way it works is by adding signatures to transactions.

In transaction 1, Alice's signature has to be added to the transaction. In transaction 2, both Alice and Bob need to sign the transaction. Incidentally, this second, more complex, transaction cannot be done in Daedalus, so you would need to use the CLI for this.

Everything explained so far is just about the UTxO model, not the (E)UTxO model.

The extended part comes in when we talk about smart contracts, so in order to understand that, let's concentrate on the consumption of Alice's UTxO of 100 ADA.

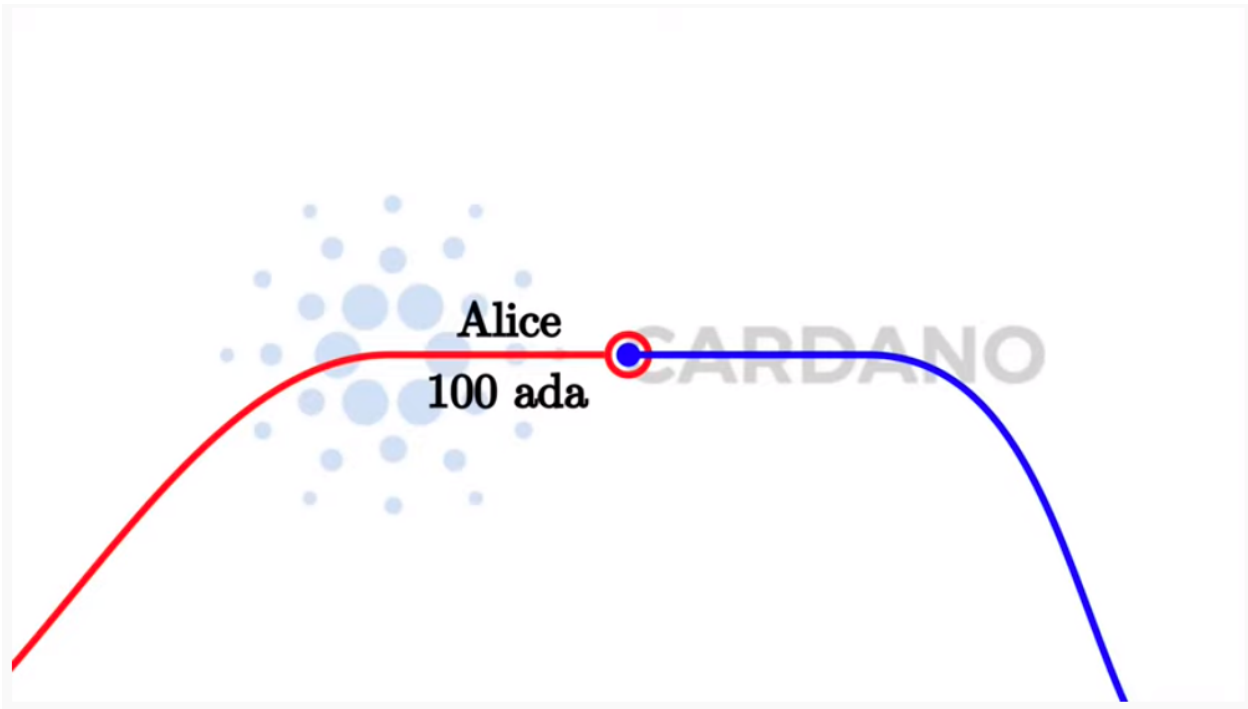
In the UTxO model, the validation that decides whether the transaction that this input belongs to is allowed to consume the UTxO, relies on digital signatures. In this case, that means that Alice has to sign the transaction in order for the consumption of the UTxO to be valid.

The idea of the (E)UTxO model is to make this more general.

Instead of having just one condition, namely that the appropriate signature is present in the transaction, we replace this with arbitrary logic.

This is where Plutus comes in.

Instead of just having an address that corresponds to a public key that can be verified by a signature that is added to the transaction, we have more general addresses, not based on public keys or the hashes of public keys, but instead contain arbitrary logic which decides under which conditions a particular UTxO can be spent by a particular transaction.



So, instead of an input being validated simply by its public key, the input will justify that it is allowed to consume this output with some arbitrary piece of data that we call the *Redeemer*.

We replace the public key address (Alice's in our example), with a script, and we replace the digital signature with a *Redeemer*.

What exactly does that mean? What do we mean by *arbitrary logic*?

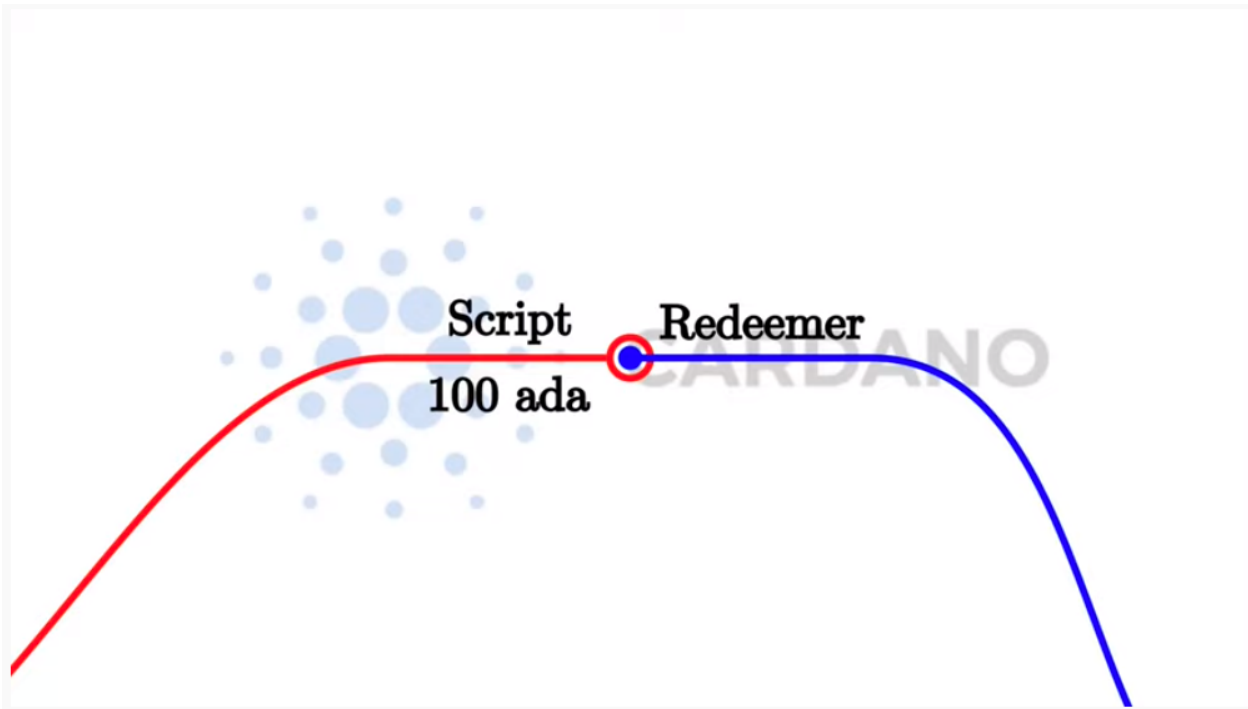
It is important to consider the context that the script has. There are several options.

### 1.2.3 Script Context

#### The Bitcoin approach

One option is that all the script sees is the Redeemer. In this case, the Redeemer contains all the logic necessary to verify the transaction. This is, incidentally, what Bitcoin does. In Bitcoin, there are smart contracts, but they are just not very smart. They are called Bitcoin Script, which works exactly like this. There is a script on the UTxO side and a redeemer on the input side, and the script gets the redeemer and uses it to determine if it is ok to consume the UTxO or not.

But this is not the only option. We can decide to give more information to the script.



### The Ethereum approach

Ethereum uses a different concept. In Ethereum, the script can see everything - the whole blockchain - the opposite extreme of Bitcoin. In Bitcoin, the script has very little context, all it can see is the redeemer. In Ethereum, the Solidity scripts can see the complete state of the blockchain.

This makes Ethereum scripts more powerful, but it also comes with problems. Because the scripts are so powerful it is difficult to predict what a given script will do and that opens the door to all sorts of security issues and dangers. It is very hard for the developers of an Ethereum smart contract to predict everything that can happen.

### The Cardano approach

What Cardano does is something in the middle.

In Plutus, the script cannot see the whole blockchain, but it can see the whole transaction that is being validated. In contrast to Bitcoin, it can't see only the redeemer of the one input, but it can also see all the inputs and outputs of the transaction, and the transaction itself. The Plutus script can use this information to decide whether it is ok to consume the output.

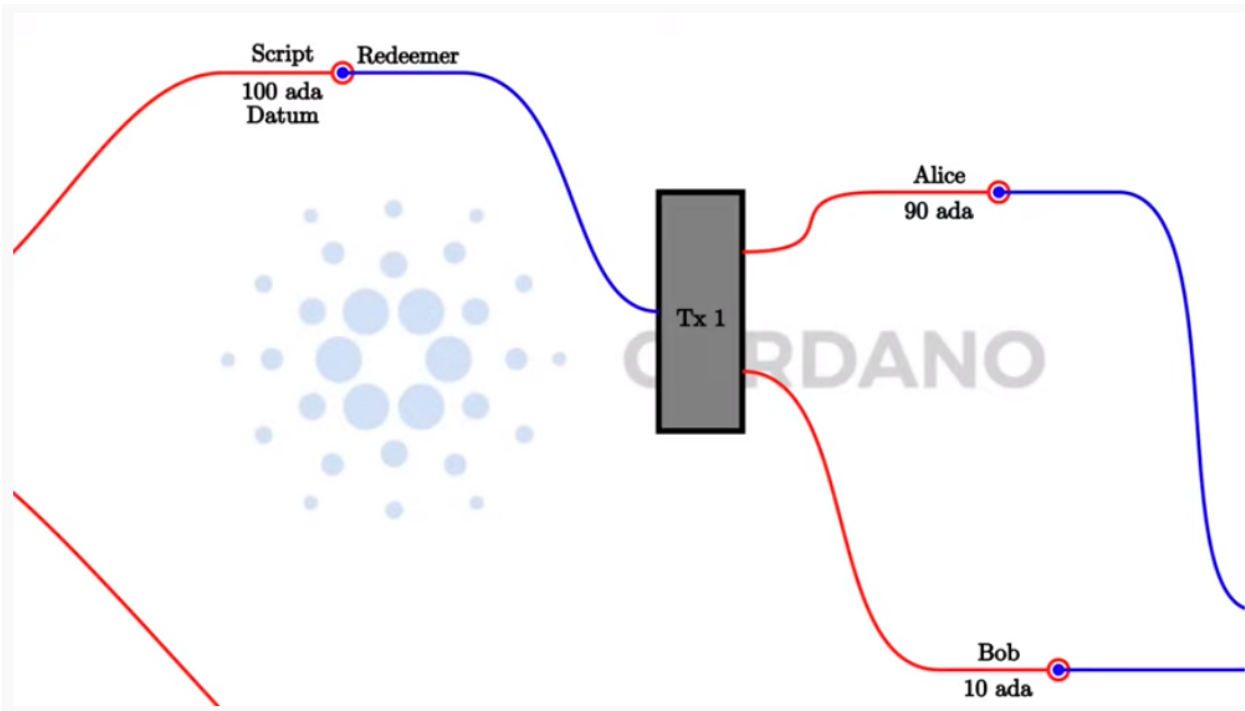
There is one last ingredient that Plutus scripts need in order to be as powerful and expressive as Ethereum scripts. That is the so-called Datum. That is a piece of data that can be associated with a UTxO along with the UTxO value.

With this it is possible to prove mathematically that Plutus is at least as powerful as the Ethereum model - any logic you can express in Ethereum you can also express using the (E)UTxO model.

But it also has a lot of advantages compared to the Ethereum model. For example, in Plutus, it is possible to check whether a transaction will validate in your wallet, before you ever send it to the chain.

Things can still go wrong with off-chain validation, however. For example in the situation where you submit a transaction that has been validated in the wallet but gets rejected when it attempts to consume an output on-chain that has already been consumed by another transaction.

In this case, your transaction will fail without you having to pay any fees.



But if all the inputs are still there that your transaction expects, then you can be sure that the transaction will validate and will have the predicted effect.

This is not the case with Ethereum. In Ethereum, the time between you constructing a transaction and it being incorporated into the blockchain, a lot of stuff can happen concurrently, and that is unpredictable and can have unpredictable effects on what will happen when your script finally executes.

In Ethereum it is always possible that you have to pay gas fees for a transaction even if the transaction eventually fails with an error. And that is guaranteed to never happen with Cardano.

In addition to that, it is also easier to analyse a Plutus script and to check, or even prove, that it is secure, because you don't have to consider the whole state of the blockchain, which is unknowable. You can concentrate on this context that just consists of the spending transaction. So you have a much more limited scope and that makes it much easier to understand what a script is actually doing and what can possibly go wrong.

Who is responsible for providing the datum, redeemer and the validator? The rule in Plutus is that the spending transaction has to do that whereas the producing transaction only has to provide hashes.

That means that if I produce an output that sits at a script address then this producing transaction only has to include the hash of the script and the hash of the datum that belongs to the output. Optionally it can include the datum and the script as well.

If a transaction wants to consume such an output then *that* transaction has to provide the datum, the redeemer and the script. Which means that in order to spend a given input, you need to know the datum, because only the hash is publicly visible on the blockchain.

This is sometimes a problem and not what you want and that's why you have the option to include the datum in the producing transaction. If this were not possible, only people that knew the datum by some means other than looking at the blockchain would ever be able to spend such an output.

The (E)UTxO model is not tied to a particular programming language. What we have is Plutus, which is Haskell, but in principal you could use the same model with a completely different programming language, and we intend to write compilers for other programming languages to Plutus Script which is the "assembly" language underlying Plutus.

## 1.3 Running an example auction contract on a local Playground

Rather than start the traditional way, i.e. starting very simple and doing a crash course on Haskell, followed by some simple Plutus contracts and slowly add more complicated stuff, it will be more interesting, especially for the first lecture, to showcase a more interesting contract and demonstrate what Plutus can do. We can then use that to look at certain concepts in more detail.

### 1.3.1 The English Auction contract

As our introductory example we are going to look at an English Auction. Somebody wants to auction an NFT (Non-fungible token) - a native token on Cardano that exists only once. An NFT can represent some digital art or maybe some real-world asset.

The auction is parameterised by the owner of the token, the token itself, a minimal bid and a deadline.

So let's say that Alice has an NFT and wants to auction it.

Auction  
NFT  
Nothing



She creates a UTxO at the script output. We will look at the code later, but first we will just examine the ideas of the UTxO model.

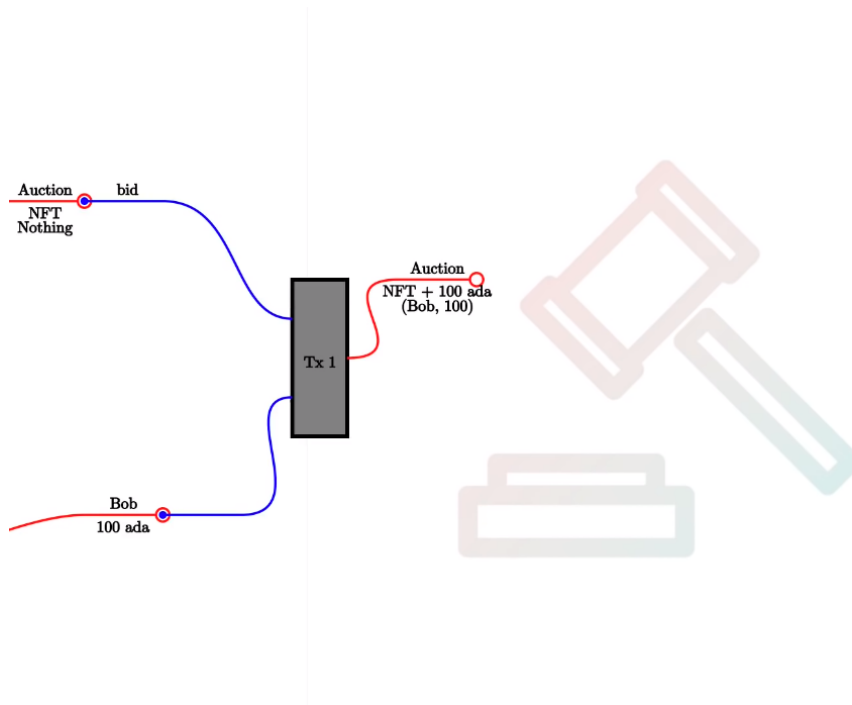
The value of the UTxO is the NFT, and the datum is *Nothing*. Later on it will be the highest bidder and the highest bid. But right now, there hasn't yet been a bid.

In the real blockchain you can't have a UTxO that just contains native tokens, they always have to be accompanied by some Ada, but for simplicity we will ignore that here.

Not let's say that Bob wants to bid 100 Ada.

In order to do this, Bob creates a transaction with two inputs and one output. The first input is the auction UTxO and the second input is Bob's bid of 100 Ada. The output is, again, at the output script, but now the value and the datum has changed. Previously the datum was *Nothing* but now it is (Bob, 100).

The value has changed because now there is not only the NFT in the UTxO, but also the 100 Ada bid.



As a redeemer, in order to unlock the original auction UTxO, we use something called *Bid*. This is just an algebraic data type. There will be other values as well but one of those is *Bid*. And the auction script will check that all the conditions are satisfied. So, in this case the script has to check that the bid happens before the deadline, that the bid is high enough.

It also has to check that the correct inputs and outputs are present. In this case that means checking that the auction is an output containing the NFT and has the correct datum.

Next, let's assume that Charlie wants to outbid Bob and bid 200 Ada.

Charlie will create another transaction, this time one with two inputs and two outputs. As in the first case, the two inputs are the bid (this time Charlie's bid of 200 Ada), and the auction UTxO. One of the outputs is the updated auction UTxO. There will also be a second output, which will be a UTxO which returns Bob's bid of 100 Ada.

---

**Note:** In reality the auction UTxO is not updated because nothing ever changes.

What really happens is that the old auction UTxO is spent and a new one is created, but it has the feel of updating the state of the auction UTxO

---

This time we again use the *Bid* redeemer. This time the script has to check that the deadline has been reached, that the bid is higher than the previous bid, it has to check that the auction UTxO is correctly created and it has to check that the previous highest bidder gets their bid back.

Finally, let's assume that there won't be another bid, so once the deadline has been reached, the auction can be closed.

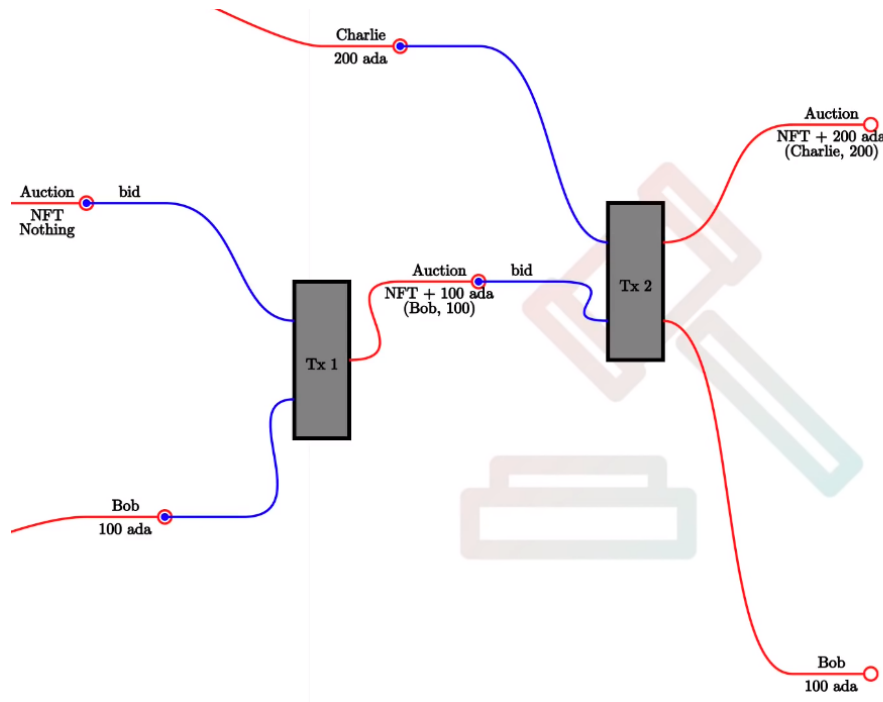
In order to do that, somebody has to create yet another transaction. That could be Alice who wants to collect the bid or it could be Charlie who wants to collect the NFT. It can be anybody, but Alice and Charlie have an incentive to do so.

This transaction will have one input - the auction UTxO, this time with the *Close* redeemer - and it will have two outputs. One of the outputs is for highest bidder, Charlie, and he gets the NFT and the second output goes to Alice who gets the highest bid.

In the *Close* case, the script has to check that the deadline has been reached and that the winner gets the NFT and the







## Off-chain

In order to unlock a UTxO, you must be able to construct a transaction that will pass validation and that is the responsibility of the off-chain part of Plutus. This is the part that runs on the wallet and not on the blockchain and will construct suitable transactions.

One of the nice things about Plutus is that both the on-chain parts and the off-chain parts are written in Haskell. One obvious advantage of that is that you don't have to learn two programming languages. The other advantage is that you can share code between the on-chain and off-chain parts.

Later in this course we talk about state machines and then this sharing between on-chain and off-chain code becomes even more direct, but even without state machines there is still a lot of opportunities to share code.

We will have a brief look at the code but don't worry, you are not expected to understand it at this point.

The code for the English Auction contract is at

```
/path/to/plutus-pioneer-program/repo/code/week01/src/Week01/EnglishAuction.hs
```

We see a data type *Auction* which represents the parameters for the contract that, in our example, Alice starts. The *aCurrency* and *aToken* parameters represent the NFT.

```
data Auction = Auction
  { aSeller    :: !PubKeyHash
  , aDeadline  :: !POSIXTime
  , aMinBid    :: !Integer
  , aCurrency  :: !CurrencySymbol
  , aToken     :: !TokenName
  } deriving (Show, Generic, ToJSON, FromJSON, ToSchema)
```

You also see other data types, but the heart of the code is the *mkAuctionValidator* function. This is the function that determines whether a given transaction is allowed to spend a UTxO sitting at this script address.

```

{-# INLINABLE mkAuctionValidator #-}
mkAuctionValidator :: AuctionDatum -> AuctionAction -> ScriptContext -> Bool
mkAuctionValidator ad redeemer ctx =
  traceIfFalse "wrong input value" correctInputValue &&
  case redeemer of
    MkBid b@Bid{..} ->
      traceIfFalse "bid too low" (sufficientBid bBid) &&
      traceIfFalse "wrong output datum" (correctBidOutputDatum b) &&
      traceIfFalse "wrong output value" (correctBidOutputValue bBid) &&
      traceIfFalse "wrong refund" correctBidRefund &&
      traceIfFalse "too late" correctBidSlotRange
    Close ->
      traceIfFalse "too early" correctCloseSlotRange &&
      case adHighestBid ad of
        Nothing ->
          traceIfFalse "expected seller to get token" (getValue (aSeller_
↪ auction) tokenValue)
        Just Bid{..} ->
          traceIfFalse "expected highest bidder to get token" (getValue_
↪ bBidder tokenValue) &&
          traceIfFalse "expected seller to get highest bid" (getValue_
↪ (aSeller auction) $ Ada.lovelaceValueOf bBid)

  where
    ...

```

And then here is where the compilation to Plutus Core happens. It uses something called Template Haskell to take the Haskell function above and compile it to Plutus Core.

```

auctionTypedValidator :: Scripts.TypedValidator Auctioning
auctionTypedValidator = Scripts.mkTypedValidator @Auctioning
  $(PlutusTx.compile [|| mkAuctionValidator ||])
  $(PlutusTx.compile [|| wrap ||])
where
  wrap = Scripts.wrapValidator

```

The off-chain part of the code defines the endpoints that can be invoked.

We have three endpoints for this example, and each has a datatype defined to represent their parameters.

```

data StartParams = StartParams
  { spDeadline :: !POSIXTime
  , spMinBid   :: !Integer
  , spCurrency :: !CurrencySymbol
  , spToken    :: !TokenName
  } deriving (Generic, ToJSON, FromJSON, ToSchema)

data BidParams = BidParams
  { bpCurrency :: !CurrencySymbol
  , bpToken    :: !TokenName
  , bpBid      :: !Integer
  } deriving (Generic, ToJSON, FromJSON, ToSchema)

data CloseParams = CloseParams

```

(continues on next page)

(continued from previous page)

```

{ cpCurrency :: !CurrencySymbol
, cpToken    :: !TokenName
} deriving (Generic, ToJSON, FromJSON, ToSchema)

```

Then the off-chain operations are defined.

First the *start* logic.

```

start :: AsContractError e => StartParams -> Contract w s e ()
start StartParams{..} = do
    pkh <- pubKeyHash <$> ownPubKey
    let a = Auction
        { aSeller    = pkh
        , aDeadline  = spDeadline
        , aMinBid    = spMinBid
        , aCurrency  = spCurrency
        , aToken     = spToken
        }
    d = AuctionDatum
        { adAuction    = a
        , adHighestBid = Nothing
        }
    v = Value.singleton spCurrency spToken 1
    tx = mustPayToTheScript d v
    ledgerTx <- submitTxConstraints auctionTypedValidator tx
    void $ awaitTxConfirmed $ txId ledgerTx
    logInfo @String $ printf "started auction %s for token %s" (show a) (show v)

```

Then the *bid* logic.

```

bid :: forall w s. BidParams -> Contract w s Text ()
bid BidParams{..} = do
    (oref, o, d@AuctionDatum{..}) <- findAuction bpCurrency bpToken
    logInfo @String $ printf "found auction utxo with datum %s" (show d)

    when (bpBid < minBid d) $
        throwError $ pack $ printf "bid lower than minimal bid %d" (minBid d)
    pkh <- pubKeyHash <$> ownPubKey
    let b = Bid {bBidder = pkh, bBid = bpBid}
        d' = d {adHighestBid = Just b}
        v = Value.singleton bpCurrency bpToken 1 <> Ada.lovelaceValueOf bpBid
        r = Redeemer $ PlutusTx.toData $ MkBid b

    lookups = Constraints.typedValidatorLookups auctionTypedValidator <>
                Constraints.otherScript auctionValidator <>
                Constraints.unspentOutputs (Map.singleton oref o)
    tx = case adHighestBid of
        Nothing    -> mustPayToTheScript d' v
                    mustValidateIn (to $ aDeadline adAuction)
                    mustSpendScriptOutput oref r
        Just Bid{..} -> mustPayToTheScript d' v
                    mustPayToPubKey bBidder (Ada.lovelaceValueOf bBid)
                    mustValidateIn (to $ aDeadline adAuction)

```

(continues on next page)

(continued from previous page)

```

                                mustSpendScriptOutput oref r
ledgerTx <- submitTxConstraintsWith lookups tx
void $ awaitTxConfirmed $ txId ledgerTx
logInfo @String $ printf "made bid of %d lovelace in auction %s for token (%s, %s)"
    bpBid
    (show adAuction)
    (show bpCurrency)
    (show bpToken)

```

And finally the *close* logic.

```

close :: forall w s. CloseParams -> Contract w s Text ()
close CloseParams{..} = do
    (oref, o, d@AuctionDatum{..}) <- findAuction cpCurrency cpToken
    logInfo @String $ printf "found auction utxo with datum %s" (show d)

    let t      = Value.singleton cpCurrency cpToken 1
        r      = Redeemer $ PlutusTx.toData Close
        seller = aSeller adAuction

    lookups = Constraints.typedValidatorLookups auctionTypedValidator <>
              Constraints.otherScript auctionValidator <>
              Constraints.unspentOutputs (Map.singleton oref o)
    tx      = case adHighestBid of
        Nothing      -> mustPayToPubKey seller t <>
                        mustValidateIn (from $ aDeadline adAuction) <>
                        mustSpendScriptOutput oref r
        Just Bid{..} -> mustPayToPubKey bBidder t <>
                        mustPayToPubKey seller (Ada.lovelaceValueOf bBid) <>
                        mustValidateIn (from $ aDeadline adAuction) <>
                        mustSpendScriptOutput oref r

    ledgerTx <- submitTxConstraintsWith lookups tx
    void $ awaitTxConfirmed $ txId ledgerTx
    logInfo @String $ printf "closed auction %s for token (%s, %s)"
        (show adAuction)
        (show cpCurrency)
        (show cpToken)

```

There is some code to tie everything up.

```

endpoints :: Contract () AuctionSchema Text ()
endpoints = (start' `select` bid' `select` close') >> endpoints
where
    start' = endpoint @"start" >=> start
    bid'   = endpoint @"bid"   >=> bid
    close' = endpoint @"close" >=> close

```

And the last lines are just helpers to create a sample NFT to allow us to try the auctioning of this NFT in the playground.

```

mkSchemaDefinitions 'AuctionSchema

myToken :: KnownCurrency
myToken = KnownCurrency (ValidatorHash "f") "Token" (TokenName "T" :| [])

```

(continues on next page)

(continued from previous page)

```
mkKnownCurrencies ['myToken]
```

An example of code reuse is the *minBid* function.

```
minBid :: AuctionDatum -> Integer
minBid AuctionDatum{..} = case adHighestBid of
    Nothing    -> aMinBid adAuction
    Just Bid{..} -> bBid + 1
```

This function gets used in the on-chain part for validation, but also in the off-chain code, in the wallet, before it even bothers to create the transaction, to check whether it is worth doing so.

## 1.4 To the Playground

We will run this contract in our local Plutus Playground.

### 1.4.1 Plutus Setup

Before compiling the sample contract code, we need to setup Plutus. It is advisable to set up a Nix shell from the main Plutus repository at which can also be used to compile the example contracts.

There are detailed notes on how to do this [here](#).

This will setup your environment with the dependencies necessary to compile the sample contracts.

Once you are inside the Nix shell, you can start the Plutus client and server from the cloned Plutus repository.

The lecture videos were recorded at various times and the Plutus code that goes along with them was compiled against specific commits of the Plutus main branch. You can find the commit tag in the `cabal.project` file.

#### Server

```
cd /path/to/plutus/repo/plutus-playground-client
plutus-playground-server
```

#### Client

```
cd /path/to/plutus/repo/plutus-playground-client
npm run start
```

To check that everything is in order, you can then compile the code for Week 01. This is not necessary to run the code in the playground, as the playground can compile the code itself.


```
cd /path/to/plutus-pioneer-program/repo/code/week01
cabal build all
```

If all went well in the setup above, you should be able to open the playground at <https://localhost:8009>. You will likely receive a certificate error, which can be bypassed.

Copy and paste the `EnglishAuction.sh` file contents into the playground, replacing the existing demo contract.

# Plutus Pioneer Program Lecture Notes

PLUTUS REFRESH · UPDATED 25TH JANUARY 2021

 PLUTUS PLAYGROUND

[Getting Started](#) [Tutorials](#) [API](#) [Privacy](#)

[Demo files](#) [Hello world](#) [Starter](#) [Game](#) [Vesting](#) [Crowd Funding](#) [Error Handling](#)

[Log In](#)

Editor


Key Bindings Default

Compile Simulate

```
1 {- Vesting scheme as a PLC contract
2 import      Control.Monad      (void, when)
3 import qualified Data.Map       as Map
4 import qualified Data.Text      as T
5
6 import      Ledger              (Address, PubKeyHash, Slot (Slot), Validator)
7 import qualified Ledger
8 import qualified Ledger.Ada     as Ada
9 import      Ledger.Constraints  (TxConstraints, mustBeSignedBy, mustPayToTheScript, mustValidateIn)
10 import     Ledger.Contexts     (ScriptContext (..), TxInfo (..))
11 import qualified Ledger.Contexts as Validation
12 import qualified Ledger.Interval as Interval
13 import qualified Ledger.Time     as Time
14 import qualified Ledger.TimeSlot as TimeSlot
15 import qualified Ledger.Tx       as Tx
16 import qualified Ledger.Typed.Scripts as Scripts
17 import      Ledger.Value       (Value)
18 import qualified Ledger.Value     as Value
19 import      Playground.Contract
20 import      Plutus.Contract     hiding (when)
21 import qualified Plutus.Contract.Typed.Tx as Typed
22 import qualified PlutusTx
23 import      PlutusTx.Prelude    hiding (Semigroup (..), fold)
24 import      Prelude            as Haskell (Semigroup (..), show)
25 import      Wallet.Emulator.Types (walletPubKey)
26
```

Not compiled

PLUTUS REFRESH · UPDATED 25TH JANUARY 2021

 PLUTUS PLAYGROUND

[Getting Started](#) [Tutorial](#) [API](#) [Privacy](#)

[Demo files](#) [Hello world](#) [Starter](#) [Game](#) [Vesting](#) [Crowd Funding](#) [Error Handling](#)

[Log In](#)

Editor

Key Bindings Default

Compile Simulate

```
1 {-# LANGUAGE DataKinds           #-}
2 {-# LANGUAGE DeriveAnyClass      #-}
3 {-# LANGUAGE DeriveGeneric       #-}
4 {-# LANGUAGE DerivingStrategies  #-}
5 {-# LANGUAGE FlexibleContexts    #-}
6 {-# LANGUAGE GeneralizedNewtypeDeriving #-}
7 {-# LANGUAGE LambdaCase         #-}
8 {-# LANGUAGE MultiParamTypeClasses #-}
9 {-# LANGUAGE NoImplicitPrelude   #-}
10 {-# LANGUAGE OverloadedStrings   #-}
11 {-# LANGUAGE RecordWildCards     #-}
12 {-# LANGUAGE ScopedTypeVariables #-}
13 {-# LANGUAGE TemplateHaskell     #-}
14 {-# LANGUAGE TypeApplications    #-}
15 {-# LANGUAGE TypeFamilies        #-}
16 {-# LANGUAGE TypeOperators       #-}
17
18 import      Control.Monad      hiding (fmap)
19 import      Data.Aeson         (ToJSON, FromJSON)
20 import      Data.List.NonEmpty (NonEmpty (..))
21 import      Data.Map           as Map
22 import      Data.Text          (pack, Text)
23 import      GHC.Generics       (Generic)
24 import      Plutus.Contract     hiding (when)
25 import qualified PlutusTx       as PlutusTx
26 import      PlutusTx.Prelude    hiding (Semigroup(..), unless)
```

Compilation successful

Click the compile button. Once it has compiled, click the Simulate button.

The default wallets are setup with 10 Lovelace and 10 T, where T is a native token simulated by the script in the following lines:

```
myToken :: KnownCurrency
myToken = KnownCurrency (ValidatorHash "f") "Token" (TokenName "T" :| [])

mkKnownCurrencies ['myToken]
```

We are going to treat the token T as a non-fungible token (NFT), and simulate this by changing the wallets such that Wallet 1 has 1 T and the other wallets have 0 T.

Also, 10 lovelace is ridiculously low, so let's give each wallet 1000 Ada, which is 1,000,000,000 lovelace.

Click the “Add Wallet” option, then adjust the balances accordingly:

You can see in the playground that the contract has three endpoints: start, bid, and close.

The “Pay to Wallet” endpoint is always there by default in the playground. It allows a simple transfer of Lovelace from one wallet to another.

Click “start” on wallet 1, to create an auction:

This is where the seller is going to set the rules for the auction.

The getSlot field specifies the deadline for the auction. Bidding after this deadline will not be allowed by the contract.

Let's say that the deadline is Slot 10.

Time is measured in POSIX time (seconds since 1st January 1970), so we need to calculate this value. Luckily in the *plutus-ledger* package in module *Ledger.Timeslot*, there is a function *slotToPOSIXTime*. If we import this into the REPL, we can get the value we need. The simulation starts at the beginning of the Shelley era, so this value - 1596059101 - reflects that and this will be on July 29th 2020 - the 10th slot of the Shelley era.

PLUTUS REFRESH - UPDATED 25TH JANUARY 2021

PLUTUS PLAYGROUND

[Getting Started](#) [Tutorials](#) [API](#) [Privacy](#)

[Demo files](#) [Hello\\_world](#) [Starter](#) [Game](#) [Vesting](#) [Crowd Funding](#) [Error Handling](#)

Simulation 1

+

Wallets

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

Wallet 1

Opening Balances

Lovelace1000000000

T1

Available functions

bidclosestartPay to Wallet

Wallet 2

Opening Balances

Lovelace1000000000

T0

Available functions

bidclosestartPay to Wallet

Wallet 3

Opening Balances

Lovelace1000000000

T0

Available functions

bidclosestartPay to Wallet

+

Add Wallet

Evaluate

Transactions

Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

+

Add Wait Action

Evaluate

Transactions

[cardano.org](#) [iohk.io](#)

© 2020 IOHK Ltd.

[GitHub](#) [Twitter](#) [Feedback](#)

```
Prelude Week01.EnglishAuction> import Ledger.TimeSlot
Prelude Ledger.TimeSlot Week01.EnglishAuction> slotToPOSIXTime 10
POSIXTime {getPOSIXTime = 1596059101}
```

Add this value to the deadline field.

The `spMinField` specifies the minimum amount of ADA that must be bid. If this minimum is not met by the deadline, no bid will succeed. Let's make this 100 Ada.

Enter 1000000000 into the `spMinBid` field.

The last two fields - `spCurrencySymbol` and `unTokenName` specify the currency of the NFT that is the subject of the auction. In Plutus a native token is defined by a currency symbol and a name.

In this case, the symbol is 66 and the token name, as we have seen is T.

Enter these values into their respective fields.

We can also insert "wait" actions, to wait for a certain number of slots. We will need to wait for at least one slot in order for the transaction to start the auction to complete.

Now bidding can start.

Let's say that Wallets 2 and 3 want to bid for this token.

Wallet 2 is faster, and bids 100 Ada by invoking the `bid` endpoint with the parameters as shown below.

We now insert another wait action, and now we add a bid by Charlie (Wallet 3) for 200 Ada.

Let's say that these two bids are the only bids.

We now add a wait action that will wait until slot 11, which is the slot after the deadline of the auction.



## Wallets

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

Wallet 1

Opening Balances

Lovelace1000000000

T1

Available functions

bid +close +start +

Pay to Wallet +

Wallet 2

Opening Balances

Lovelace1000000000

T0

Available functions

bid +close +start +

Pay to Wallet +

Wallet 3

Opening Balances

Lovelace1000000000

T0

Available functions

bid +close +start +

Pay to Wallet +

+

Add Wallet

## Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: start

spDeadline

getPOSIXTime

1596059101

✓

spMinBid

100000000

✓

spCurrency

unCurrencySymbol

66

✓

spToken

unTokenName

T

✓

2

Wait

☒ Wait For...☐ Wait Until...

Blocks1

+

Add Wait Action

Evaluate

Transactions

[cardano.org](#)[iohk.io](#)

© 2020 IOHK Ltd.

[GitHub](#)[Twitter](#)[Feedback](#)

Wallets

EvaluateTransactions

Wallet 1

Opening Balances

Lovelace1000000000

T1

Available functions

bidclosestartPay to Wallet

Wallet 2

Opening Balances

Lovelace1000000000

T0

Available functions

bidclosestartPay to Wallet

Wallet 3

Opening Balances

Lovelace1000000000

T0

Available functions

bidclosestartPay to Wallet

Add Wallet

Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: start

spDeadline

getPOSIXTime

1596059101

spMinBid

1000000000

spCurrency

unCurrencySymbol

66

spToken

unTokenName

T

EvaluateTransactions

2

Wait

Wait For...Wait Until...

Blocks1

3

Wallet 2: bid

bpCurrency

unCurrencySymbol

66

bpToken

unTokenName

T

bpBid

1000000000

Add Wait Action

cardano.orgiohk.io

© 2020 IOHK Ltd.

GitHubTwitterFeedback

Wallets

EvaluateTransactions

Wallet 1

Opening Balances

Lovelace1000000000

T1

Available functions

bidclosestartPay to Wallet

Wallet 2

Opening Balances

Lovelace1000000000

T0

Available functions

bidclosestartPay to Wallet

Wallet 3

Opening Balances

Lovelace1000000000

T0

Available functions

bidclosestartPay to Wallet

Add Wallet

Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: start

spDeadline

getPOSIXTime

1596059101

spMinBid

1000000000

spCurrency

unCurrencySymbol

66

spToken

unTokenName

T

Add Wait Action

EvaluateTransactions

2

Wait

Wait For...Wait Until...

Blocks1

3

Wallet 2: bid

bpCurrency

unCurrencySymbol

66

bpToken

unTokenName

T

bpBid

1000000000

4

Wait

Wait For...Wait Until...

Blocks1

5

Wallet 3: bid

bpCurrency

unCurrencySymbol

66

bpToken

unTokenName

T

bpBid

2000000000

The screenshot displays the Plutus Pioneer Program simulator interface. At the top, there are three wallet configuration panels, each with a 'Lovelace' field set to 1000000000 and a 'T' field set to 1. Below these are 'Available functions' buttons: 'bid', 'close', 'start', and 'Pay to Wallet'. The main 'Actions' section contains a sequence of six actions:

- 1. Wallet 1: start**: Includes fields for 'spDeadline' (1596059101), 'spMinBid' (100000000), 'spCurrency' (66), 'unCurrencySymbol' (T), and 'spToken' (unTokenName).
- 2. Wait**: Includes a 'Wait For...' dropdown, a 'Wait Until...' dropdown, and a 'Blocks' field set to 1.
- 3. Wallet 2: bid**: Includes fields for 'bpCurrency' (66), 'unCurrencySymbol' (T), 'bpToken' (unTokenName), and 'bpBid' (100000000).
- 4. Wait**: Includes a 'Wait For...' dropdown, a 'Wait Until...' dropdown, and a 'Blocks' field set to 1.
- 5. Wallet 3: bid**: Includes fields for 'bpCurrency' (66), 'unCurrencySymbol' (T), 'bpToken' (unTokenName), and 'bpBid' (200000000).
- 6. Wait**: Includes a 'Wait For...' dropdown, a 'Wait Until...' dropdown, and a 'Slot' field set to 11.

At the bottom of the actions list is an 'Add Wait Action' button. Below the actions are 'Evaluate' and 'Transactions' buttons. The footer includes links to 'cardano.org', 'iohk.io', '© 2020 IOHK Ltd.', 'GitHub', 'Twitter', and 'Feedback'.

At this point, anybody can invoke the *close* endpoint. The auction will not settle on its own, it needs to be triggered by an endpoint.

When the *close* endpoint is triggered, the auction will be settled according to the rules.

- If there was at least one bid, the highest bidder will receive the token. This will always be the last bidder as the script will not allow bids that are not higher than the existing highest bid or bids that are lower than the minimum bid level.
- If there were no bidders, Wallet 1 will get the token back.

Let's say that Alice (Wallet 1) invokes the *close* endpoint. We will add this and also add another wait action, which we need at the end in order to see the final transaction when we run the simulation.

Now, click the "Evaluate" button - either the one at the bottom or the one at the top of the page.

After a little while, you will see the simulator view.

Towards the top of the page you will see the slots that are relevant to the simulation, that is, the slots where an action occurred. Here we see that these are slots 1,2,3,4 and 20.

Slot zero is not caused by our contract, it is the Genesis transaction that sets up the initial balances of the wallets. There are three outputs for this transaction.

Now click on the Slot 1 transaction.

The transaction has one input and three outputs. The input is the only UTxO that Wallet 1 has. Even though it is two tokens, 1000 Ada and 1T, they sit in one UTxO. As mentioned earlier, UTxOs always need to be consumed in their entirety, so the entire UTxO is sent as input.

The outputs are a 10 lovelace fee (this is a demo fee and does not reflect what a real fee would be), 999,999,990 lovelace back Wallet 1, and 1 T to the contract to hold onto while the bidding takes place. Here you also see the script address.

As we know from the introduction to the UTxO model, there can also be a datum, and there is a datum, but this is not visible in this display.

Available functions

bid + close + start +

Pay to Wallet +

Available functions

bid + close + start +

Pay to Wallet +

Available functions

bid + close + start +

Pay to Wallet +

Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

2

3

4

5

6

7

8

Wallet 1: start

spDeadline

getPOSIXTime

1596059101

spMinBid

100000000

spCurrency

unCurrencySymbol

66

spToken

unTokenName

T

Wait

Wait For... Wait Until...

Blocks

1

Wallet 2: bid

bpCurrency

unCurrencySymbol

66

bpToken

unTokenName

T

bpBid

100000000

Wait

Wait For... Wait Until...

Blocks

1

Wallet 3: bid

bpCurrency

unCurrencySymbol

66

bpToken

unTokenName

T

bpBid

200000000

Wait

Wait For... Wait Until...

Slot

11

Wallet 1: close

cpCurrency

unCurrencySymbol

66

cpToken

unTokenName

T

Wait

Wait For... Wait Until...

Blocks

1

Add Wait Action

Evaluate

Transactions

cardano.org iohk.io

© 2020 IOHK Ltd.

[GitHub](#)
[Twitter](#)
[Feedback](#)

## Simulator

Simulation 1 +

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0 Slot 3, Tx 0 Slot 11, Tx 0

Inputs

Transaction

Slot 0, Tx 0

Tx: e7474b9a08609a9ebb87542d364c6e58b6ea83b2c977e98d9a32d7d5247a45c1

Validity: All time

Signatures:None

Forge

Ada

Lovelace

3,000,000,000

66

T

1

Outputs

Wallet 2

PubKeyHash 39f713d0a644253f04529421b9f51e9b...

Ada

Lovelace

1,000,000,000

66

T

0

Unspent

Wallet 1

PubKeyHash 21fe31dfa154a261626b854046fd2271...

Ada

Lovelace

1,000,000,000

66

T

1

Spent In: Slot 1, Tx 0

Wallet 3

PubKeyHash dac073e0123bdea59dd9b3bda9cfe03...

Ada

Lovelace

1,000,000,000

66

T

0

Unspent

Balances Carried Forward (as at Slot 0, Tx 0)

24

Chapter 1. Week 01 - English Auction

Simulator [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0   Slot 1, Tx 0   Slot 2, Tx 0   Slot 3, Tx 0   Slot 11, Tx 0

#### Inputs

Wallet 1  
PubKeyHash 21fe31dfa154a261626b854046fd2271...

Ada  
Lovelace 1,000,000,000

66  
T 1

Created by: Slot 0, Tx 0

#### Transaction

Slot 1, Tx 0

Tx: 1ae51f0514155523289cd4ba1bd391011143dce3933543fc69bcd3748119e435

Validity: All time

Signatures:

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021ae68f707511a

#### Outputs

Fee

Ada  
Lovelace 10

Wallet 1  
PubKeyHash 21fe31dfa154a261626b854046fd2271...

Ada  
Lovelace 999,999,990

66  
T 0

Spent in: Slot 11, Tx 0

Script aff3035b5ba64d5d6805efb94b97e...

66  
T 1

Spent in: Slot 2, Tx 0

Balances Carried Forward (as at Slot 1, Tx 0)

	Ada	66
Beneficial Owner		
Wallet 1 PubKeyHash 21fe31dfa154a261626b854046fd2271...	999,999,990	0

So now the auction is set up, let's look at the next transaction, where Bob (Wallet 2) makes a bid of 100 Ada.

There are two inputs - the script UTxO and the UTxO that Bob owns.

There are also three outputs. The first is a fee of 14,129 lovelace. The second gives Bob his change - his original sum minus the fees and bid. The third output locks the bid into the contract.

The script validator here must make sure that Wallet 2 can't just take the token, so it will only validate in a scenario where there is an output where the token ends up in the contract again. Remember that in the (E)UTxO model, all inputs and outputs are visible to the script.

Now let's look at the next transaction. This is where Charlie bids 200 Ada Lovelace (it is 5 Lovelace in Lars' videos, but I entered it as 4 and I'd rather not take all those screenshots again).

The inputs here are Wallet 3's UTxO and the script address.

The outputs are the change of 6 Lovelace to Wallet 3, the updated script with the new high bid of 4 Lovelace, and the return of Wallet 2's bid of 3 Lovelace to Wallet 2's address.

Again, the logic in the script must make sure that all of this is handled correctly, i.e. that the new bid is higher than the previous bid and that the token T continues to be locked in the contract along with the new bid.

The last transaction is the *close* action. This two inputs - one from Alice in order to pay for the fees, and the second is the script UTxO as input. There are four outputs - the fees from Alice and the change back to Alice, and then the successful bid of 200 Ada to Alice and the transfer of the NFT to Charlie.

If we scroll down, we can now see the final balances.

Let's check what happens when something goes wrong, for example, if Charlie makes a bid that is lower than Bob's bid. Let's say Charlie makes a mistake and bids only 20 Ada.

Now we see that we have only four transactions, and Bob wins the auction.

Let's see what happens if there are no valid bids.

**Simulator** [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0 Slot 3, Tx 0 Slot 11, Tx 0

#### Inputs

Script aff3035b5ba64d5d6805efb94b97e...

66  
T 1  
Created by: Slot 1, Tx 0

Wallet 2  
PubKeyHash 39f713d0a644253f04529421b9f51b9b...

Ada  
Lovelace 1,000,000,000  
66  
T 0  
Created by: Slot 0, Tx 0

#### Transaction

Slot 2, Tx 0

Tx: 5ea56597307614b11f3768bfb285fddbbe96f8b6ed4d591053b79399bc4f3

Validity: From the start of time (inclusive) to Slot 10 (inclusive)

Signatures:

- PubKey 3d4017c3e843895a92b70aa74d1b7ebc9c982cc2ec4968cc0cd59f12af4660c

#### Outputs

Fee  
Ada  
Lovelace 14,129

Wallet 2  
PubKeyHash 39f713d0a644253f04529421b9f51b9b...

Ada  
Lovelace 899,985,871  
66  
T 0  
Unspent

Script aff3035b5ba64d5d6805efb94b97e...

66  
T 1  
Ada  
Lovelace 100,000,000  
Spent In: Slot 3, Tx 0

#### Balances Carried Forward (as at Slot 2, Tx 0)

Beneficial Owner	Ada	66
Wallet 1	999,999,990	0

**Simulator** [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0 Slot 3, Tx 0 Slot 11, Tx 0

#### Inputs

Script aff3035b5ba64d5d6805efb94b97e...

66  
T 1  
Ada  
Lovelace 100,000,000  
Created by: Slot 2, Tx 0

Wallet 3  
PubKeyHash dac073e0123bdea59dd9b3bda9cf603...

Ada  
Lovelace 1,000,000,000  
66  
T 0  
Created by: Slot 0, Tx 0

#### Transaction

Slot 3, Tx 0

Tx: c00e4a72fd1ed9ea2d73951a2b33447b897f18ce536cedb7a11f91b3fb384ae

Validity: From the start of time (inclusive) to Slot 10 (inclusive)

Signatures:

- PubKey fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025

#### Outputs

Fee  
Ada  
Lovelace 14,340

Wallet 3  
PubKeyHash dac073e0123bdea59dd9b3bda9cf603...

Ada  
Lovelace 799,985,660  
66  
T 0  
Unspent

Script aff3035b5ba64d5d6805efb94b97e...

66  
T 1  
Ada  
Lovelace 200,000,000  
Unspent

Wallet 2  
PubKeyHash 39f713d0a644253f04529421b9f51b9b...

Ada  
Lovelace 100,000,000  
Unspent



## Simulator

[Return to Editor](#)

Simulation 1 +

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 11, Tx 0

Inputs

Wallet 1

PubKeyHash 21fe31dfa154a261626b854046fd2271...

Ada

Lovelace

999,999,990

66

T

0

Created by: Slot 1, Tx 0

Script aff3035b5ba64d5d6805efb94b97e...

66

T

1

Ada

Lovelace

100,000,000

Created by: Slot 2, Tx 0

Transaction

Slot 11, Tx 0

Tx: 1e2d24ec6990f132bf68c5b20483b0c55d354dc7bf34c044dac8cd2957befed6

Validity: From Slot 10 (inclusive) to the end of time (inclusive)

Signatures:

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

Outputs

Fee

Ada

Lovelace

13,784

Wallet 1

PubKeyHash 21fe31dfa154a261626b854046fd2271...

Ada

Lovelace

999,986,206

66

T

0

Unspent

Wallet 1

PubKeyHash 21fe31dfa154a261626b854046fd2271...

Ada

Lovelace

100,000,000

Unspent

Wallet 2

PubKeyHash 39f713d0a644253f04529421b9f51b9b...

66

T

1

Unspent

Balances Carried Forward (as at Slot 11, Tx 0)

## Simulator

[Return to Editor](#)

Simulation 1 +

Wallets

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

Wallet 1

Opening Balances

Lovelace

1000000000

T

1

Available functions

bid + close + start +

Pay to Wallet +

Wallet 2

Opening Balances

Lovelace

1000000000

T

0

Available functions

bid + close + start +

Pay to Wallet +

Wallet 3

Opening Balances

Lovelace

1000000000

T

0

Available functions

bid + close + start +

Pay to Wallet +

+ Add Wallet

Evaluate

Transactions

Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: start

spDeadline

getPOSIXTime

1596059101

spMinBid

100000000

spCurrency

unCurrencySymbol

66

spToken

unTokenName

T

Evaluate

Transactions

2

Wait

Wait For... Wait Until...

Slot

11

3

Wallet 1: close

cpCurrency

unCurrencySymbol

66

cpToken

unTokenName

T

4

Wait

Wait For... Wait Until...

Blocks

1

+ Add Wait Action

28

Chapter 1. Week 01 - English Auction



Now there are only three transactions, the last of which is the close transaction. As this is a failed auction, where there was no successful bid, this transaction returns the NFT to Wallet 1.

**Simulator** [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0   Slot 1, Tx 0   Slot 11, Tx 0

#### Inputs

**Wallet 1**  
PubKeyHash 21fe31dfa154a261626b854046d2271...

**Ada**  
Lovelace 999,999,990

**66**  
T 0

Created by: Slot 1, Tx 0

**Script** aff3035b5ba64d5d6805efb94b97e...

**66**  
T 1

Created by: Slot 1, Tx 0

#### Transaction

Slot 11, Tx 0

**Tx:** 5a1bd0c08bf8d36624f4b4940be4cd5b1ac92fd100d4a9be4247339c38b9c3b

**Validity:** From Slot 10 (inclusive) to the end of time (inclusive)

**Signatures:**

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

#### Outputs

**Fee**

**Ada**  
Lovelace 13,573

**Wallet 1**  
PubKeyHash 21fe31dfa154a261626b854046d2271...

**Ada**  
Lovelace 999,986,417

**66**  
T 0

Unspent

**Wallet 1**  
PubKeyHash 21fe31dfa154a261626b854046d2271...

**66**  
T 1

Unspent

#### Balances Carried Forward (as at Slot 11, Tx 0)

	Ada	66
<b>Beneficial Owner</b>	Lovelace	T
<b>Wallet 1</b> PubKeyHash 21fe31dfa154a261626b854046d2271b7bed4b5abe45a58871e4769721b9	999,986,417	1



## WEEK 02 - VALIDATION

---

**Note:** This is a written version of [Lecture #2, Iteration #2](#).

It covers low-level, untyped on-chain validation scripts and high-level, typed on-chain validation scripts.

The code in this lecture uses the Plutus commit `81ba78edb1d634a13371397d8c8b19829345ce0d`

---

### 2.1 Before We Start

Let's talk about an important point that was brought up by one of the pioneers following lecture #1.

You will recall in the auction example, we created three endpoints - `start`, `bid` and `close`. For `close` there were two scenarios. If there was a high-enough bid, the token goes to the highest bidder. If there was not a high-enough bid, the token goes back to the seller.

What would happen if the `close` endpoint wasn't there? Could the money be locked forever in the contract?

This is a really important point, because what you have to realise is that that UTxOs on the blockchain are just data, they are absolutely passive. In order for anything to happen there must be a transaction. In order to make progress and to change the state of the blockchain, there must be a new transaction submitted that consumes various UTxOs and produces new UTxOs.

Only new transactions change the state. A UTxO will never spring into action by itself and do something. You can't have a smart contract that sits on the blockchain and then, at some point, suddenly performs an action.

So, we really need the `close` endpoint if we want the auction to be settled. In our case, the endpoint was manually triggered. You could write a contract that runs in the wallet that would automatically generate the `close` transaction - it is possible to write quite sophisticated off-chain code.

However, from the point-of-view of the blockchain, it is always an external trigger that does something. Nothing happens if it is not externally triggered.

So, if there were no `close` endpoint, or the `close` endpoint never got triggered, the funds would remain sitting at the script address forever.

## 2.2 Introduction

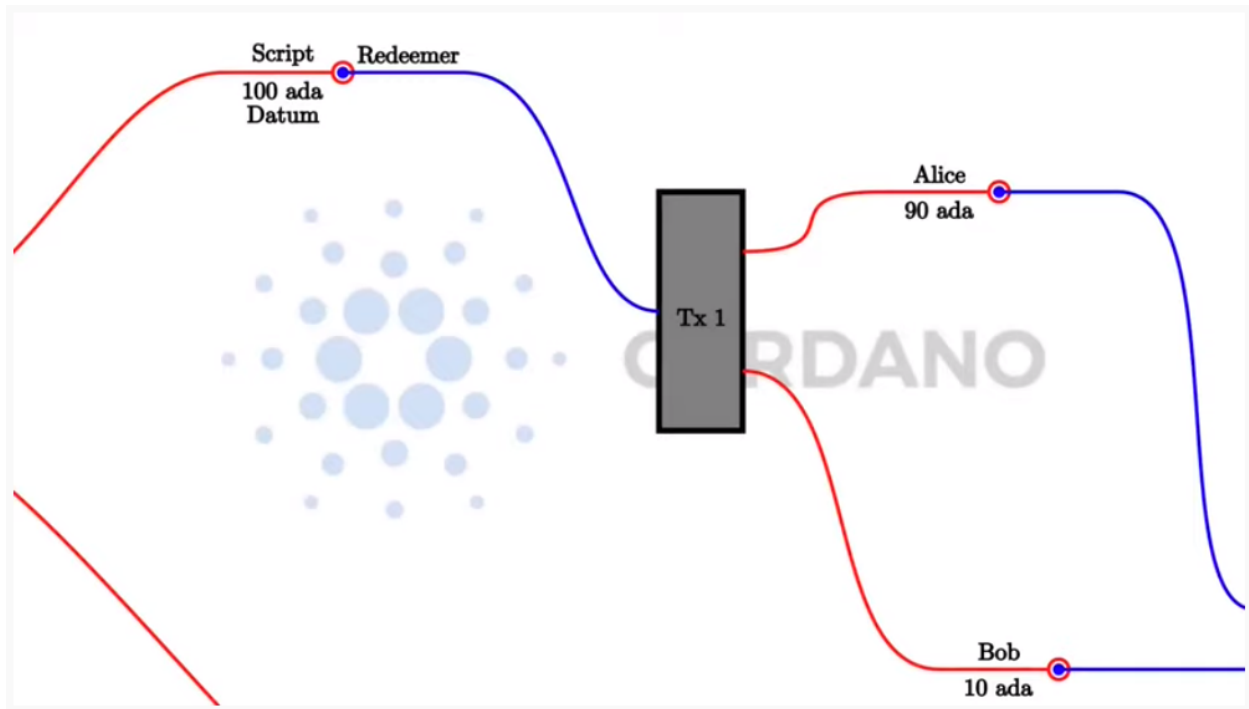
We saw in the first lecture that there are two sides to a smart contract - an on-chain part and an off-chain part.

The on-chain part is about validation. It allows nodes to validate a given transaction and whether it is allowed to consume a given UTxO.

The off-chain part lives in the user's wallet. It constructs and submits suitable transactions.

Both are important topics. We have to master both in order to write smart contracts, but for now we will concentrate on the on-chain part.

Let's recall the Extended UTxO model where the idea is that we introduce a new type of address.



In the simple UTxO model are so-called public key addresses, where the address is given by the hash of the public key. If a UTxO sits at such a public key address, then a transaction can consume that UTxO as an input if the signature belonging to that public key is included in the transaction.

What the (E)UTxO model does is extend this by adding script addresses that can run arbitrary logic.

When a transaction wants to consume a UTxO sitting at a script address is validated by a node, the node will run the script and then, depending on the result of the script, decide whether the transaction is valid or not.

And recall that two were three more additions:

1. Instead of just having signatures on transactions, we have so-called Redeemers - arbitrary pieces of data.
2. On the UTxO output side, we have an additional arbitrary piece of data called Datum, which you can think of as a little piece of state that sits on the UTxO.

Finally, we have the context. There are various choices of what this context can be. It can be very restrictive, consisting just of the Redeemer (as in Bitcoin), or very global, consisting of the whole state of the blockchain (as in Ethereum). In Cardano, it is the transaction that is being validated, including all its inputs and outputs.

So, there are three pieces of data that a Plutus script gets. The Datum, sitting at the UTxO, the redeemer, coming from the input and the validation, and the context, consisting of the transaction being validated and its inputs and outputs.

In a concrete implementation like Plutus, these pieces of information need to be represented by a concrete data type - a Haskell data type. As it happens, the choice was made to use the same data type for all three of them. At least at the low-level implementation.

We will look at that first, but in real life nobody would actually use this low-level approach. There are more convenient ways to use more suitable data types for these things, and we will come to that later in this lecture.

## 2.3 PlutusTx.Data

As mentioned, the datum, redeemer and context share a data type.

That data type is defined in the package `plutus-core`, in the module `PlutusCore.Data`.

It is called, simply, `Data`.

```
data Data =
  Constr Integer [Data]
  | Map [(Data, Data)]
  | List [Data]
  | I Integer
  | B BS.ByteString
deriving stock (Show, Eq, Ord, Generic)
deriving anyclass (NFData)
```

It has five constructors.

- `Constr` takes an `Integer` and, recursively, a list of `Data`
- `Map` takes a list of pairs of `Data`. You can think of this as a lookup table of key-value pairs where both the key and the value are of type `Data`
- `List` takes a list of `Data`
- `I` takes a single `Integer`
- `B` takes a `ByteString`

For those familiar with the JSON format, this is very similar. The constructors are not exactly the same, but, like JSON, you can represent numbers, strings, lists of data and key-value pairs. It can represent arbitrary data, which makes it very suitable for our purpose.

We can also explore this type in the REPL.

Run the following from the `plutus-pioneers-program` repository. You may need to start a `nix-shell` from the `Plutus` repository before changing into the `week02` directory.

```
cd code/week02
cabal repl
```

From with the REPL, we need to import `PlutusTx` so that we have access to the `Data` type. `Data` is not defined in `PlutusTx`, but it gets re-exported from there.

```
import PlutusTx
```

We can now get some information about `Data`.

```
:i Data
```

This will give information about the type `Data`.

```
Prelude Week02.Burn> import PlutusTx
Prelude PlutusTx Week02.Burn> :i Data
type Data :: *
data Data
  = Constr Integer [Data]
  | Map [(Data, Data)]
  | List [Data]
  | I Integer
  | B bytestring-0.10.12.0:Data.ByteString.Internal.ByteString
  -- Defined in 'plutus-core-0.1.0.0:PlutusCore.Data'
instance Eq Data
  -- Defined in 'plutus-core-0.1.0.0:PlutusCore.Data'
instance Ord Data
  -- Defined in 'plutus-core-0.1.0.0:PlutusCore.Data'
instance Show Data
  -- Defined in 'plutus-core-0.1.0.0:PlutusCore.Data'
instance IsData Data -- Defined in 'PlutusTx.IsData.Class'
```

Now we can play with it. We can use the I constructor to create a value of type Data.

```
Prelude PlutusTx.Data Week02.Burn> I 42
I 42
```

We can ask for its type, and confirm that it is indeed of type Data:

```
Prelude PlutusTx.Data Week02.Burn> :t I 42
I 42 :: Data
```

The easiest way to create a value of type Data using the B constructor is to use the GHC Extension OverloadedStrings. This allows literal strings to be used in place of string-like data types and the compiler will interpret them as their intended type.

```
Prelude PlutusTx.Data Week02.Burn> :set -XOverloadedStrings
Prelude PlutusTx.Data Week02.Burn> :t B "Haskell"
B "Haskell" :: Data
```

We can also use more complicated constructors, like Map and List:

```
Prelude PlutusTx.Data Week02.Burn> :t Map [(I 42, B "Haskell"), (List [I 0], I 1000)]
Map [(I 42, B "Haskell"), (List [I 0], I 1000)] :: Data
```

## 2.4 Plutus Validator

Now we are ready to implement our very first validator.

### 2.4.1 Example 1 - The Gift Contract

#### The Code

We start the script by copy pasting a list of GHC language extensions, plus some dependency imports from the example we used in the last lecture.

```
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE FlexibleContexts     #-}
...

module Week02.Gift where

import      Control.Monad      hiding (fmap)
import      Data.Map           as Map
...
import      Text.Printf        (printf)
```

Then, we write the validator. Ultimately, the validator will be a script, living on the blockchain in Plutus Core, which is a lower-level language based on the lambda calculus. But, we don't have to write Plutus Core. We can write Haskell and we will see later how we convert that Haskell into Plutus Core script.

So, we write a Haskell function that represents our validator. As we know, a validator is a script that takes three pieces of input - the datum, the redeemer and the context, respectively, which, at the lowest level are represented by the `Data` data type.

```
mkValidator :: Data -> Data -> Data -> ()
```

Somewhat surprisingly, the result of the function is `()`. This is the Haskell `Unit` type, similar to `void` in some other languages, like C or C# or Java - it's the type that carries no information.

`Unit` is a built-in type in Haskell and it has just one value, which is written in the same way as the type itself, as we can see from the REPL.

```
Prelude Week02.Gift> ()
()
Prelude Week02.Gift> :t ()
() :: ()
```

A function with a return type of `()` is quite unusual in Haskell. In more mainstream languages, it is quite common for functions or procedures to return no value. In these situations, the functions are only important for their side-effects, such as a Java function that prints something to the console.

But Haskell is a pure language. If you want side-effects, this will be shown by the type system. For example if the `mkValidator` were to perform any IO, it would have a type signature of:

```
mkValidator :: Data -> Data -> Data -> IO ()
```

This would indicate a function that performs IO side-effects but has no interesting return value.

But, as we know that the real `mkValidator` function performs no side-effects and returns no value, there is really nothing useful that it can do.

However, there is something that the function can do as well as returning `()`, namely it can throw an exception or have an error. And that's what Plutus uses.

The idea is that if the `mkValidator` function does not run into an error or throw an exception, then validation succeeds. If it throws an error then validation fails and the transaction is rejected.

Let's write the simplest validator that we can.

```
mkValidator :: Data -> Data -> Data -> ()
mkValidator _ _ _ = ()
```

The first argument is the datum, the second argument is the redeemer and the third argument is the context. The most simple thing we can do is to completely ignore all three arguments and immediately return `()`.

What this means is that the script address that corresponds to this validator doesn't care about the datum, it doesn't care about the redeemer, and it doesn't care about the Context. It will always succeed, and this means that any transaction can consume the script at this address as an input. It does not matter what datum exists for a UTxO at this script address, it doesn't matter which redeemer is used for the transaction and it doesn't matter what structure the transaction has.

If you send any funds to this script address, anybody can immediately take it.

This function is not yet Plutus code, it is just a Haskell function. In order to turn it into a Plutus script, we need to compile it.

The result of our compilation to Plutus will be of type `Validator`. Below the function in `Gift.hs`, we add the following code.

```
validator :: Validator
validator = mkValidatorScript $$ (PlutusTx.compile [| mkValidator |])
```

The `mkValidatorScript` function takes the type `CompiledCode (Data -> Data -> Data -> ()) -> Validator`. In order to create this type, we must compile the `mkValidator` script using something called Template Haskell.

Template Haskell is an advanced feature of Haskell that solves a similar problem as macro systems in other languages. A macro being something that gets expanded at compile time.

So, with this code

```
$(PlutusTx.compile [| mkValidator |])
```

We are asking the compiler to write the code for the `validator` function at compile time based on our `mkValidator` function, and then proceed with the normal compilation.

You do not need to understand very much about Template Haskell to write Plutus as it is always the same pattern. Once you have seen a couple of examples, you can more or less just copy and paste.

Template Haskell expects all the code to be available within the Oxford Brackets - `[| |]`.

With more complicated validators you will likely be relying on multiple helper functions, and you do not want to have to add them within the Oxford Brackets. To avoid this, there is one thing we need to do to the `mkValidator` function, and that is to make it inlinable by adding the `INLINABLE` pragma.

```
{-# INLINABLE mkValidator #-}
mkValidator :: Data -> Data -> Data -> ()
mkValidator _ _ _ = ()
```



You will see this often in Plutus scripts, and it is usually an indication that a function is meant to be used within a validation script. All the functions on which the validator depends must be inlinable.

Let's go back to the REPL and take a look at the validator.

```
:l src/Week02/Gift.hs
Ok, one module loaded.
Prelude PlutusTx Week02.Gift> import Ledger.Scripts
Prelude PlutusTx Ledger.Scripts Week02.Gift> validator
Validator { <script> }
```

We can ask for information about Validator.

```
Prelude PlutusTx Ledger.Scripts Week02.Gift> :i Validator
type Validator :: *
newtype Validator = Validator {getValidator :: Script}
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
instance Eq Validator
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
instance Ord Validator
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
instance Show Validator
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
```

We see that it is a wrapper around getValidator

```
Prelude PlutusTx Ledger.Scripts Week02.Gift> getValidator validator
<Script>
```

We can then get some information about Script

```
Prelude PlutusTx Ledger.Scripts Week02.Gift> :i Script
type Script :: *
newtype Script
    = Script {unScript :: plutus-core-0.1.0.0:UntypedPlutusCore.Core.Type.Program
                plutus-core-0.1.0.0:PlutusCore.DeBruijn.Internal.DeBruijn
                plutus-core-0.1.0.0:PlutusCore.Default.Universe.DefaultUni
                plutus-core-0.1.0.0:PlutusCore.Default.Builtins.DefaultFun
                ()}
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
instance Eq Script
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
instance Ord Script
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
instance Show Script
    -- Defined in 'plutus-ledger-api-0.1.0.0:Plutus.V1.Ledger.Scripts'
```

And here we see that we have an unScript function, which we can run

```
Prelude PlutusTx Ledger.Scripts Week02.Gift> unScript $ getValidator validator
Program () (Version () 1 0 0) (Apply () (Apply () (LamAbs () (DeBruijn {dbnIndex = 0}))
↳ (LamAbs () (DeBruijn {dbnIndex = 0}) (Apply () (Apply () (Apply () (LamAbs ()
↳ (DeBruijn {dbnIndex = 0}) (LamAbs () (DeBruijn {dbnIndex = 0}) (LamAbs () (DeBruijn
↳ {dbnIndex = 0}) (Apply () (Apply () (Apply () (Apply () (Apply () (Apply () (LamAbs ()
↳ (DeBruijn {dbnIndex = 0}) (LamAbs () (DeBruijn {dbnIndex = 0}) (LamAbs () (DeBruijn
↳ {dbnIndex = 0}) (LamAbs () (DeBruijn {dbnIndex = 0}) (LamAbs () (DeBruijn {dbnIndex =
↳ 0}) (LamAbs () (DeBruijn {dbnIndex = 0}) (Apply () (Apply () (LamAbs () (DeBruijn
↳ {dbnIndex = 0}) (LamAbs () (DeBruijn {dbnIndex = 0}) (Apply () (LamAbs () (DeBruijn
↳ {dbnIndex = 0}) (LamAbs () (DeBruijn {dbnIndex = 0}) (Var () (DeBruijn {dbnIndex = 1}))
↳ (LamAbs () (DeBruijn {dbnIndex = 0}) (LamAbs () (DeBruijn {dbnIndex = 0}) (Var ()
↳ (DeBruijn {dbnIndex = 5}))))))))) (Delay () (LamAbs () (DeBruijn {dbnIndex = 0}) (Var
↳ () (DeBruijn {dbnIndex = 1})))))) (LamAbs () (DeBruijn {dbnIndex = 0}) (Var ()
```

(continued from previous page)

And here you can see an honest-to-goodness representation of the Plutus Core script for the validator.

Back to the code.

Now we have our first validator, there are two more types that we can define.

One is the `ValidatorHash`, which, as the name suggests is the hash of the validator.

```
valHash :: Ledger.ValidatorHash
valHash = Scripts.validatorHash validator
```

And, we can also turn the validator into a script address, which is the script's address on the blockchain.

```
scrAddress :: Ledger.Address
scrAddress = ScriptAddress valHash
```

Now we have a script address represented as `scrAddress`.

We can look at these two results in the REPL

```
Prelude PlutusTx Ledger.Scripts Week02.Gift> valHash
c3168d465a84b7f50c2eeb51ccacd53a305bd7883787adb54236d8d17535ca14

Prelude PlutusTx Ledger.Scripts Week02.Gift> scrAddress
Address {addressCredential = ScriptCredential_
↳ c3168d465a84b7f50c2eeb51ccacd53a305bd7883787adb54236d8d17535ca14, _
↳ addressStakingCredential = Nothing}
```

With the exception of the `mkValidator` function logic (in our case, one line), the rest of the code we have written so far is boilerplate and will be very similar for all Plutus scripts.

In order to actually try this script, we need wallet code. The focus of this lecture is validation and not wallet code, but briefly, here is the rest of the code.

Two endpoints are defined. Endpoints are ways for a user to trigger something with input parameters.

The `give` endpoint will take an `Integer` argument to specify the number of lovelace that will be deposited to the contract.

The `grab` endpoint will take no argument and will simply look for UTxOs at this script address and consume them.

```
type GiftSchema =
    Endpoint "give" Integer
  .\ Endpoint "grab" ()
```

The `give` endpoint uses the helper function `mustPayToOtherScript` which takes the `valHash` of the recipient script and a `Datum` that, in this example, is completely ignored. It uses the `Datum` constructor to turn a `Data` into a `Datum`. In this case the `Data` is created using the `Constr` constructor taking a 0 and an empty list.

Finally the amount to send to the address is specified using the helper function `Ada.lovelaceValueOf`.

The transaction is then submitted, the script waits for it to be confirmed and then prints a log message.

```
give :: AsContractError e => Integer -> Contract w s e ()
give amount = do
    let tx = mustPayToOtherScript valHash (Datum $ Constr 0 []) $ Ada.lovelaceValueOf_
↳ amount
```

(continues on next page)

(continued from previous page)

```

ledgerTx <- submitTx tx
void $ awaitTxConfirmed $ txId ledgerTx
logInfo @String $ printf "made a gift of %d lovelace" amount

```

The grab endpoint is a little bit more complicated.

We use `utxoAt` with our new script address `scrAddress` to lookup all the UTxOs sitting at that address. We then need lookups, which will be used by the wallet to construct the transaction. Here, we tell the wallet where to find all the UTxOs, and we inform it about the validator. Remember, if you want to consume a UTxO sitting at a script address, then the spending transaction needs to provide the validator code, whereas the transaction that produces the UTxO only needs to provide the hash.

We then define the transaction by using `mustSpendScriptOutput` for each UTxO found. This is saying that every UTxO sitting at this script address must be spent by the transaction we are constructing.

We also pass a redeemer which is completely ignored in our example, so we can put anything there - in this case a redeemer created using the `I` constructor of type `Data` with a value of 17.

Again, we submit, wait for confirmation, and then write a log message.

```

grab :: forall w s e. AsContractError e => Contract w s e ()
grab = do
  utxos <- utxoAt scrAddress
  let orefs = fst <$> Map.toList utxos
      lookups = Constraints.unspentOutputs utxos          <>
                Constraints.otherScript validator
      tx :: TxConstraints Void Void
      tx = mconcat [mustSpendScriptOutput oref $ Redeemer $ I 17 | oref <- orefs]
  ledgerTx <- submitTxConstraintsWith @Void lookups tx
  void $ awaitTxConfirmed $ txId ledgerTx
  logInfo @String $ "collected gifts"

```

Finally, we put it all together in the `endpoints` function. This is boilerplate code that is telling the wallet to give the option of certain endpoints to the user and then, once one has been selected, to recurse and continue to offer the same options again and again. In the case of `give` the user will be required to provide the `Integer` argument.

```

endpoints :: Contract () GiftSchema Text ()
endpoints = (give' `select` grab') >> endpoints
where
  give' = endpoint @"give" >>= give
  grab' = endpoint @"grab" >> grab

```

Then we have a little piece of boilerplate.

```
mkSchemaDefinitions 'GiftSchema
```

And then some code that is used only by the Plutus Playground which allows us to specify additional tokens that can be used for testing.

```
mkKnownCurrencies []
```

### Testing

We will now test the Gift script in the playground.

Copy the Gift script into the playground, then compile the script in the playground and press the **Simulate** button.

The screenshot shows the Plutus Playground Simulator interface. At the top, there's a header with the Plutus Playground logo and navigation links: Getting Started, Tutorial, API, Privacy. Below the header, there's a sub-header with Demo files: Hello\_world, Starter, Game, Vesting, Crowd Funding, Error Handling, and a Log In button. The main area is titled "Simulator" and contains a "Simulation 1" tab. Under "Wallets", there are two wallet cards: "Wallet 1" and "Wallet 2". Each wallet card has an "Opening Balances" section with a "Lovelace" input field set to "10". Below this is an "Available functions" section with buttons for "give", "grab", and "Pay to Wallet". To the right of the wallets is an "Add Wallet" button. Below the wallets is an "Actions" section with an "Add Wait Action" button. At the bottom right of the simulator area are "Evaluate" and "Transactions" buttons.

And let's add a third wallet and give all the wallets 10 Ada (10 million lovelace).

The screenshot shows the Plutus Playground Simulator interface after adding a third wallet. The header and sub-header are the same as in the previous screenshot. The main area is titled "Simulator" and contains a "Simulation 1" tab. Under "Wallets", there are now three wallet cards: "Wallet 1", "Wallet 2", and "Wallet 3". Each wallet card has an "Opening Balances" section with a "Lovelace" input field set to "10000000". Below this is an "Available functions" section with buttons for "give", "grab", and "Pay to Wallet". To the right of the wallets is an "Add Wallet" button. Below the wallets is an "Actions" section with an "Add Wait Action" button. At the bottom right of the simulator area are "Evaluate" and "Transactions" buttons.

We will create a scenario where wallets 1 and 2 give lovelace, and wallet 3 grabs all of it.

You will see that the playground has rendered UI buttons for the two endpoints **give** and **grab**. Use the **give** endpoint for to make wallet 1 give 4 Ada and to make wallet 2 give 6 Ada. Then add a wait action to wait for 1 block, and then use to **grab** endpoint to make wallet 3 grab the funds. Then add another wait action to wait for 1 block.

And now click **Evaluate**. We see that there have been four transactions.

The first transaction is, as always, the genesis transaction that distributes the initial funds to the wallets.

## Wallets

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

Evaluate

Transactions

Wallet 1

Opening Balances

Lovelace

Available functions

give +

grab +

Pay to Wallet +

Wallet 2

Opening Balances

Lovelace

Available functions

give +

grab +

Pay to Wallet +

Wallet 3

Opening Balances

Lovelace

Available functions

give +

grab +

Pay to Wallet +

+

Add Wallet

## Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: give

4000000

✓

2

Wallet 2: give

6000000

✓

3

Wait

☒ Wait For...
 ☐ Wait Until...

Blocks

4

Wallet 3: grab

5

Wait

☒ Wait For...
 ☐ Wait Until...

Blocks

+

Add Wait Action

Evaluate

Transactions

## Simulator

[Return to Editor](#)

Simulation 1

+

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 1, Tx 1

Inputs

Transaction

Slot 0, Tx 0

Tx: 2a6418b7774bde70711578aa02de3b4941f9596b255cbd599fd88de8bb324f11

Validity: All time

Signatures: None

Forge

Ada

Lovelace

30,000,000

Outputs

Wallet 2

PubKeyHash 39f713d0a644253f04529421b9f51b9...

Ada

Lovelace

10,000,000

Spent in: Slot 1, Tx 0

Wallet 1

PubKeyHash 21fe31dfa154a261626bf854046d227...

Ada

Lovelace

10,000,000

Spent in: Slot 1, Tx 1

Wallet 3

PubKeyHash dac073e0123bdea59d49b3bda9cf603...

Ada

Lovelace

10,000,000

Spent in: Slot 1, Tx 1

2.4. Plutus Validator

41

And there are two transactions which occur at slot 1. They are the two **give** transactions.

The first one, Tx 0, is from wallet 2. The order here is not determined by the order that we created the transactions in the simulator. The important thing to note is that both **give** transactions occurred at the same slot.

We see the three outputs. The first output is the 10 lovelace fee paid by wallet 2. The second output is the 6 Ada sent to the script address, and the third output is the returning of the change to wallet 2, which is 4 Ada minus the fees.

**Simulator** [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0

Slot 1, Tx 1

#### Inputs

Wallet 2	
PubKeyHash 39f713d0a644253f04529421b9f51b9...	
Ada Lovelace	10,000,000
Created by: Slot 0, Tx 0	

#### Transaction

Slot 1, Tx 0

**Tx:** 9ba3da7a39cbb84643e0625d5f2c5a5c42c75e2f8e7b1e2e3294d91f5b9ac

**Validity:** All time

**Signatures:**

- PubKey 3d4017c3e843895a92b70aa74d1b7ebc982cc2ec4968cc0cd55f12af4660c

#### Outputs

Fee	
Ada Lovelace	10

Wallet 2	
PubKeyHash 39f713d0a644253f04529421b9f51b9...	
Ada Lovelace	3,999,990
Unspent	

Script c3168d465a84b7f50c2eeb51ccacd...	
Ada Lovelace	6,000,000
Unspent	

Balances Carried Forward (as at Slot 1, Tx 0)

And the second, Tx 1, is from wallet 1. Again, with similar output UTxOs.

We now have two UTxOs sitting at the script address.

Then we have the **grab** at slot 2 triggered by wallet 3. We see the two UTxOs from the script as inputs, and then two outputs. One output is the fees and the other is the output, paid to wallet 3, is of 10 Ada minus those fees. You'll notice that the fees are now higher than we saw before, and this is because a script has now been executed, which makes it more expensive. However, the fees here are not yet entirely calibrated with those that would be charged on the real blockchain.

And, by scrolling down, we see the final wallet balances.

If you were to scroll down further you would see some traces and log outputs that would give more detail about the execution.

As mentioned, this script uses the simplest validator possible, one that always succeeds. But this silly little validator may be useful in a situation where someone wants to donate some lovelace to the community and leave it up for grabs!

## Simulator

[< Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain  
Click a transaction for details

Slot 0, Tx 0   Slot 1, Tx 0   Slot 2, Tx 0  
Slot 1, Tx 1

**Inputs**

Wallet 1	
PubKeyHash 21fe31dfa154a261626bf854046fd227...	
Ada Lovelace	10,000,000
Created by: Slot 0, Tx 0	

**Transaction**

Slot 1, Tx 1	
Tx: 0ec3e16ab43b550124169d1b40d24eabd05f586818d72830ef849bd6d804c031	
Validity: All time	
Signatures:	
• PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a	

**Outputs**

Fee	
Ada Lovelace	10
Wallet 1	
PubKeyHash 21fe31dfa154a261626bf854046fd227...	
Ada Lovelace	5,999,990
Unspent	
Script c3168d465a84b7f50c2eeb51ccacd...	
Ada Lovelace	4,000,000
Spent in: Slot 2, Tx 0	

Balances Carried Forward (as at Slot 1, Tx 1)

## Simulator

[< Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain  
Click a transaction for details

Slot 0, Tx 0   Slot 1, Tx 0   Slot 2, Tx 0  
Slot 1, Tx 1

**Inputs**

Script c3168d465a84b7f50c2eeb51ccacd...	
Ada Lovelace	4,000,000
Created by: Slot 1, Tx 1	
Script c3168d465a84b7f50c2eeb51ccacd...	
Ada Lovelace	6,000,000
Created by: Slot 1, Tx 0	

**Transaction**

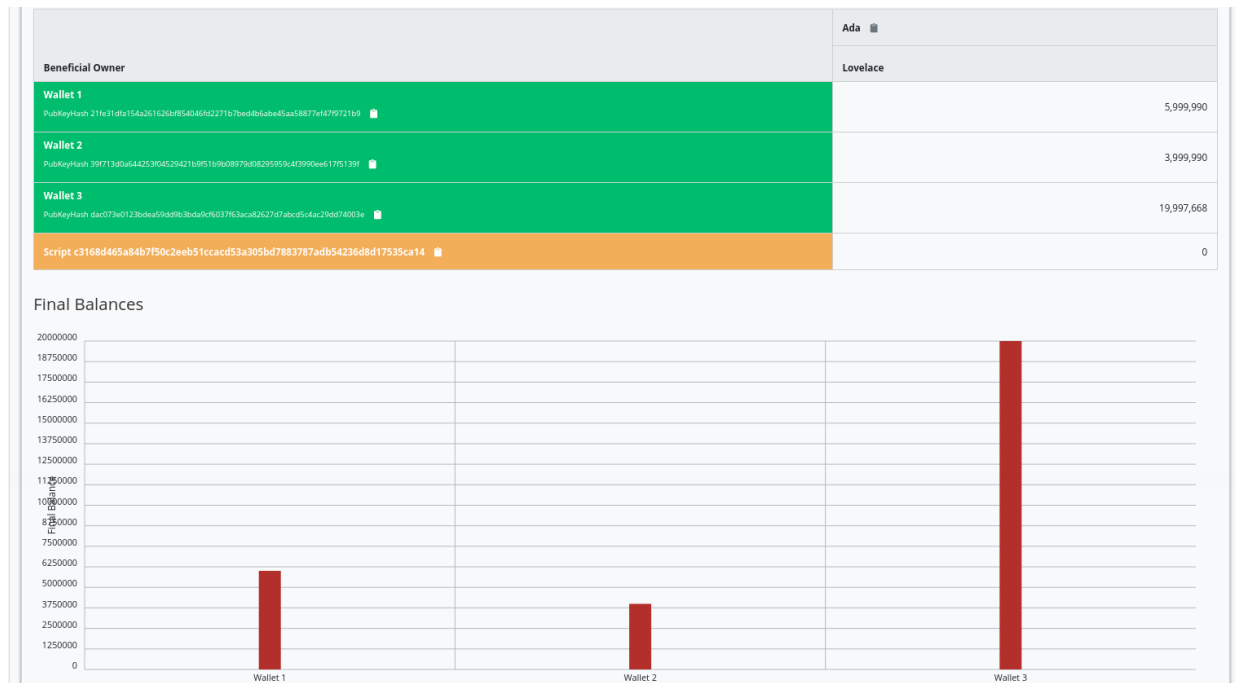
Slot 2, Tx 0	
Tx: d364757ba826bf1cbdc6368a97e4dd24f92284e7acb2ceeb5a727895cdde452	
Validity: All time	
Signatures:	
• PubKey fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025	

**Outputs**

Fee	
Ada Lovelace	2,332
Wallet 3	
PubKeyHash dac073e0123bdea59dd9b3bda9cf603...	
Ada Lovelace	9,997,668
Unspent	

Balances Carried Forward (as at Slot 2, Tx 0)

Beneficial Owner	Ada
Wallet 1	Lovelace



## 2.4.2 Example 2 - Burn

Let's look at the second example of validation.

We will start by copying the `Gift.hs` code and renaming it `Burn.hs`.

In the `Gift` example we had a validator that would always succeed. In this example, we want to do the opposite - a validator that always fails.

Recall that a validator indicates failure by throwing an error. So we can modify our validator accordingly.

```
mkValidator :: Data -> Data -> Data -> ()
mkValidator _ _ _ = error ()
```

If we load the module in the REPL and look at `error`

```
Prelude Week02.Burn> :t error
error :: [Char] -> a
```

We see the definition for the `error` function defined in the standard Haskell Prelude. However, the one in scope in our code is in fact the following `error` function.

```
Prelude Week02.Burn> :t PlutusTx.Prelude.error
PlutusTx.Prelude.error :: () -> a
```

In regular Haskell, you have the `error` function which takes an error message string and triggers an error. In Plutus, the `error` function does not take a string - it just takes `()` and returns an arbitrary type.

And that takes us to an important point.

We mentioned earlier that we use the `INLINABLE` pragma on the `mkValidator` function in order to allow it to be used by the Template Haskell code. In Haskell there are many functions available via the `Prelude` module, but these will not be usable in Plutus as they are not defined as inlinable. So, the Plutus team have provided an alternative Prelude that can be used in validation.



The way that the Plutus Prelude is able to take precedence over the Haskell Prelude, which is normally in scope by default, is by using the following LANGUAGE pragma in the code.

```
{-# LANGUAGE NoImplicitPrelude #-}
```

Then, by importing `PlutusTx.Prelude`, its functions are used in place of the standard Prelude functions.

```
import PlutusTx.Prelude hiding (Semigroup(..), unless)
```

You may also notice that the standard Prelude is also imported. However, it is only in order to bring in some functions that have nothing to do with validation but is for the off-chain code and the playground.

```
import Prelude (IO, Semigroup(..), String)
```

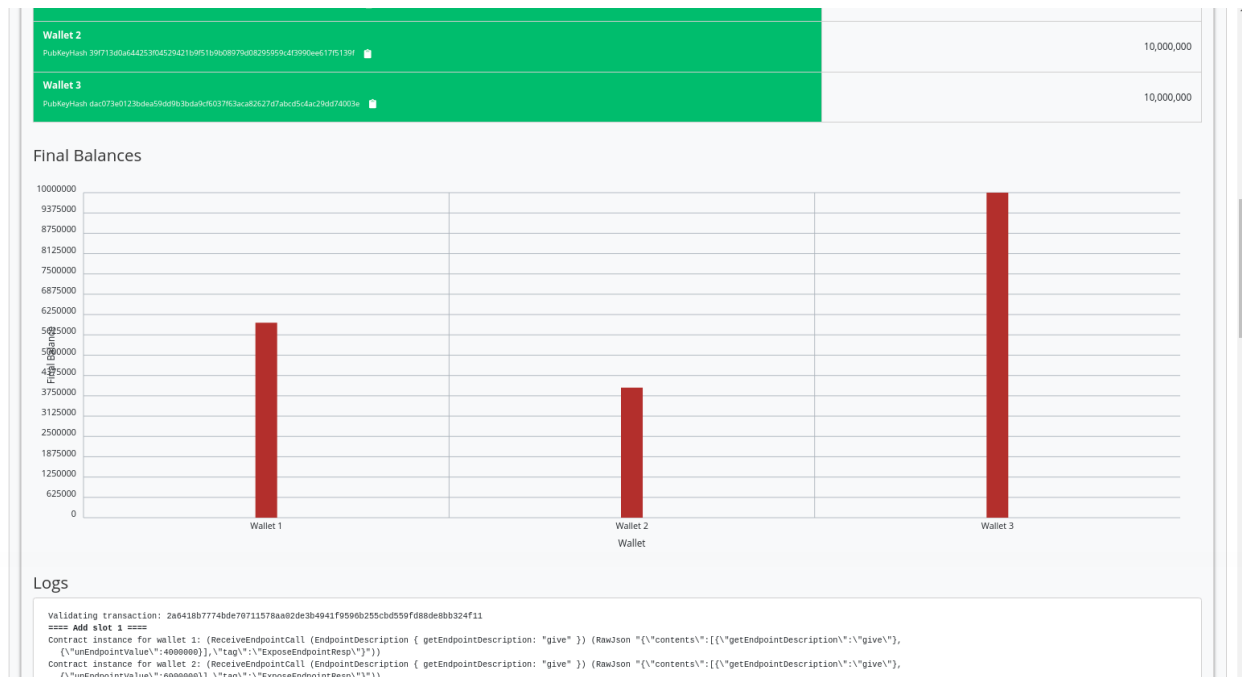
It can be confusing. A lot of the functions in the Plutus Prelude do have the same signatures and same behaviour as their namesakes in the standard Prelude, but that is not always the case, and `error` is an example.

Just remember that when you are using something in a Plutus script that looks like a function from the standard Prelude, what you are actually using is a function from the Plutus Prelude. Often they will have the same signature, but they are not always identical - for example operator precedents may not be the same

Looking again at our new validator, we now have a validator that will always fail.

```
mkValidator :: Data -> Data -> Data -> ()
mkValidator _ _ _ = error ()
```

We will leave everything else as it was and check the effect of this change, using the playground. After clicking Compile, the previous scenario should still be present in the simulator. And after clicking Evaluate and scrolling down a little, we can see that wallets 1 and 2 have made their gifts but wallet 3 has been unable to grab.



If we scroll down further, we will find a log message showing that validation failed.

```
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Contract instance stopped with error: "WalletError (ValidationError (ScriptFailure_
↳ (EvaluationError [])))" ]
```

So, in our first example we had a validator that would always succeed and would allow anyone to grab the UTxOs from it. In the second example, we have a validator that always fails and any UTxOs sent to this script address can never be retrieved. This is basically a way to burn funds, which may be useful under some circumstances.

When we look at the logs, we see that validation fails, but we have no clue why it fails. here's a way to change that by using a variant of error - `traceError`.

```
Prelude Week02.Burn> :t PlutusTx.Prelude.traceError
PlutusTx.Prelude.traceError :: PlutusTx.Builtins.String -> a
```

The function takes a string, but not a Haskell string. It is a Plutus string. In order for this to compile, we need to use the `OverloadedStrings` GHC extension.

```
{-# LANGUAGE OverloadedStrings #-}
```

Then, we can update our validator.

```
mkValidator _ _ = traceError "BURNT!"
```

If we now run the same scenario in the playground with the new code, we will see the custom error message that we added.

```
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Contract instance stopped with error: "WalletError (ValidationError (ScriptFailure_
↳ (EvaluationError ["BURNT!"])))" ]
```

```
, Slot 2: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Handled request: Iteration 3 request ID 1
  Response: "{ \"contents\": { \"getTxId\": { \"a\": \"a5d35c450234b59b400\" } } }"
, Slot 2: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Current requests (2): Iteration 4 request ID 2
  Request: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
  Iteration 4 request ID 1
  Request: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
, Slot 2: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  No requests handled
, Slot 2: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  No requests handled
, Slot 3: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  No requests handled
, Slot 1: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Contract instance started
, Slot 1: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Current requests (2): Iteration 1 request ID 2
  Request: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
  Iteration 1 request ID 1
  Request: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
, Slot 1: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  No requests handled
, Slot 1: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  No requests handled
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  No requests handled
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Receive endpoint call on 'grab' for Object (fromList [\"contents\",Array (Object (fromList [\"getEndpointDescription\",String \"grab\"]),Object (fromList [\"unEndpointValue\",Array []]))]),(\"tag\",String \"ExposedEndpointResp\")
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Handled request: Iteration 1 request ID 2
  Response: "{ \"contents\": { \"getEndpointDescription\": { \"a\": \"grab\" } } }"
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Current requests (1): Iteration 2 request ID 1
  Request: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Handled request: Iteration 2 request ID 1
  Response: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Current requests (1): Iteration 3 request ID 1
  Request: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Handled request: Iteration 3 request ID 1
  Response: "{ \"contents\": { \"a\": \"a5d35c450234b59b400\" } }"
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Current requests (0):
, Slot 2: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 3}:
  Contract instance stopped with error: "WalletError (ValidationError (ScriptFailure (EvaluationError [\"BURNT!\"])))"
```

### 2.4.3 Example 3 - Forty Two

For the next example, we will write a validator that does not completely ignore all its arguments. We'll write one that expects a simple redeemer.

Now that we care about the redeemer, we need to be able to reference it. Let's call it `r`.

```
{-# INLINABLE mkValidator #-}
mkValidator :: Data -> Data -> Data -> ()
mkValidator _ r _
```

Let's say that we want validation to pass if the redeemer is `I 42`.

```
{-# INLINABLE mkValidator #-}
mkValidator :: Data -> Data -> Data -> ()
mkValidator _ r _
  | r == I 42 = ()
  | otherwise = traceError "wrong redeemer"
```

If we were to run this now in the playground, validation would always fail. We need to modify the off-chain code to add an input to the `grab` endpoint so that wallet 3 can pass in an `Integer` which we can then pass to the validator as the redeemer.

```
type GiftSchema =
  Endpoint "give" Integer
  .\ Endpoint "grab" Integer
```

We add the redeemer argument to the `grab` declaration. Note the addition of the `Integer` in the function signature, as well as the new `n` parameter which is used to reference it.

```
grab :: forall w s e. AsContractError e => Integer -> Contract w s e ()
grab n = do
```

We can then pass it to the `mustSpendScriptOutput` function instead of the throw-away value we used earlier.

```
tx = mconcat [mustSpendScriptOutput oref $ Redeemer $ I n | oref <- orefs]
```

One more change, we need to change the `>>` to `>>=` in the following code, now that `grab` has an argument. You can use the REPL to look at the types `>>` and `>>=` to see why the second one is now needed. Basically, they both sequence actions, but `>>` ignores any wrapped values, whereas `>>=` accesses the wrapped value and passes it to the next action.

```
grab' = endpoint @"grab" >>= grab
```

Now we can try it out in the playground. After adding the new code and clicking `Simulate` you will notice that the old scenario has gone. That is because the endpoints have changed and the old scenario is no longer valid.

Let's set up a scenario that uses just two wallets. Wallet one is going to give 3 Ada to the contract, and wallet 2 is going to try to grab them, but this time, wallet 2 will need to pass in a value which will be used to construct the redeemer.

For our first attempt, we will add the wrong redeemer value, in this case 100.

If we click `Evaluate`, we see that we only have two transactions, and we see that the Ada remains in the script, which shows that wallet 2 failed to grab it.

The final balances also show this.

And, if we look at the trace, we find the error.

**PLUTUS PLAYGROUND**

[Getting Started](#)
[Tutorials](#)
[API](#)
[Privacy](#)

[Demo files](#)
[Hello\\_world](#)
[Starter](#)
[Game](#)
[Vesting](#)
[Crowd Funding](#)
[Error Handling](#)

[Log In](#)

**Simulator**
[Return to Editor](#)

Simulation 1

+

**Wallets**

Evaluate

Transactions

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

Wallet 1

×

Opening Balances

Lovelace

10000000

Available functions

give

+

grab

+

Pay to Wallet

Wallet 2

×

Opening Balances

Lovelace

10000000

Available functions

give

+

grab

+

Pay to Wallet

+

Add Wallet

**Actions**

Evaluate

Transactions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

×

Wallet 1: give

3000000

✓

2

×

Wait

☒ Wait For...
 ☐ Wait Until...

Blocks

1

3

×

Wallet 2: grab

100

✓

4

×

Wait

☒ Wait For...
 ☐ Wait Until...

Blocks

1

+

Add Wait Action

[cardano.org](#)
[iohk.io](#)

© 2020 IOHK Ltd.

[GitHub](#)
[Twitter](#)
[Feedback](#)

**Simulator**
[Return to Editor](#)

Simulation 1

+

**Transactions**

×

**Blockchain**

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

**Inputs**

Wallet 1

PubKeyHash 21fe31dfa154a261626b854046fd227...

Ada

Lovelace

10,000,000

Created by: Slot 0, Tx 0

**Transaction**

Slot 1, Tx 0

Tx: 805244fb32aceb1ba93c0a11c66c16fa795d470803b9812ac2ed8e965c38bc57

Validity: All time

Signatures:
 

- PubKey d75a980182b10ab7d54bfd3c964073a0ee172f3daa62325af021a68f707511a

**Outputs**

Fee

Ada

Lovelace

10

Wallet 1

PubKeyHash 21fe31dfa154a261626b854046fd227...

Ada

Lovelace

6,999,990

Unspent

Script 566d707aae63b287a13cfe5c6cf4d...

Ada

Lovelace

3,000,000

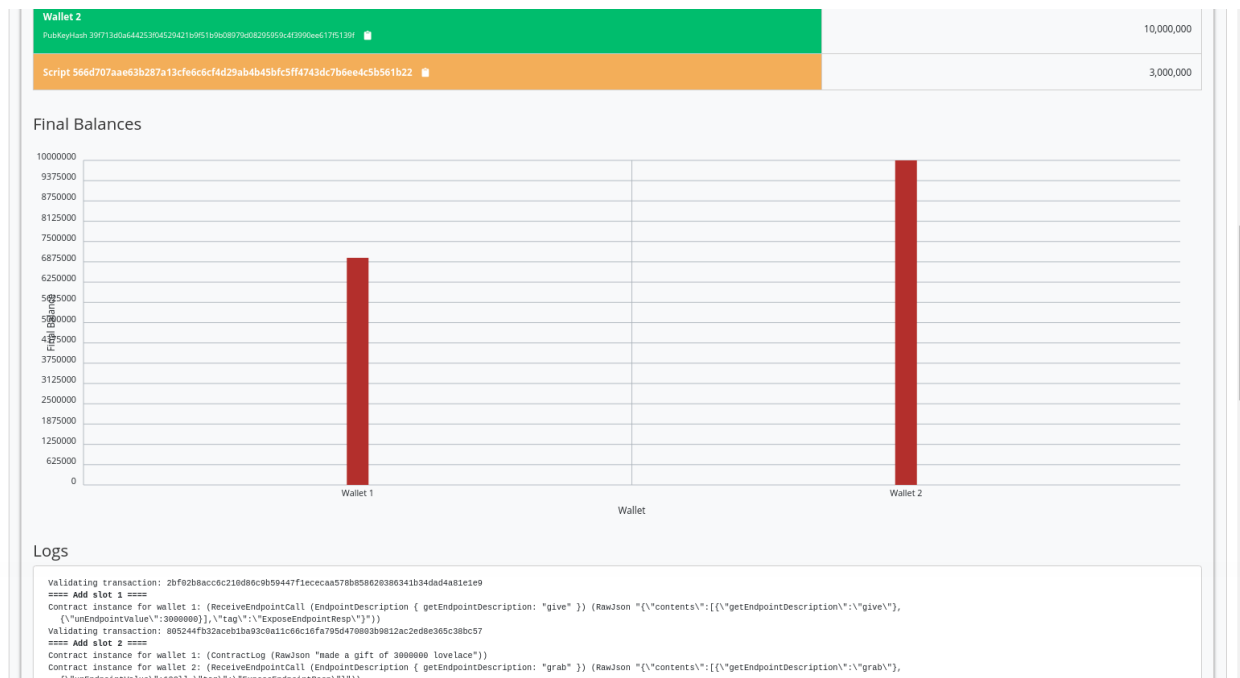
Unspent

**Balances Carried Forward (as at Slot 1, Tx 0)**

Beneficial Owner	Ada
	Lovelace

48

Chapter 2. Week 02 - Validation



```
, Slot 2: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Contract instance stopped with error: "WalletError (ValidationError (ScriptFailure_
  ↳ (EvaluationError [\"wrong redeemer\"])))" ]
```

If we go back to scenario, change the value to 42 and click Evaluate again, we should see that validation succeeds.

Now we see the third transaction where wallet 2 manages to collect the funds, minus fees.

We see that the final balances are as we expect, and also the logs show that validation did not throw an error, which means that validation succeeded.

So that's the first example of a validator that looks at at least one of its arguments.

## 2.4.4 Example 4 - Typed

It was mentioned at the beginning of the lecture, this is low-level Plutus and in reality, no-one will write validation functions like this.

Now we will see how it is actually done using a typed version.

Even though the Data type is powerful and you can encode all sorts of data into it, it doesn't really feel like Haskell. It is almost like you are writing in an untyped language like Javascript or Python. It is just a like a blob of data, it can contain anything so you don't really have any type safety. You will always need to check, for example, if you are expecting an integer that you are indeed given an integer.

It is especially bad with the third argument, the context. Even though it's easy to imagine that you can somehow encode a transaction with its inputs and outputs into the Data type, it is not at all clear how that is done.

We would rather use more specific data types that are tailored to the business logic.

This is indeed possible with so-called Typed Validators. What this means is that we can replace the occurrences of Data in the mkValidator signature with more suitable types.

**Simulator** [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0

#### Inputs

Script 566d707aae63b287a13cfe6c6cf4d...

Ada Lovelace 3,000,000

Created by: Slot 1, Tx 0

#### Transaction

Slot 2, Tx 0

Tx: 7aaae4b2c2ffc893996abcc5003868aff702b6698718997121eb741eedda80

Validity: All time

Signatures:

- PubKey 3d4017c3e843895a92b70aa74d1b7ebc9c982cc2ec4968cc0cd55f12af4660c

#### Outputs

Fee

Ada Lovelace 1,589

Wallet 2

PubKeyHash 39f713d0a644253f04529421b9f51b9...

Ada Lovelace 2,998,411

Unspent

#### Balances Carried Forward (as at Slot 2, Tx 0)

Beneficial Owner	Ada	Lovelace
Wallet 1		6,999,990
Wallet 2		12,998,411

PubKeyHash 39f713d0a644253f04529421b9f51b9...

Script 566d707aae63b287a13cfe6c6cf4d29ab4b45bfc5f4743dc7b6ee4c5b561b22

0

### Final Balances

Wallet	Balance (Lovelace)
Wallet 1	6,999,990
Wallet 2	12,998,411

### Logs

```
Validating transaction: 2bf02baacc6c21d88c9b594471eecca578b858626386341b34dd44a81e1e9
==== Add slot 1 ====
Contract instance for wallet 1: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "give" }) (RawJson "[\"contents\":[[\"getEndpointDescription\\\":\\\"give\\\",
{\\\"unEndpointValue\\\":3000000}],\\\"tag\\\":\\\"ExposeEndpointResp\\\"]"))
Validating transaction: 805244fb32aceb1ba93c8a11c6c16fa795d470803b9812ac2e0e305c38bc57
==== Add slot 2 ====
Contract instance for wallet 1: (ContractLog (RawJson "made a gift of 3000000 lovelace"))
Contract instance for wallet 2: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "grab" }) (RawJson "[\"contents\":[[\"getEndpointDescription\\\":\\\"grab\\\",
{\\\"unEndpointValue\\\":42}],\\\"tag\\\":\\\"ExposeEndpointResp\\\"]"))
Validating transaction: 7aaae4b2c2ffc893996abcc5003868aff702b6698718997121eb741eedda80
==== Add slot 3 ====
```

```
mkValidator :: Data -> Data -> Data -> ()
```

In our silly little example, we completely ignore the Datum, so a more suitable type would be just the Unit type - ().

```
mkValidator :: () -> Data -> Data -> ()
```

For the redeemer, in this example, we are only dealing with integers, so it would probably make more sense to use Integer instead.

```
mkValidator :: () -> Integer -> Data -> ()
```

For the context, there is a much nicer type called ScriptContext that's made exactly for this purpose.

```
mkValidator :: () -> Integer -> ScriptContext -> ()
```

Finally, we have already mentioned that it is a bit unusual to use () as a return type. Much more natural would be to use Bool to indicate successful or failed validation.

```
mkValidator :: () -> Integer -> ScriptContext -> Bool
```

So, this is a better way to write validation code. The last two types ScriptContext and Bool, but the first two types can be different depending on the situation.

In this case, let's now rewrite the function accordingly using these new types. The parameter r is now no longer of type Data - it is an Integer, so we can simply check that it is equal to 42 rather than checking it against a constructed Data type.

And, as we are now returning a Bool, we can we just make the function a boolean expression.

```
{-# INLINABLE mkValidator #-}
mkValidator :: () -> Integer -> ScriptContext -> Bool
mkValidator _ r _ = r == 42
```

This will have the same problem that we had before in that, in the case of an error, we won't get a nice error message. There is a nice Plutus function called traceIfFalse which takes a String and a Bool and returns a Bool. If the first Bool is True, the result will be True and the String is ignored. However, if the first Bool is False, then the result will be False and the String will be logged.

```
PlutusTx.Prelude.traceIfFalse
  :: PlutusTx.Builtins.String -> Bool -> Bool
```

This is exactly what we need.

```
{-# INLINABLE mkValidator #-}
mkValidator :: () -> Integer -> ScriptContext -> Bool
mkValidator _ r _ = traceIfFalse "wrong redeemer" $ r == 42
```

This will not yet compile as other parts of the code are not yet type correct. We need to adapt our boilerplate.

First, we introduce a new dummy data type, which here we call Typed, simply based on the name of the script. For this type we must provide an instance of Scripts.ValidatorTypes.

The purpose of this instance is to declare the types for the datum and the redeemer.

```
data Typed
instance Scripts.ValidatorTypes Typed where
```

(continues on next page)

(continued from previous page)

```
type instance DatumType Typed = ()
type instance RedeemerType Typed = Integer
```

This is quite advanced Haskell, so-called type-level programming, but just like the Template Haskell we have already encountered, you don't really need a deep understanding of it as all scripts will follow the same pattern.

Now we need to compile the validator. Where previously we used `mkValidatorScript`, now we use something called `mkTypedValidator`, which takes our new data type as parameter and produces something of type `TypedValidator`.

```
typedValidator :: Scripts.TypedValidator Typed
typedValidator = Scripts.mkTypedValidator @Typed
  $(PlutusTx.compile [| mkValidator |])
  $(PlutusTx.compile [| wrap |])
where
  wrap = Scripts.wrapValidator @() @Integer
```

This is similar to the `mkValidator` code, but this time we also compile a `wrapValidator` function that takes the datum and redeemer types.

In order for this to work we first need one more import.

```
import qualified Ledger.Typed.Scripts as Scripts
```

In this example, it is being imported qualified and using the `Scripts` prefix, but this is arbitrary and you could pick some other way of referencing the module.

With these changes, the Haskell code will compile, and we now need to change the Template Haskell boilerplate that creates the validator function.

```
validator :: Validator
validator = Scripts.validatorScript typedValidator
```

Here we have used the `validatorScript` function to create an untyped validator from our typed version.

To get the hash we could, of course, use the validator we now have and turn it into a `ValidatorHash` as we did before, but there is a more direct way, which looks identical, but in this case `Scripts` is coming from the module `Ledger.Typed.Scripts` rather than `Ledger.Scripts`. This version takes the typed validator directly.

```
valHash :: Ledger.ValidatorHash
valHash = Scripts.validatorHash typedValidator
```

The script address is calculated as before.

```
scrAddress :: Ledger.Address
scrAddress = scriptAddress validator
```

In this extremely simple example, it probably doesn't seem worth the effort, but for realistic contracts, it is much nicer to do it like this.

The off-chain code is almost identical.

There is a small change to the `give` endpoint. Although we have not yet gone over this part of the code in detail, the following changes can be made.

```
let tx = mustPayToTheScript () $ Ada.lovelaceValueOf amount
ledgerTx <- submitTxConstraints inst tx
```



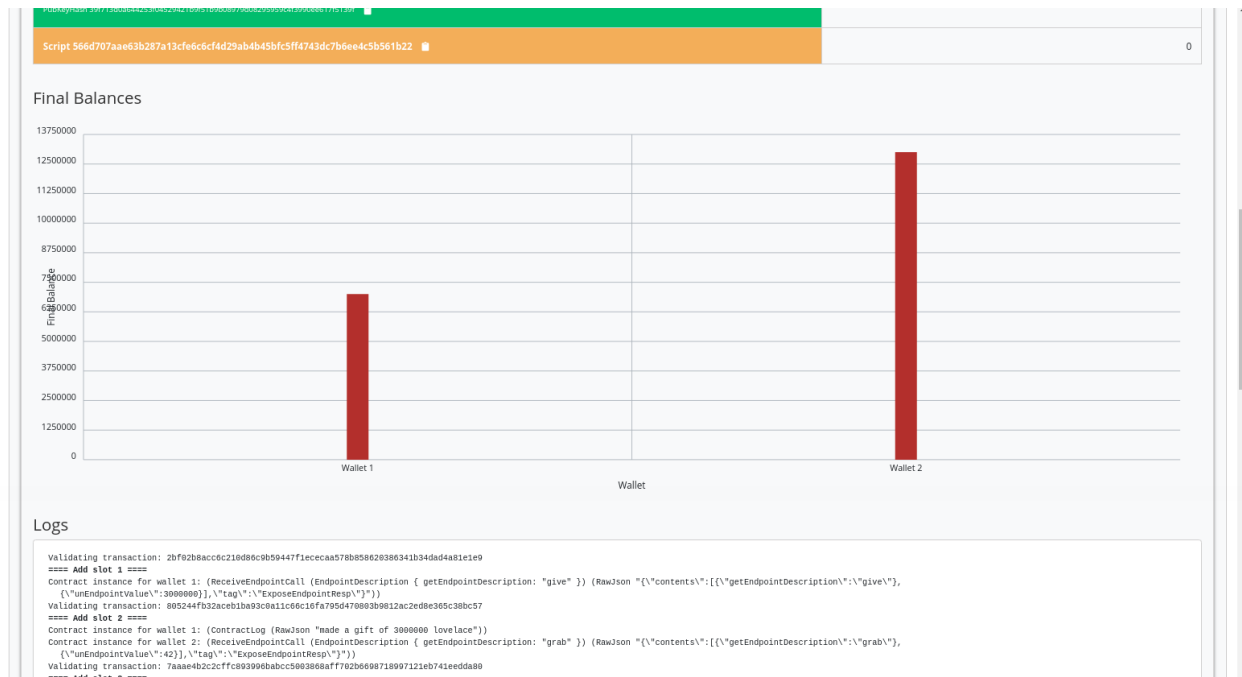
The `mustPayToOtherScript` function has been replaced with `mustPayToTheScript`. This is a convenience script which allows us to pass in just `()` as we no longer need to construct a value of type `Data`. We also no longer need to pass in the script hash.

The behaviour of this code will be identical to the behaviour in the previous example, so we won't go over it in the playground.

Now we will explain how that actually works. How does Plutus convert these custom data types to the actual low-level implementation - the `Data` type.

We can look at the code in the `PlutusTx.IsData.Class` module.

Here we see that there is a quite simple type class defined called `IsData`.



This class provides two functions

- `toData` takes a value and converts it to `Data`
- `fromData` takes a value of type `Data` and attempts to convert it to an instance of type `IsData`. This can fail because not all values of type `Data` will be convertible to the target type.

Let's try this out in the REPL.

```
Prelude Week02.Typed> import PlutusTx
Prelude PlutusTx Week02.Typed> import PlutusTx.IsData.Class
Prelude PlutusTx PlutusTx.IsData.Class Week02.Typed> :i IsData
```

We know that `()` and `Integer` are both instances of `IsData` because they worked in our example.

Let's convert an `Integer` to `Data`

```
Prelude PlutusTx PlutusTx.IsData.Class Week02.Typed> toData (42 :: Integer)
I 42
```

We see that this has been converted to an instance of type `Data` using the `I` constructor, which we did manually before we used typed validation.

Now let's do it the other way around

```
Prelude PlutusTx PlutusTx.IsData.Class Week02.Typed> fromData (I 42) :: Maybe Integer
Just 42
```

We get a Just 42 back - Just being the Maybe constructor when Maybe is not Nothing.

And when it fails, when it can't convert to the target type, we will get back Nothing.

```
Prelude PlutusTx PlutusTx.IsData.Class Week02.Typed> fromData (List []) :: Maybe Integer
Nothing
```

If we examine IsData we can see all the types that this pattern will work for all the types that have an IsData instance defined.

```
Prelude PlutusTx PlutusTx.IsData.Class Week02.Typed> :i IsData
type IsData :: * -> Constraint
class IsData a where
  toData :: a -> Data
  fromData :: Data -> Maybe a
  {-# MINIMAL toData, fromData #-}
  -- Defined in 'PlutusTx.IsData.Class'
instance IsData a => IsData (Maybe a)
  -- Defined in 'plutus-tx-0.1.0.0:PlutusTx.IsData.Instances'
instance (IsData a, IsData b) => IsData (Either a b)
  -- Defined in 'plutus-tx-0.1.0.0:PlutusTx.IsData.Instances'
instance IsData Bool
  -- Defined in 'plutus-tx-0.1.0.0:PlutusTx.IsData.Instances'
instance (IsData a, IsData b, IsData c, IsData d) =>
  IsData (a, b, c, d)
  -- Defined in 'plutus-tx-0.1.0.0:PlutusTx.IsData.Instances'
instance (IsData a, IsData b, IsData c) => IsData (a, b, c)
  -- Defined in 'plutus-tx-0.1.0.0:PlutusTx.IsData.Instances'
instance (IsData a, IsData b) => IsData (a, b)
  -- Defined in 'plutus-tx-0.1.0.0:PlutusTx.IsData.Instances'
instance IsData ()
  -- Defined in 'plutus-tx-0.1.0.0:PlutusTx.IsData.Instances'
instance IsData a => IsData [a]
  -- Defined in 'PlutusTx.IsData.Class'
instance IsData Integer -- Defined in 'PlutusTx.IsData.Class'
instance (TypeError ...) => IsData Int
  -- Defined in 'PlutusTx.IsData.Class'
instance IsData Data -- Defined in 'PlutusTx.IsData.Class'
```

This is still quite a short list of possible types. We would like to use many more types than this for our datum and redeemer.

In order to do this, we would normally need to define an IsData instance for any type that we wish to use. This will allow us to tell the compiler how to do the back and forth conversions. However, this again would be tedious as it is such a mechanical process. So, there is a mechanism in Plutus that does this for us.

## 2.4.5 Example 5 - Custom IsData types

Now let's talk about custom data types. Let's define a silly one and use it in our validator function.

```
newtype MySillyRedeemer = MySillyRedeemer Integer

PlutusTx.unstableMakeIsData 'MySillyRedeemer

{-# INLINABLE mkValidator #-}
mkValidator :: () -> MySillyRedeemer -> ScriptContext -> Bool
mkValidator () (MySillyRedeemer r) _ = traceIfFalse "wrong redeemer" $ r == 42
```

**Note:** There is also a stable version of the `PlutusTx.unstableMakeIsData` function, and the stable version should always be used in production code. The difference between the two is that, in the case where more than one `Data` constructor is required, the unstable version makes no guarantee, between Plutus versions, that the order of constructors will be preserved.

And we need to change some of the boilerplate.

```
data Typed
instance Scripts.ValidatorTypes Typed where
...
    type instance RedeemerType Typed = MySillyRedeemer

typedValidator :: Scripts.TypedValidator Typed
...
where
    wrap = Scripts.wrapValidator @() @MySillyRedeemer
```

We also need to change some off-chain code in the grab endpoint.

Instead of using `I r`, we will use `toData (MySillyRedeemer r)`.

```
grab :: forall w s e. AsContractError e => Integer -> Contract w s e ()
grab r = do
    utxos <- utxoAt scrAddress
    let orefs = fst <$> Map.toList utxos
        lookups = Constraints.unspentOutputs utxos <>
                  Constraints.otherScript validator
    tx :: TxConstraints Void Void
    tx = mconcat [mustSpendScriptOutput oref $ Redeemer $ PlutusTx.toData_
->(MySillyRedeemer r) | oref <- orefs]
    ledgerTx <- submitTxConstraintsWith @Void lookups tx
    void $ awaitTxConfirmed $ txId ledgerTx
    logInfo @String $ "collected gifts"
```

If we try to compile the code now, either on the command line or in the playground, we will get an error because Plutus doesn't know how to convert back and forth between `IsData` and `MySillyRedeemer`.

We could write an instance of `IsData` for `MySillyRedeemer` by hand. But, we don't need to.

Instead we can use another bit of Template Haskell magic.

```
PlutusTx.unstableMakeIsData 'MySillyRedeemer
```

At compile time, the compiler will use the Template Haskell to write an `IsData` instance for us. And now, it will compile.

Let's check it in the REPL.

```
Prelude PlutusTx PlutusTx.IsData.Class> :l src/Week02/IsData.hs
Ok, one module loaded.
Prelude PlutusTx PlutusTx.IsData.Class Week02.IsData> toData (MySillyRedeemer 42)
Constr 0 [I 42]
```

If you try this code, which is in `IsData.hs`, in the playground, you should see that it behaves in the same way as before.

## 2.5 Summary

We have seen a couple of examples of simple validators.

We started with a validator that will always succeed, completely ignoring its arguments. Then we looked at a validator that always fails, again completely ignoring its arguments. Then we looked at one that examines its redeemer to check for a certain predefined value.

We then turned this validator into a typed version which is the one which would be used in practice. First we used built-in data types and then we saw how we can use custom data types.

We have not yet looked at examples where the datum or the context are inspected, which would be required for more realistic examples.

We will look at that in the next lecture.

## WEEK 03 - SCRIPT CONTEXT

---

**Note:** This is a written version of [Lecture #3, Iteration #2](#).

In this lecture we learn about the script context (the third validation argument), handling time, and parameterized contracts.

The code in this lecture uses Plutus commit `81ba78edb1d634a13371397d8c8b19829345ce0d`.

---

### 3.1 Before We Start

Since the last lecture there has been an update to the playground, which is present in the Plutus commit we are using for this lecture (see note above).

There was an issue whereby the timeout, which was hardcoded into the playground was too short. This would cause simulations to fail if they took longer than the hardcoded timeout.

There is now an option when you start the Plutus Playground Server which allows you to specify the timeout. The following example sets the timeout to 120 seconds.

```
plutus-playground-server -i 120s
```

### 3.2 Recap

When we explained the (E)UTxO model in the first lecture, we mentioned that in order to unlock a script address, the script attached to the address is run, and that script gets three pieces of information - the *datum*, the *redeemer* and the *context*.

In the second lecture, we saw examples of that, and we saw how it actually works in Haskell.

We saw the low-level implementation, where all three arguments are represented by the `Data` type. We also saw that in practice this is not used.

Instead, we use the typed version, where the datum and redeemer can be custom types (as long as they implement the `IsData` type class), and where the third argument is of type `ScriptContext`.

In the examples we have seen so far we have looked at the datum and the redeemer, but we have always ignored the context. But the context is, of course, very important. So, in this lecture we will start looking at the context.

### 3.3 ScriptContext

The `ScriptContext` type is defined in package `plutus-ledger-api`, which is a package that, until now, we haven't needed. But now we do need it, and it is included in this week's `.cabal` file. It is defined in module `Plutus.V1.Ledger.Contexts`.

```
data ScriptContext = ScriptContext {
    scriptContextTxInfo :: TxInfo,
    scriptContextPurpose :: ScriptPurpose
}
```

It is a record type with two fields.

The second field is of type `ScriptPurpose`, which is defined in the same module. It defines for which purpose a script is being run.

```
data ScriptPurpose
= Minting CurrencySymbol
| Spending TxOutRef
| Rewarding StakingCredential
| Certifying DCert
```

For us, the most important is `Spending`. This is what we have talked about so far in the context of the (E)UTxO model. This is when a script is run in order to validate a spending input for a transaction.

The `Minting` purpose comes into play when you want to define a native token. Its purpose is to describe under which circumstances the native token can be minted or burned.

There are also two new brand new purposes - `Rewarding` - related to staking and `Certifying` - related to stake delegation.

The most interesting field, the one that contains the actual context, is `scriptContextTxInfo`, which is of type `TxInfo`, also defined in the same module.

```
data TxInfo = TxInfo
{ txInfoInputs      :: [TxInInfo] -- ^ Transaction inputs
, txInfoOutputs     :: [TxOut]   -- ^ Transaction outputs
, txInfoFee         :: Value     -- ^ The fee paid by this transaction.
, txInfoForge       :: Value     -- ^ The 'Value' forged by this transaction.
, txInfoDCert       :: [DCert]   -- ^ Digests of certificates included in this
↳ transaction
, txInfoWdrl        :: [(StakingCredential, Integer)] -- ^ Withdrawals
, txInfoValidRange  :: SlotRange -- ^ The valid range for the transaction.
, txInfoSignatories :: [PubKeyHash] -- ^ Signatures provided with the transaction,
↳ attested that they all signed the tx
, txInfoData        :: [(DatumHash, Datum)]
, txInfoId          :: TxId
  -- ^ Hash of the pending transaction (excluding witnesses)
} deriving (Generic)
```

It describes the spending transaction. In the (E)UTxO model, the context of validation is the spending transaction and its inputs and outputs. This context is expressed in the `TxInfo` type.

There are a couple of fields that are global to the whole transaction and in particular we have the list of all the inputs `txInfoInputs` and the list of all the outputs `txInfoOutputs`. Each of those has a variety of fields to drill into each individual input or output.

We also see fields for fees `txFee`, the forge value `txInfoForge`, used when minting or burning native tokens.

Then we have a list of delegation certificates in `txInfoDCert` and a field `txInfoWdr1` to hold information about staking withdrawals.

The field `txInfoValidRange`, which we will look at in much more detail in a moment, defines the slot range for which this transaction is valid.

`txInfoSignatories` is the list of public keys that have signed this transaction.

Transactions that spend a script output need to include the datum of the script output. The `txInfoData` field is a list associating datums with their respective hashes. If there is a transaction output to a script address that carries some datum, you don't need to include the datum, you can just include the datum hash. However, scripts that spend an output do need to include the datum, in which case it will be included in the `txInfoData` list.

Finally, the `txInfoId` field is the ID of this transaction.

### 3.3.1 `txInfoValidRange`

While there is a lot of information contained in this `txInfo` type, for our first example of how to use the third argument to validation, we will concentrate on the `txInfoValidRange` field.

This brings us to an interesting dilemma. We have stressed several times that the big advantage that Cardano has over something like Ethereum is that validation can happen in the wallet. But we have also noted that a transaction can still fail on-chain following validation if, when the transaction arrives on the blockchain, it has been consumed already by someone else. In this case, the transaction fails without having to pay fees.

What should never happen under normal circumstances is that a validation script runs and then fails. This is because you can always run the validation under exactly the same conditions in the wallet, so it would fail before you ever submit it.

So that is a very nice feature, but it is not obvious how to manage time in that context. Time is important, because we want to be able to express that a certain transaction is only valid before or only valid after a certain time has been reached.

We saw an example of this in lecture one - the auction example, where bids are only allowed until the deadline has been reached, and the `close` endpoint can only be called after the deadline has passed.

That seems to be a contradiction, because time is obviously flowing. So, when you try to validate a transaction that you are constructing in your wallet, the time that you are doing that can, of course, be different than the time that the transaction arrives at a node for validation. So, it's not clear how to bring these two together so that validation is deterministic, and to guarantee that if, and only if, validation succeeds in the wallet, it will also succeed in the node.

The way Cardano solves that, is by adding the slot range field `txInfoValidRange` to a transaction, which essentially says "This transaction is valid between *this* and *that* slot".

When a transaction gets submitted to the blockchain and validated by a node, then before any scripts are run, some general checks are made, for example that all inputs are present and that the balances add up, that the fees are included and so on.

One of those checks that happens before validation is to check that the slot range is valid. The node will look at the current time and check that it falls into the valid slot range of the transaction. If it does not, then validation fails immediately without ever running the validator scripts.

So, if the pre-checks succeed, then this means that the current time does fall into the valid slot range. This, in turn, means that we are completely deterministic again. The validation script can simply assume that it is being run at a valid slot.

By default, a script will use the infinite slot range, one that covers all slots starting from the genesis block and running until the end of time.

There is one slight complication with this, and that is that Ouroboros, the consensus protocol powering Cardano doesn't use POSIX time, it uses slots. But Plutus uses real time, so we need to be able to convert back and forth between real time and slots. This is no problem so long as the slot time is fixed. Right now it is one second, so right now it is easy.

However, this could change in the future. There could be a hard fork with some parameter change that would change the slot time. We can't know that in advance. We don't know what the slot length will be in ten years, for example.

That means that slot intervals that are defined for transactions mustn't have a definite upper bound that is too far in the future. It must only be as far in the future as it is possible to know what the slot length will be. This happens to be something like 36 hours. We know that if there is going to be a hard fork, we would know about it at least 36 hours in advance.

### 3.3.2 POSIXTimeRange

Let's look at this `POSIXTimeRange` type, which is defined in `Plutus.V1.Ledger.Time`.

```
type POSIXTimeRange = Interval POSIXTime.
```

It is a type synonym for `Interval POSIXTime` and we see that `Interval` is defined by a `LowerBound` and an `UpperBound`.

```
Interval
  ivFrom :: LowerBound a
  ivTo   :: UpperBound a
```

If we drill into `LowerBound` we see the constructor

```
data LowerBound a = LowerBound (Extended a) Closure
```

`Closure` is a synonym for `Bool` and specifies whether a bound is included in the `Interval` or not.

`Extended` can be `NegInf` for negative infinity, `PosInf` for positive infinity, or `Finite a`.

We also find some helper functions including the member function which checks if a given `a` is part of a given `Interval`, so long as the type of `a` is a subtype of `Ord`, which is the case for `POSIXTime`.

```
member :: Ord a => a -> Interval a -> Bool
member a i = i `contains` singleton a
```

`interval` is a smart constructor for the `Interval` type which creates an `Interval` with an inclusive upper and lower bound.

```
interval :: a -> a -> Interval a
interval s s' = Interval (lowerBound s) (upperBound s')
```

Then we have `from` which constructs an `Interval` which starts at `a` and lasts until eternity.

```
from :: a -> Interval a
from s = Interval (lowerBound s) (UpperBound PosInf True)
```

And we have `to`, which is the opposite. It constructs an `Interval` starting from the genesis block up to, and including `a`.

```
to :: a -> Interval a
to s = Interval (LowerBound NegInf True) (upperBound s)
```

`always` is the default `Interval` which includes all times.



```
always :: Interval a
always = Interval (LowerBound NegInf True) (UpperBound PosInf True)
```

And we have the opposite, never, which contains no slots.

```
never :: Interval a
never = Interval (LowerBound PosInf True) (UpperBound NegInf True)
```

There is also the singleton helper, which constructs an interval which consists of just one slot.

```
singleton :: a -> Interval a
singleton s = interval s s
```

The function hull gives the smallest interval containing both the given intervals.

```
hull :: Ord a => Interval a -> Interval a -> Interval a
hull (Interval l1 h1) (Interval l2 h2) = Interval (min l1 l2) (max h1 h2)
```

The intersection function determines the largest interval that is contained in both the given intervals. This is an Interval that starts from the largest lower bound of the two intervals and extends until the smallest upper bound.

```
intersection :: Ord a => Interval a -> Interval a -> Interval a
intersection (Interval l1 h1) (Interval l2 h2) = Interval (max l1 l2) (min h1 h2)
```

The overlaps function checks whether two intervals overlap, that is, whether there is a value that is a member of both intervals.

```
overlaps :: Ord a => Interval a -> Interval a -> Bool
overlaps l r = isEmpty (l `intersection` r)
```

contains takes two intervals and determines if the second interval is completely contained within the first one.

```
contains :: Ord a => Interval a -> Interval a -> Bool
contains (Interval l1 h1) (Interval l2 h2) = l1 <= l2 && h2 <= h1
```

And we have the before and after functions to determine, if a given time is, respectively, before or after everything in a given Interval.

```
before :: Ord a => a -> Interval a -> Bool
before h (Interval f _) = lowerBound h < f

after :: Ord a => a -> Interval a -> Bool
after h (Interval _ t) = upperBound h > t
```

Let's have a play in the REPL.

```
Prelude Week03.Homework1> import Plutus.V1.Ledger.Interval
Prelude Plutus.V1.Ledger.Interval Week03.Homework1>
```

Let's construct the Interval between 10 and 20, inclusive.

```
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> interval (10 :: Integer) 20
Interval {ivFrom = LowerBound (Finite 10) True, ivTo = UpperBound (Finite 20) True}
```

We can check whether a value is a member of an interval:

```
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 9 $ interval (10 :: Integer)␣  
↪20  
False  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 10 $ interval (10 :: Integer)␣  
↪20  
True  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 12 $ interval (10 :: Integer)␣  
↪20  
True  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 20 $ interval (10 :: Integer)␣  
↪20  
True  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 21 $ interval (10 :: Integer)␣  
↪20  
False
```

We can use the `from` constructor. Here the lower bound is again a finite slot, but the upper bound is positive infinity.

```
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 21 $ from (30 :: Integer)  
False  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 30 $ from (30 :: Integer)  
True  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 300000 $ from (30 :: Integer)  
True
```

And the `to` constructor. Here the lower bound is negative infinity, while the upper bound is a finite slot number.

```
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 300000 $ to (30 :: Integer)  
False  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 31 $ to (30 :: Integer)  
False  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 30 $ to (30 :: Integer)  
True  
  
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> member 7 $ to (30 :: Integer)  
True
```

Now, let's try the intersection function on the Interval from 10 to 20 and the Interval from 18 to 30.

```
Prelude Plutus.V1.Ledger.Interval Week03.Homework1> intersection (interval (10 :: Integer) 20) $ interval 18 30  
Interval {ivFrom = LowerBound (Finite 18) True, ivTo = UpperBound (Finite 20) True}
```

As expected, we get the Interval that runs from 18 to 20, inclusive.

We can check whether one Interval contains another.

```

Prelude Plutus.V1.Ledger.Interval Week03.Homework1> contains (to (100 :: Integer)) $␣
↳ interval 30 80
True

Prelude Plutus.V1.Ledger.Interval Week03.Homework1> contains (to (100 :: Integer)) $␣
↳ interval 30 100
True

Prelude Plutus.V1.Ledger.Interval Week03.Homework1> contains (to (100 :: Integer)) $␣
↳ interval 30 101
False

```

We see that as soon as the second Interval extends to 101, it is no longer fully contained within the Interval that runs to 100.

However, if we check with overlaps, then it will be true because there are elements, such as 40, that are contained in both intervals.

```

Prelude Plutus.V1.Ledger.Interval Week03.Homework1> overlaps (to (100 :: Integer)) $␣
↳ interval 30 101
True

Prelude Plutus.V1.Ledger.Interval Week03.Homework1> overlaps (to (100 :: Integer)) $␣
↳ interval 101 110
False

```

## 3.4 Example - Vesting

Imagine you want to give a gift of Ada to a child. You want the child to own the Ada, but you only want the child to have access to it he or she turns eighteen.

Using Plutus, it is very easy to implement. As our first contract that will look at the context argument, we will implement a contract that implements a vesting scheme. Money will be put into a script and then it can be retrieved by a certain person, but only once a certain deadline has been reached.

We start by copying the IsData contract from lecture two into a new module called Vesting.

The first step is to think about the types for the datum and redeemer.

For datum, it makes sense to have two pieces of information, the beneficiary and the deadline. So, let's define this type:

```

data VestingDatum = VestingDatum
  { beneficiary :: PubKeyHash
  , deadline    :: POSIXTime
  } deriving Show

PlutusTx.unstableMakeIsData ''VestingDatum

```

In order to know if someone can spend this script output, two pieces information are required, i.e. the beneficiary's signature and the time of the transaction. In this case, both those pieces of information are contained in the transaction itself. This means that we don't need any information in the redeemer, so we can just use () for the redeemer.

```

mkValidator :: VestingDatum -> () -> ScriptContext -> Bool

```

We need to check two conditions.

1. That only the correct beneficiary can unlock a UTxO sitting at this address. This we can validate by checking that the beneficiary's signature is included in the transaction.
2. That this transaction is only executed after the deadline is reached.

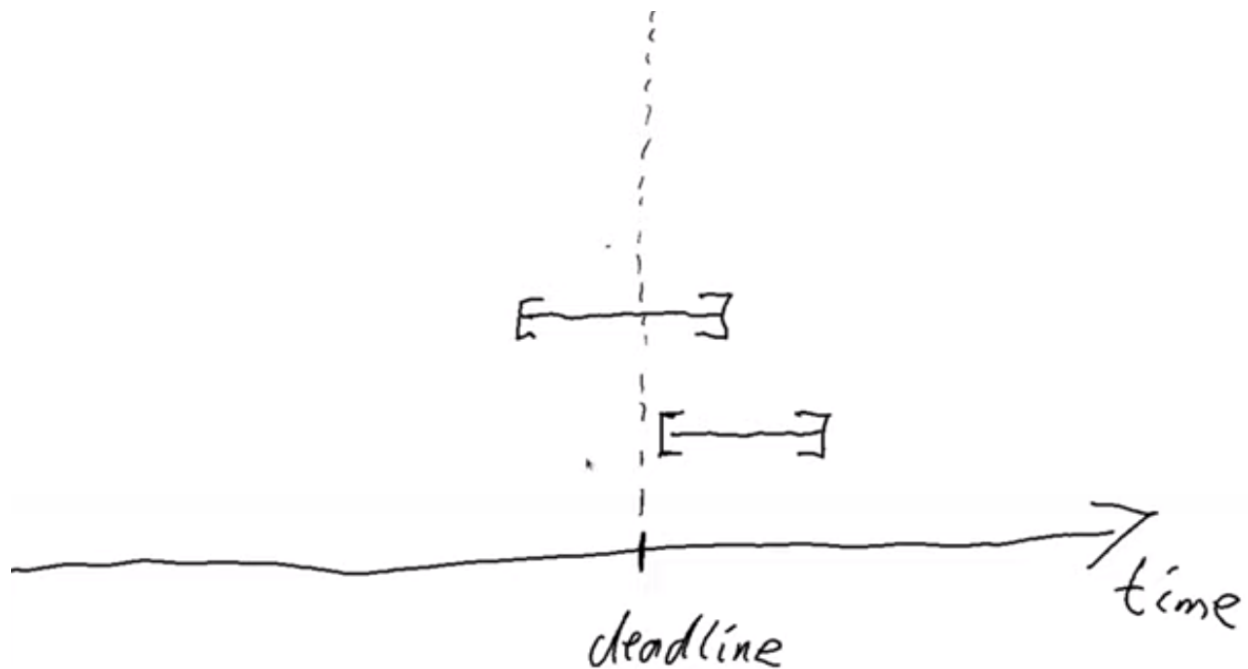
We could probably just write this in one go, but we will write it in a more top-down fashion and delegate to some helper functions.

```
mkValidator dat () ctx =  
    mkValidator dat () ctx = traceIfFalse "beneficiary's signature missing"   
    ↳ signedByBeneficiary &&  
                                traceIfFalse "deadline not reached" deadlineReached  
where  
    info :: TxInfo  
    info = scriptContextTxInfo ctx
```

To check that the transaction is signed by the beneficiary, we can get the public key of the beneficiary from the datum and pass it, along with the transaction information to the `txSignedBy` function.

```
signedByBeneficiary :: Bool  
signedByBeneficiary = txSignedBy info $ beneficiary dat
```

How do we check that the deadline has passed?



Let's consider a transaction with a validity that crosses the deadline, which is shown as the uppermost range in the above diagram.

Recall that before the validator script is run, other checks are made, including the time check. The node checks that the current time falls into the valid range of the transaction and only then is the validator run. So we know that, if we are in the validator, the current time lies somewhere within the validity interval.

In the case of the range that crosses the deadline, the validator code cannot know whether the current time is before or after the deadline. In this case, the validator must declare that the transaction is invalid.

The second example in the diagram, however, is fine. We still don't know what the current time is exactly, but we know that whatever the time is, it will be after the deadline.

So, what we are checking for is that the whole validity interval is to the right of the deadline. One way to do this is to use the `contains` function to check whether the validity interval is fully contained within the interval that starts from the deadline and extends until the end of time.

```
deadlineReached :: Bool
deadlineReached = contains (from $ deadline dat) $ txInfoValidRange info
```

That completes the validation logic. Let's take care of some boilerplate.

```
data Vesting
instance Scripts.ValidatorTypes Vesting where
  type instance DatumType Vesting = VestingDatum
  type instance RedeemerType Vesting = ()

typedValidator :: Scripts.TypedValidator Vesting
typedValidator = Scripts.mkTypedValidator @Vesting
  $(PlutusTx.compile [| mkValidator |])
  $(PlutusTx.compile [| wrap |])
where
  wrap = Scripts.wrapValidator @VestingDatum @()
```

We will focus more on the wallet part of the script later, but here are the changes.

In addition to some new LANGUAGE pragmas and some extra imports, we have created a `GiveParams` type, and modified the `grab` endpoint to require no parameters.

The `VestingSchema` type defines the endpoints that we want to expose to the user. As in our last example, `give` will be used by the user who puts funds into the contract, then `grab` will be used by the user wanting to claim the funds.

```
type VestingSchema =
  .\ Endpoint "give" GiveParams
  .\ Endpoint "grab" ()
```

So what parameters do we need for `give`? The endpoint will create a UTxO at the vesting script address with an amount and a datum. If you recall, our datum contains the beneficiary and the deadline. So, there are three pieces of information that we must pass to the `give` endpoint.

```
data GiveParams = GiveParams
{ gpBeneficiary :: !PubKeyHash
, gpDeadline    :: !POSIXTime
, gpAmount      :: !Integer
} deriving (Generic, ToJSON, FromJSON, ToSchema)
```

The `grab` endpoint doesn't require any parameters because the beneficiary will just look for UTxOs sitting at the script address and can then check whether they are the beneficiary and whether the deadline has passed. If so, they can consume them.

Let's quickly look at the `give` endpoint.

```
give :: AsContractError e => GiveParams -> Contract w s e ()
give gp = do
  let dat = VestingDatum
      { beneficiary = gpBeneficiary gp
```

(continues on next page)

(continued from previous page)

```

        , deadline    = gpDeadline gp
      }
      tx = mustPayToTheScript dat $ Ada.lovelaceValueOf $ gpAmount gp
      ledgerTx <- submitTxConstraints typedValidator tx
      void $ awaitTxConfirmed $ txId ledgerTx
      logInfo @String $ printf "made a gift of %d lovelace to %s with deadline %s"
        (gpAmount gp)
        (show $ gpBeneficiary gp)
        (show $ gpDeadline gp)

```

First we compute the datum we want to use, and we can get both pieces of information from the GiveParams which is passed into the function.

Then, for the transaction, we add a constraint that there must be an output at this script address with the datum that we just defined and a certain number of lovelace, which we also get from the GiveParams.

The rest of the function is as before, just with a different log message.

The grab endpoint is a bit more involved.

There can be many UTxOs at this script address and some of them might not be suitable for us, either because we are not the beneficiary, or because the deadline has not yet passed. If we try to submit a transaction when there are no suitable UTxOs, we will pay fees, but get nothing in return.

```

grab :: forall w s e. AsContractError e => Contract w s e ()
grab = do
  now    <- currentTime
  pkh    <- pubKeyHash <$> ownPubKey
  utxos  <- Map.filter (isSuitable pkh now) <$> utxoAt scrAddress
  if Map.null utxos
  then logInfo @String $ "no gifts available"
  else do
    let orefs = fst <$> Map.toList utxos
        lookups = Constraints.unspentOutputs utxos <>
                  Constraints.otherScript validator
        tx :: TxConstraints Void Void
        tx = mconcat [mustSpendScriptOutput oref $ Redeemer $ PlutusTx.
  ↳toData () | oref <- orefs] <>
        mustValidateIn (from now)
    ledgerTx <- submitTxConstraintsWith @Void lookups tx
    void $ awaitTxConfirmed $ txId ledgerTx
    logInfo @String $ "collected gifts"
  where
    isSuitable :: PubKeyHash -> POSIXTime -> TxOutTx -> Bool
    isSuitable pkh now o = case txOutDatumHash $ txOutTxOut o of
      Nothing -> False
      Just h -> case Map.lookup h $ txData $ txOutTxTx o of
        Nothing -> False
        Just (Datum e) -> case PlutusTx.fromData e of
          Nothing -> False
          Just d -> beneficiary d == pkh && deadline d <= now

```

First, we get the current time and calculate our public key hash. We then look up all the UTxOs at this address and filter them using the isSuitable helper function, which is defined in the where clause.

It first checks the datum hash, and, if it finds it, it attempts to look up the corresponding datum. Recall that the producing transaction, in this case `give` doesn't have to supply the datum, it need only supply the datum hash. However, in our case we need to have the datum available to the `grab` endpoint, so the `give` endpoint does provide the datum.

If the `grab` endpoint finds the datum, it must deserialise it to the `Vesting` type.

If all of this succeeds we can check whether we are the beneficiary and whether the deadline has passed.

At this point, `utxos` contains all the UTxOs that we can consume. If we find none, then we just log a message to that effect. If there is at least one, then we construct one transaction that consumes all of them as inputs and pays the funds to our wallet.

As `lookups`, we provide the list of UTxOs as well as the validator script. Recall that, in order to consume UTxOs at this address, the spending transaction must provide the validation script.

We then create a transaction that spends all the suitable UTxOs along with a constraint that it must validate in the `Interval` which stretches from now until the end of time. If we don't provide the interval here, then validation will fail, because the default interval is from genesis until the end of time. The on-chain validation would reject this as it needs an interval that is fully contained in the interval stretching from the deadline until the end of time.

We could use the singleton `Interval` now, but, if there were any issues, for example network delays, and the transaction arrived at a node a slot or two later, then validation would no longer work.

The, we just bundle up the endpoints.

```
endpoints :: Contract () VestingSchema Text ()
endpoints = (give' `select` grab') >> endpoints
  where
    give' = endpoint @"give" >=> give
    grab' = endpoint @"grab" >> grab
```

Then there is some boilerplate which is just used in the playground.

```
mkSchemaDefinitions "'VestingSchema
mkKnownCurrencies []
```

### 3.4.1 In the playground

First, let's add a third wallet. We are going to create a scenario where Wallet 1 makes two gifts to Wallet 2 with different deadlines and also makes one gift to Wallet 3.

Normally it would be possible to submit both `give` transactions in the same slot, but the way our code is implemented, we wait for confirmation, which means we need to add a wait action. This is maybe not the best way to do it, but that's how it is for the time being.


Here we run into our first problem. We need to supply the beneficiary address, but there is no way in the playground to get the public key hash of a wallet.

But we can get it from the REPL.

```
Prelude Week03.Homework1> :l src/Week03/Vesting.hs
Ok, one module loaded.
Prelude Week03.Vesting> import Ledger
Prelude Ledger Week03.Vesting> import Wallet.Emulator
Prelude Ledger Wallet.Emulator Week03.Vesting> pubKeyHash $ walletPubKey $ Wallet 2
39f713d0a644253f04529421b9f51b9b08979d08295959c4f3990ee617f5139f
```

(continues on next page)

## Plutus Pioneer Program Lecture Notes

 PLUTUS PLAYGROUND

Getting StartedTutorialsAPIPrivacy

Demo filesHello, world!StarterGameVestingCrowd FundingError Handling

Log In

Simulator

Simulation 1 +

### Wallets

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

Wallet 1

Opening Balances

Lovelace100000000

Available functions

give +grab +

Pay to Wallet +

Wallet 2

Opening Balances

Lovelace100000000

Available functions

give +grab +

Pay to Wallet +

Wallet 3

Opening Balances

Lovelace100000000

Available functions

give +grab +

Pay to Wallet +

+  
Add Wallet

Evaluate

Transactions


### Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

+  
Add Wait Action

Evaluate

Transactions

 PLUTUS PLAYGROUND

Getting StartedTutorialsAPIPrivacy

Demo filesHello, world!StarterGameVestingCrowd FundingError Handling

Log In

Simulator

Simulation 1 +

### Wallets

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

Wallet 1

Opening Balances

Lovelace100000000

Available functions

give +grab +

Pay to Wallet +

Wallet 2

Opening Balances

Lovelace100000000

Available functions

give +grab +

Pay to Wallet +

Wallet 3

Opening Balances

Lovelace100000000

Available functions

give +grab +

Pay to Wallet +

+  
Add Wallet

Fix Errors

Transactions

### Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: give

gpBeneficiary

getPubKeyHash

String

gpDeadline

Integer

Required

gpAmount

Integer

Required

2

Wait

Wait For... Wait Until...

Blocks

10

3

Wallet 1: give

gpBeneficiary

getPubKeyHash

String

gpDeadline

Integer

Required

gpAmount

Integer

Required

4

Wait

Wait For... Wait Until...

Blocks

1

5

Wallet 1: give

gpBeneficiary

getPubKeyHash

String

gpDeadline

Integer

Required

gpAmount

Integer

Required

+  
Add Action



(continued from previous page)

```
Prelude Ledger Wallet.Emulator Week03.Vesting> pubKeyHash $ walletPubKey $ Wallet 3
dac073e0123bdea59dd9b3bda9cf6037f63aca82627d7abcd5c4ac29dd74003e
```

The next problem is the deadline. In the last lecture we saw how to convert between slots and POSIX times. This has changed. Previously you just needed a slot and out came a POSIX time. Now there is a second argument.

```
Prelude Ledger Wallet.Emulator Week03.Vesting> import Ledger.TimeSlot
Prelude Ledger Wallet.Emulator Ledger.TimeSlot Week03.Vesting> :t slotToBeginPOSIXTime
slotToBeginPOSIXTime :: SlotConfig -> Slot -> POSIXTime
```

There are also versions of `slotToBeginPOSIXTime` that have a begin and an end time. This is because a slot is not just a point in time, it's a duration in time.

So what is this `SlotConfig`?

```
Prelude Ledger Wallet.Emulator Ledger.TimeSlot Week03.Vesting> :i SlotConfig
type SlotConfig :: *
data SlotConfig
  = SlotConfig {scSlotLength :: Integer, scZeroSlotTime :: POSIXTime}
  -- Defined in 'Ledger.TimeSlot'
instance Eq SlotConfig -- Defined in 'Ledger.TimeSlot'
instance Show SlotConfig -- Defined in 'Ledger.TimeSlot'
```

It takes the slot length and the time at which slot zero starts.

So now we have to find out what `SlotConfig` to use for the playground. Luckily, it's the default. For that we need to use the `Data.Default` module.

```
Prelude Ledger Wallet.Emulator Ledger.TimeSlot Week03.Vesting> import Data.Default
Prelude Ledger Wallet.Emulator Ledger.TimeSlot Data.Default Week03.Vesting> def :: SlotConfig
SlotConfig {scSlotLength = 1000, scZeroSlotTime = POSIXTime {getPOSIXTime = 1596059091000}}
```

(continues on next page)

(continued from previous page)

Now we can use `slotToBeginPOSIXTime` with the default config to get the POSIX time for slot 10 and slot 20.

```
Prelude Ledger Wallet.Emulator Ledger.TimeSlot Data.Default Week03.Vesting>
↳ slotToBeginPOSIXTime def 10
POSIXTime {getPOSIXTime = 1596059101000}

Prelude Ledger Wallet.Emulator Ledger.TimeSlot Data.Default Week03.Vesting>
↳ slotToBeginPOSIXTime def 20
POSIXTime {getPOSIXTime = 1596059111000}
```

And we can use these in the playground. We'll use slot 10 as the deadline for the first and third give`s and slot 20 for the second `give. We'll also give 10 Ada in each case.

The screenshot shows the Plutus Playground Simulator interface. At the top, there's a navigation bar with links like 'Getting Started', 'Tutorials', 'API', and 'Privacy'. Below that, a 'Demo files' section lists 'Hello, world', 'Starter', 'Game', 'Vesting', 'Crowd Funding', and 'Error Handling'. The main area is titled 'Simulator' and contains a 'Simulation 1' tab. Under 'Wallets', there are three wallet configurations (Wallet 1, 2, and 3) and an 'Add Wallet' button. Each wallet has an 'Opening Balances' section with a 'Lovelace' input set to 100000000 and an 'Available functions' section with 'give', 'grab', and 'Pay to Wallet' buttons. Below the wallets is an 'Actions' section with a sequence of five actions: 1. 'Wallet 1: give' with 'gpBeneficiary' set to '39f713d0a64425304529421b9f51b5', 'gpDeadline' set to '1596059101000', and 'gpAmount' set to '10000000'. 2. 'Wait' with 'Wait For...' selected and 'Blocks' set to '1'. 3. 'Wallet 1: give' with 'gpBeneficiary' set to '39f713d0a64425304529421b9f51b5', 'gpDeadline' set to '1596059111000', and 'gpAmount' set to '10000000'. 4. 'Wait' with 'Wait For...' selected and 'Blocks' set to '1'. 5. 'Wallet 1: give' with 'gpBeneficiary' set to 'dac073e0123bdea59d9b3bda9cf60', 'gpDeadline' set to '1596059101000', and 'gpAmount' set to '10000000'. Each action has a green checkmark indicating it's valid.

Let's create a scenario where everything works. Wallet 3 grabs at slot 10 when the deadline for Wallet 3 has passed, and Wallet 2 grabs at slot 20, when both the Wallet 2 deadlines have passed. We will use the `Wait Until...` option for this.

After evaluation, we first see the Genesis transaction.

If we look at the next transaction, we see the gift from Wallet 1 to Wallet 2 with the deadline of 10. Here, ten Ada get locked in the script address.

The next transaction is the gift from Wallet 1 to Wallet 2 with the deadline of 20. A new UTxO is now created at the script address with ten Ada.

And the third gift, this time to Wallet 3, with a deadline of 10. Wallet 1 now has about 70 Ada, and another UTxO is created with 10 Ada locked at the script address.

At slot 10, Wallet 3 grabs successfully. The third UTxO is the input, some fees are paid, and then the remainder of the lovelace is sent to Wallet 3.

Then at slot 20, Wallet 2 successfully grabs both the UTxOs for which they are the beneficiary. This time the fee is higher because two validators have to run.

The final balances reflect the changes.

### 3.4. Example - Vesting

Simulation 1 +

## Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0 Slot 3, Tx 0 Slot 10, Tx 0 Slot 20, Tx 0

### Inputs

Wallet 1  
PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed4...

Ada  
Lovelace 100,000,000  
Created by: Slot 0, Tx 0

### Transaction

Slot 1, Tx 0

Tx: 753bb67e1cadda1cfbd059577cf9f1d5a17a099db60dd9c3ad94485c158f35

Validity: All time

Signatures:

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

### Outputs

Fee  
Ada  
Lovelace 10

Wallet 1  
PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed4...

Ada  
Lovelace 89,999,990  
Spent in: Slot 2, Tx 0

Script b9762066c2931c2416876f5f945e3e8fc73...

Ada  
Lovelace 10,000,000  
Spent in: Slot 20, Tx 0

### Balances Carried Forward (as at Slot 1, Tx 0)

Beneficial Owner	Ada	Lovelace
Wallet 1 PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed46a5e58877e47072109		89,999,990
Wallet 2		

Simulation 1 +

## Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0 Slot 3, Tx 0 Slot 10, Tx 0 Slot 20, Tx 0

### Inputs

Wallet 1  
PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed4...

Ada  
Lovelace 89,999,990  
Created by: Slot 1, Tx 0

### Transaction

Slot 2, Tx 0

Tx: c9cf53a79ba5e70c33f7da624d7d446b5c9902799a79db3d2ecd401594e2a

Validity: All time

Signatures:

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

### Outputs

Fee  
Ada  
Lovelace 10

Wallet 1  
PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed4...

Ada  
Lovelace 79,999,980  
Spent in: Slot 3, Tx 0

Script b9762066c2931c2416876f5f945e3e8fc73...

Ada  
Lovelace 10,000,000  
Unspent

### Balances Carried Forward (as at Slot 2, Tx 0)

Beneficial Owner	Ada	Lovelace
Wallet 1 PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed46a5e58877e47072109		79,999,980
Wallet 2		

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0   Slot 1, Tx 0   Slot 2, Tx 0   **Slot 3, Tx 0**   Slot 10, Tx 0   Slot 20, Tx 0

**Inputs**

Wallet 1  
PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed4...

Ada  
Lovelace 79,999,980

Created by: Slot 2, Tx 0

**Transaction**

Slot 3, Tx 0

**Tx:** 86d0882b45b26613bc745d1915862a09c8cae04704764b53bd715956f9cd2e0

**Validity:** All time

**Signatures:**

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68707511a

**Outputs**

**Fee**

Ada  
Lovelace 10

Wallet 1  
PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed4...

Ada  
Lovelace 69,999,970

Unspent

Script b9762066c2931c241687ef5f945e3e8fc73...

Ada  
Lovelace 10,000,000

Spent in: Slot 10, Tx 0

**Balances Carried Forward (as at Slot 3, Tx 0)**

Beneficial Owner	Ada	Lovelace
Wallet 1 PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed46abe6aa58877ef476721b9		69,999,970
Wallet 2		

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0   Slot 1, Tx 0   Slot 2, Tx 0   Slot 3, Tx 0   **Slot 10, Tx 0**   Slot 20, Tx 0

**Inputs**

Script b9762066c2931c241687ef5f945e3e8fc73...

Ada  
Lovelace 10,000,000

Created by: Slot 3, Tx 0

**Transaction**

Slot 10, Tx 0

**Tx:** 2a0dee5876db4cfa90635c2a7adeb273d768a19fbdacebecd9968f60e21a0a

**Validity:** From Slot 10 (inclusive) to the end of time (inclusive)

**Signatures:**

- PubKey fcf51cd8e6218a138da47ed00230f0580816ed13ba3303ac5deb911548908025

**Outputs**

**Fee**

Ada  
Lovelace 8,707

Wallet 3  
PubKeyHash dac073e0123bdea59d49b3bda9c9637963acab...

Ada  
Lovelace 9,991,293

Unspent

**Balances Carried Forward (as at Slot 10, Tx 0)**

Beneficial Owner	Ada	Lovelace
Wallet 1 PubKeyHash 21fe31dfa154a261626c854046fd2271b7bed46abe6aa58877ef476721b9		69,999,970
Wallet 2 PubKeyHash 39f713d0a6442e304529421109f110a08979d0829593a4f3990ee41785139		100,000,000
Wallet 3 PubKeyHash dac073e0123bdea59d49b3bda9c9637963acab...		109,991,293
Script b9762066c2931c241687ef5f945e3e8fc73279083528bf0a2ec57d4266076be28		20,000,000



Now let's look at the case where the grab happens too early. We'll make Wallet 2 grab at slot 15 instead of slot 20.

The screenshot shows the Plutus Pioneer Program interface with three wallets and a sequence of actions:

- Wallet 1: start** (Action 1):
  - spDeadline: 1596059101
  - spMinBid: 100000000
  - spCurrency: 66
  - unCurrencySymbol: 66
  - spToken: T
  - unTokenName: T
- Wait** (Action 2):
  - Wait For... (selected)
  - Blocks: 1
- Wallet 2: bid** (Action 3):
  - bpCurrency: 66
  - unCurrencySymbol: 66
  - bpToken: T
  - unTokenName: T
  - bpBid: 100000000
- Wait** (Action 4):
  - Wait For... (selected)
  - Blocks: 1
- Wallet 3: bid** (Action 5):
  - bpCurrency: 66
  - unCurrencySymbol: 66
  - bpToken: T
  - unTokenName: T
  - bpBid: 200000000
- Wait** (Action 6):
  - Wait Until... (selected)
  - Slot: 11

Buttons at the bottom: Evaluate, Transactions.

Now we see that the first transactions are the same, but that the final transaction at slot 15 has only one input, because the second UTxO is not yet available.

And we can see that there are 10 Ada still locked at the script address.

Our off-chain code was written in such a way that it will only submit a transaction if there is a suitable UTxO that can be grabbed. This means that we don't really exercise the validator because we are only sending transactions to the blockchain that will pass validation.

If you want to test the validator, you could modify the wallet code so that the grab endpoint attempts to grab everything and then validation will fail if you are not the beneficiary or the deadline has not been reached.

You need to keep in mind that anybody can write off-chain code. So, even though it works now as long as you use the grab endpoint that we wrote ourselves, somebody could write a different piece of off-chain code that doesn't filter the UTxOs as we did. In this case, if the validator is not correct something could be horribly wrong.

Simulation 1

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 3, Tx 0

Slot 10, Tx 0

Slot 15, Tx 0

Inputs

Script b9762066c2931c241687ef5f945e3e8fc73...

Ada

Lovelace

10,000,000

Created by: Slot 1, Tx 0

Transaction

Slot 15, Tx 0

Tx: b6dc119b661c203be6238577c72a7c47777745fa44dc07b8a6d5b5948ed541

Validity: From Slot 15 (inclusive) to the end of time (inclusive)

Signatures:

- PubKey 3d4017c3e843895a92b70aa74d1b7ebc9c982cc2ec4968cc0cd55f12af4660c

Outputs

Fee

Ada

Lovelace

8,707

Wallet 2

PubKeyHash 39f713d0a644253f04529421b9f51b0c08979d0...

Ada

Lovelace

9,991,293

Unspent

Balances Carried Forward (as at Slot 15, Tx 0)

	Ada	Lovelace
<b>Beneficial Owner</b>		
<b>Wallet 1</b>		
PubKeyHash 21b31d9154a261620a9f54d46d2271b7bed4db4a5aa58877a47872199		69,999,970
<b>Wallet 2</b>		
PubKeyHash 39f713d0a644253f04529421b9f51b0c08979d08255959a4f3990ae17b5139f		109,991,293
<b>Wallet 3</b>		
PubKeyHash da073a0123bde959d49b3bda9c6103763ac82627d7abdcf54ac29607403e		109,991,293
Script b9762066c2931c241687ef5f945e3e8fc73279083528bf2ec974266076be28		10,000,000

Simulation 1

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 3, Tx 0

Slot 10, Tx 0

Slot 15, Tx 0

Inputs

Script b9762066c2931c241687ef5f945e3e8fc73...

Ada

Lovelace

10,000,000

Created by: Slot 1, Tx 0

Transaction

Slot 15, Tx 0

Tx: b6dc119b661c203be6238577c72a7c47777745fa44dc07b8a6d5b5948ed541

Validity: From Slot 15 (inclusive) to the end of time (inclusive)

Signatures:

- PubKey 3d4017c3e843895a92b70aa74d1b7ebc9c982cc2ec4968cc0cd55f12af4660c

Outputs

Fee

Ada

Lovelace

8,707

Wallet 2

PubKeyHash 39f713d0a644253f04529421b9f51b0c08979d0...

Ada

Lovelace

9,991,293

Unspent

Balances Carried Forward (as at Slot 15, Tx 0)

	Ada	Lovelace
<b>Beneficial Owner</b>		
<b>Wallet 1</b>		
PubKeyHash 21b31d9154a261620a9f54d46d2271b7bed4db4a5aa58877a47872199		69,999,970
<b>Wallet 2</b>		
PubKeyHash 39f713d0a644253f04529421b9f51b0c08979d08255959a4f3990ae17b5139f		109,991,293
<b>Wallet 3</b>		
PubKeyHash da073a0123bde959d49b3bda9c6103763ac82627d7abdcf54ac29607403e		109,991,293
Script b9762066c2931c241687ef5f945e3e8fc73279083528bf2ec974266076be28		10,000,000



## 3.5 Example 2 - Parameterized Contract

We'll start the next example by copying the code from the vesting example into a new module called `Week03.Parameterized`.

### 3.5.1 On-Chain

Note that in the vesting example we used the `Vesting` type as the datum, but it was just fixed, it didn't change. Alternatively, we could have baked it into the contract, so to speak, so that we have a contract where the script itself already contains the beneficiary and deadline information.

All the examples of contracts we have seen so far were fixed. We used a `TypedValidator` as a compile-time constant. The idea of parameterized scripts is that you can have a parameter and, depending on the value of the parameter, you get different values of `TypedValidator`.

So, instead of defining one script, with a single script address, with all UTxOs sitting at the same address, you can define a family of scripts that are parameterized by a given parameter. In our case, this will mean that UTxOs for different beneficiaries and/or deadlines will be a different script addresses, as they will have parameterized validators specific to their parameters rather than specific to the datum of the UTxO.

We are going to demonstrate how to do this by, instead of using datum for the beneficiary and deadline values, using a parameter.

Let's start by renaming `VestingDatum` to something more suitable.

```
data VestingParam = VestingParam
  { beneficiary :: PubKeyHash
  , deadline    :: POSIXTime
  } deriving Show
```

We will also remove the `unstableMakeIsData` call as we don't need this anymore.

The reason we don't need it, is because we are just going to use `()` for the datum in the `mkValidator` function. All the information we require will be in a new argument to `mkValidator`, of type `VestingParam`, which we add at the beginning of the list of arguments.

```
{-# INLINABLE mkValidator #-}
mkValidator :: VestingParam -> () -> () -> ScriptContext -> Bool
mkValidator p () () ctx = traceIfFalse "beneficiary's signature missing" \
  ↪ signedByBeneficiary &&
                                traceIfFalse "deadline not reached" deadlineReached
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    signedByBeneficiary :: Bool
    signedByBeneficiary = txSignedBy info $ beneficiary p

    deadlineReached :: Bool
    deadlineReached = contains (from $ deadline p) $ txInfoValidRange info
```

We also change the `Vesting` type to reflect the change to the datum.

```
data Vesting
instance Scripts.ValidatorTypes Vesting where
```

(continues on next page)

(continued from previous page)

```

type instance DatumType Vesting = ()
type instance RedeemerType Vesting = ()

```

Now, the TypedValidator will no longer be a constant value. Instead it will take a parameter.

Recall that the function mkTypedValidator requires as its first argument the compiled code of a function that takes three arguments and returns a Bool. But now, it has four arguments, so we need to account for that.

```

typedValidator :: VestingParam -> Scripts.TypedValidator Vesting
typedValidator p = Scripts.mkTypedValidator @Vesting

```

Now, what we would like to do is something like this, passing in the new parameter p to mkValidator so that the compiled code within the Oxford brackets would have the correct type.

```

-- this won't work
$(PlutusTx.compile [| mkValidator p |])
$(PlutusTx.compile [| wrap |])
where
  wrap = Scripts.wrapValidator @() @()

```

This code will not work, but before we investigate, let's leave the code as it is for now and make some more changes to the rest of the code.

validator now will take a VestingParam and will return a composed function. The returned function has the effect that any parameter passed to validator would now effectively get passed to the typedValidator function, whose return value would in turned get passed to the validatorScript function.

```

validator :: VestingParam -> Validator
validator = Scripts.validatorScript . typedValidator

```

And the same for valHash and scrAddress.

```

valHash :: VestingParam -> Ledger.ValidatorHash
valHash = Scripts.validatorHash . typedValidator

scrAddress :: VestingParam -> Ledger.Address
scrAddress = scriptAddress . validator

```

Now, let's find out what's wrong with our typedValidator function.

If we try to launch the REPL, we get a compile error.

```

GHC Core to PLC plugin: E043:Error: Reference to a name which is not a local, a builtin,
↳ or an external INLINABLE function: Variable p
No unfolding
Context: Compiling expr: p
Context: Compiling expr: Week03.Parameterized.mkValidator p
Context: Compiling expr at "plutus-pioneer-program-week03-0.1.0.0-inplace:Week03.
↳ Parameterized:(67,10)-(67,48)"

```

The problem is this line.

```

-- this won't work
$(PlutusTx.compile [| mkValidator p |])

```

Recall that everything inside the Oxford brackets must be explicitly known at compile time. Normally it would even need all the code to be written explicitly, but by using the `INLINABLE` pragma on the `mkValidator` function we can reference the function instead. However, it must still be known at compile time, because that's how Template Haskell works - it is executed before the main compiler.

The `p` is not known at compile time, because we intend to supply it at runtime. Luckily there is a way around this.

On the Haskell side, we have our `mkValidator` function and we have `p` of type `VestingParam`. We can compile `mkValidator` to Plutus, but we can't compile `p` to Plutus because we don't know what it is. But, if we could get our hands on the compiled version of `p`, we could apply this compiled version to the compiled `mkValidator`, and this would give us what we want.

This seems to solve nothing, because we still need a compiled version of `p` and we have the same problem that `p` is not known at compile time.

However, `p` is not some arbitrary Haskell code, it's data, so it doesn't contain any function types. If we make the type of `p` an instance of a type class called `Lift`. We can use `liftCode` to compile `p` at runtime to Plutus Core and then, using `applyCode` we can apply the Plutus Core `p` to the Plutus Core `mkValidator`.

## The Lift Class

Let's briefly look at the `Lift` class. It is defined in package `plutus-tx`.

```
module PlutusTx.Lift.Class
```

It only has one function, `Lift`. However, we won't use this function directly.

The importance of the class is that it allows us to, at runtime, lift Haskell values into corresponding Plutus script values. And this is exactly what we need to convert our parameter `p` into code.

We will use a different function, defined in the same package but in a different module.

```
module PlutusTx.Lift
```

The function we will use is called `liftCode`.

```
-- | Get a Plutus Core program corresponding to the given value as a 'CompiledCodeIn',  
--   throwing any errors that occur as exceptions and ignoring fresh names.  
liftCode  
  :: (Lift.Lift uni a, Throwable uni fun, PLC.ToBuiltinMeaning uni fun)  
  => a -> CompiledCodeIn uni fun a  
liftCode x = unsafely $ safeLiftCode x
```

It takes a Haskell value of type `a`, provided `a` is an instance of the `Lift` class, and turns it into a piece of Plutus script code corresponding to the same type.

Now we can fix our validator.

```
typedValidator :: VestingParam -> Scripts.TypedValidator Vesting  
typedValidator p = Scripts.mkTypedValidator @Vesting  
  ($$(PlutusTx.compile [| mkValidator |])) `PlutusTx.applyCode` PlutusTx.liftCode p)  
  ($$(PlutusTx.compile [| wrap |]))  
where  
  wrap = Scripts.wrapValidator @() @()
```

This code is fine, but it won't yet compile, because `VestingParam` is not an instance of `Lift`. To fix this, we can use `makeLift`.

```
PlutusTx.makeLift ''VestingParam
```

And, we need to enable a GHC extension.

```
{-# LANGUAGE MultiParamTypeClasses #-}
```

Now it will compile.

### 3.5.2 Off-Chain

The off-chain code hasn't changed much.

The GiveParams are still the same.

```
data GiveParams = GiveParams
  { gpBeneficiary :: !PubKeyHash
  , gpDeadline    :: !POSIXTime
  , gpAmount      :: !Integer
  } deriving (Generic, ToJSON, FromJSON, ToSchema)
```

VestingSchema has slightly changed because the grab endpoint now relies on knowing the beneficiary and deadline in order to know determine the script address. We know the beneficiary because it will be the public key hash of the wallet that calls grab, but we don't know the deadline, so we must pass it to grab.

```
type VestingSchema =
  Endpoint "give" GiveParams
  .\ Endpoint "grab" POSIXTime
```

The give endpoint is similar to the vesting example, but there are some differences.

Instead of computing the datum, we will construct something of type VestingParam. We also change the reference to the datum in mustPayToTheScript to become (), and we provide the type p to typedValidator as it is no longer a constant.

```
give :: AsContractError e => GiveParams -> Contract w s e ()
give gp = do
  let p = VestingParam
        { beneficiary = gpBeneficiary gp
        , deadline    = gpDeadline gp
        }
      tx = mustPayToTheScript () $ Ada.lovelaceValueOf $ gpAmount gp
  ledgerTx <- submitTxConstraints (typedValidator p) tx
  void $ awaitTxConfirmed $ txId ledgerTx
  logInfo @String $ printf "made a gift of %d lovelace to %s with deadline %s"
    (gpAmount gp)
    (show $ gpBeneficiary gp)
    (show $ gpDeadline gp)
```

In the grab endpoint, there are also some changes.

Recall that earlier we got all the UTxOs sitting at this one script address and that they could be for arbitrary beneficiaries and for arbitrary deadlines. For this reason, we had to filter those UTxOs which were for us and where the deadline had been reached.

We now have the additional parameter, which we'll call d, which represents the deadline. So we can immediately see if the deadline has been reached or not.

If it has not been reached, we write a log message and stop, otherwise we continue and construct the `VestingParam`.

Then, we look up the UTxOs that are sitting at this address. Address is not a constant anymore, it takes a parameter. So, now, we will only get UTxOs which are for us and that have a deadline that has been reached. We don't need to filter anything.

If there are none, we log a message to that effect and stop, otherwise we do more or less what we did before.

```
grab d = do
now    <- currentTime
pkh    <- pubKeyHash <$> ownPubKey
if now < d
  then logInfo @String $ "too early"
  else do
    let p = VestingParam
        { beneficiary = pkh
        , deadline    = d
        }
    utxos <- utxoAt $ scrAddress p
    if Map.null utxos
      then logInfo @String $ "no gifts available"
      else do
        let orefs  = fst <$> Map.toList utxos
            lookups = Constraints.unspentOutputs utxos      <>
                        Constraints.otherScript (validator p)
        tx :: TxConstraints Void Void
        tx  = mconcat [mustSpendScriptOutput oref $ Redeemer
                        mustValidateIn (from now)
        ledgerTx <- submitTxConstraintsWith @Void lookups tx
        void $ awaitTxConfirmed $ txId ledgerTx
        logInfo @String $ "collected gifts"
```

The endpoints function is slightly different due to the new parameter for `grab`.

```
endpoints :: Contract () VestingSchema Text ()
endpoints = (give' `select` grab') >> endpoints
  where
    give' = endpoint @"give" >>= give
    grab' = endpoint @"grab" >>= grab
```

### 3.5.3 Back to the playground

We will now copy and paste this new contract into the playground and setup a new scenario.

The give transactions are the same.

The `grab` is slightly different. In our earlier implementation, one wallet could grab UTxOs with different deadlines provided that the deadlines had passed. Now the deadline is part of the script parameter, so we need to specify it in order to get the script address. This means that Wallet 2 cannot grab the gifts for slots 10 and 20 at the same time, at least not in the way that we have implemented it.

First we can wait until slot 10 and then Wallet 2 should be able to grab its first gift and Wallet 3 should be able to claim its single gift.

The screenshot shows the Plutus Playground Simulator interface. At the top, there's a navigation bar with 'PLUTUS PLAYGROUND' and links for 'Getting Started', 'Tutorials', 'API', 'Privacy', 'Demo files', 'Hello world', 'Starter', 'Game', 'Vesting', 'Crowd Funding', 'Error Handling', and a 'Log In' button. Below this is a 'Simulator' tab with a '+ Return to Editor' link. The main area is divided into 'Wallets' and 'Actions' sections. The 'Wallets' section shows three wallets, each with an 'Opening Balances' field set to '100000000' and 'Available functions' buttons: 'give', 'grab', and 'Pay to Wallet'. The 'Actions' section shows a sequence of five actions: 1. 'Wallet 1: give' with 'gpBeneficiary' set to '39f713d0a644253f04529421b9f51b5', 'gpDeadline' set to '1596059101000', and 'gpAmount' set to '10000000'. 2. 'Wait' with 'Wait For...' selected and 'Blocks' set to '1'. 3. 'Wallet 1: give' with 'gpBeneficiary' set to '39f713d0a644253f04529421b9f51b5', 'gpDeadline' set to '1596059111000', and 'gpAmount' set to '10000000'. 4. 'Wait' with 'Wait For...' selected and 'Blocks' set to '1'. 5. 'Wallet 1: give' with 'gpBeneficiary' set to 'dac073e0123bd5a59d9b3bda9c60', 'gpDeadline' set to '1596059101000', and 'gpAmount' set to '10000000'. Each action has a green checkmark indicating it's valid. There are 'Evaluate' and 'Transactions' buttons at the top right of the simulator area.

We'll add a **grab** for Wallets 2 and 3. Here, we don't need to wait in between each transaction because it is two different wallets.

We then wait until slot 20 and perform Wallet 2's second **grab** and then wait for 1 block, as usual.

So let's see if it works by clicking **Evaluate**.

Take note of the script address for that transaction out at slot 1.

And compare this with the script address for the transaction output at slot 2.

Notice that the script address for the UTxOs is different. In our first version of the vesting contract, the script address was a constant. This meant that all our gifts ended up at the same script address and only the datum in each UTxO was different.

Now, the datum is just `()` and the beneficiary and the deadline are included as part of the script itself, so the addresses are now different depending on the beneficiary and deadline parameters.

For the gift to Wallet 3 we see yet another address.

We see two grabs in slot 10, one by Wallets 2 and one by Wallet 3. The order in which they are processed is not deterministic.

Then, finally in slot 20, Wallet 2 grabs its remaining gift.

And the final balances reflect the transactions that have occurred.

Wallet 1

Opening Balances

Lovelace

100000000

Available functions

give

+

grab

+

Pay to Wallet

+

Wallet 2

Opening Balances

Lovelace

100000000

Available functions

give

+

grab

+

Pay to Wallet

+

Wallet 3

Opening Balances

Lovelace

100000000

Available functions

give

+

grab

+

Pay to Wallet

+

+

Add Wallet

Actions

This is your action sequence. Click 'Evaluate' to run these actions against a simulated blockchain.

1

Wallet 1: give

gpBeneficiary

getPubKeyHash

39f713d0a644253f04529421b9f51b5

gpDeadline

1596059101000

gpAmount

10000000

2

Wait

Wait For... Wait Until...

Blocks

1

3

Wallet 1: give

gpBeneficiary

getPubKeyHash

39f713d0a644253f04529421b9f51b5

gpDeadline

1596059111000

gpAmount

10000000

4

Wait

Wait For... Wait Until...

Blocks

1

5

Wallet 1: give

gpBeneficiary

getPubKeyHash

dac073e0123bdea59d9b3bda9cfe60

gpDeadline

1596059101000

gpAmount

10000000

6

Wait

Wait For... Wait Until...

Slot

10

7

Wallet 2: grab

1596059101000

8

Wallet 3: grab

1596059101000

9

Wait

Wait For... Wait Until...

Slot

20

10

Wallet 2: grab

1596059111000

11

Wait

Wait For... Wait Until...

Blocks

1

+

Add Wait Action

## Simulator

[Return to Editor](#)

Simulation 1

+

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 3, Tx 0

Slot 10, Tx 0

Slot 20, Tx 0

Slot 10, Tx 1

Inputs

Transaction

Slot 0, Tx 0

Tx: 030970255fb42b84c206737064b290e58b167d93f4bcb044328d23625c13bec

Validity: All time

Signatures:None

Forge

Ada Lovelace 300,000,000

Outputs

Wallet 2

PubKeyHash 39f713d0a644253f04529421b9f51b9e08979a0...

Ada Lovelace 100,000,000

Unspent

Wallet 1

PubKeyHash 21fe31db154a261626b8f54046f227167bed4...

Ada Lovelace 100,000,000

Spent in: Slot 1, Tx 0

Wallet 3

PubKeyHash dac073e0123bdea59d9b3bda9cfe6037f63aca8...

Ada Lovelace 100,000,000

Unspent

**Simulator** [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0 Slot 3, Tx 0 Slot 10, Tx 0 Slot 20, Tx 0 Slot 10, Tx 1

#### Inputs

<b>Wallet 1</b> PubKeyHash 21fe31dfa154a261628b854046fd2271b7bed4...	<b>Ada</b> Lovelace 100,000,000 Created by: Slot 0, Tx 0
---	--

#### Transaction

Slot 1, Tx 0

**Tx:** 80dc77577fcc70b0f49ffe63b62c666e4db545934cb7a9f712ddd5a6e39902b

**Validity:** All time

**Signatures:**

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

#### Outputs

<b>Fee</b> Ada Lovelace 10	<b>Wallet 1</b> PubKeyHash 21fe31dfa154a261628b854046fd2271b7bed4...
<b>Ada</b> Lovelace 89,999,990 Spent in: Slot 2, Tx 0	<b>Script 421bc1cb601d8ca9b57f42c7141669af42d...</b> Ada Lovelace 10,000,000 Spent in: Slot 10, Tx 1

Balances Carried Forward (as at Slot 1, Tx 0)

Ada
-----

**Simulator** [Return to Editor](#)

Simulation 1 +

### Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 2, Tx 0 Slot 3, Tx 0 Slot 10, Tx 0 Slot 20, Tx 0 Slot 10, Tx 1

#### Inputs

<b>Wallet 1</b> PubKeyHash 21fe31dfa154a261628b854046fd2271b7bed4...	<b>Ada</b> Lovelace 89,999,990 Created by: Slot 1, Tx 0
---	---

#### Transaction

Slot 2, Tx 0

**Tx:** 3c731b13bc95ada5f250e739d0707417d4c9ef2af97ccc8b87904288d07b0b4

**Validity:** All time

**Signatures:**

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

#### Outputs

<b>Fee</b> Ada Lovelace 10	<b>Wallet 1</b> PubKeyHash 21fe31dfa154a261628b854046fd2271b7bed4...
<b>Ada</b> Lovelace 79,999,980 Spent in: Slot 3, Tx 0	<b>Script d0d28db39e2b1a849ca9440ea12a0cab6...</b> Ada Lovelace 10,000,000 Spent in: Slot 20, Tx 0

Balances Carried Forward (as at Slot 2, Tx 0)

Ada
-----



**Simulator**
[← Return to Editor](#)

---

Simulation 1
+

## Transactions

### Blockchain

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

**Slot 3, Tx 0**

Slot 10, Tx 0

Slot 20, Tx 0

Slot 10, Tx 1

### Inputs

Wallet 1	
PubKeyHash 21fe31dfa154a261626bf854046d2271b7bed4...	
<b>Ada</b> Lovelace	79,999,980
Created by: Slot 2, Tx 0	

### Transaction

Slot 3, Tx 0

**Tx:** 4554b72c799784676c6286d4e3d922415d0df944f35b8f3e311dc889f925d0b

**Validity:** All time

**Signatures:**

- PubKey d75a980182b10ab7054bfed3c964073a0ee172f5daae6325af021ea88707511a

### Outputs

Fee	
<b>Ada</b> Lovelace	10
Wallet 1	
PubKeyHash 21fe31dfa154a261626bf854046d2271b7bed4...	
<b>Ada</b> Lovelace	69,999,970
Unspent	
Script 9db59dca91e23212bbab5b0c8daf31b075...	
<b>Ada</b> Lovelace	10,000,000
Spent in: Slot 10, Tx 0	

### Balances Carried Forward (as at Slot 3, Tx 0)

<b>Ada</b>
------------

Script Hash	Value
Script 9db59dca91e23212bbab5b0c8daf31b0758880397c1fa605711e5bf09711fc28	0
Script d0a28db39e2b1a849ca9440ea12a0cab6b8a188fbaa1773fece7a5568e3420a	10,000,000

### Final Balances

Wallet	Balance
Wallet 1	65,000,000
Wallet 2	115,000,000
Wallet 3	105,000,000

### Logs

```
Validating transaction: 03897025f5fb42b84c296737064b299e50b167d93f4bb0944328d23629c13bec
error: Add xList 1 error
Contract instance for wallet 1: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "give" })) (RawJson {"\ncontents": [{"\ngetEndpointDescription": "\ngive"}, {"\namountValue": {"\ngpBeneficiary":
```



## WEEK 04 - MONADS

---

**Note:** These is a written version of [Lecture #4](#).

In this lecture we learn about Monads. In particular the EmulatorTrace and Contract monads.

---

### 4.1 Overview

We have spent the last two lectures talking about the on-chain part of Plutus - the validation logic that is compiled to Plutus script and actually lives on the blockchain and is executed by nodes that validate a transaction.

There is a lot more to say about that on-chain part.

We haven't looked at more complex examples of validation yet that make more sophisticated use of the context, and we haven't see how native tokens work, yet (Plutus script is also used to validate the minting and burning of native tokens).

We will definitely have to talk about those topics, and come back to that.

However, before we go into too many sophisticated topics of on-chain validation, we mustn't neglect the off-chain part, because it is equally important.

The on-chain part takes care of validation but, in order for there to be something to be validated, we must build a transaction and submit it to the blockchain. And, that is what the off-chain part does.

So, we will start talking about how to write off-chain Plutus code.

Unfortunately there is a slight problem concerning the Haskell features needed.

The on-chain part that we have seen so far is somewhat alien and takes a little getting used to, due to the fact that we have the additional complication of the compilation to Plutus script. But, we don't really have to worry about that if we use the Template Haskell magic. In that case the validator function is just a plain function.

And it is actually a very simple Haskell function from the technical point of view. We don't use any fancy Haskell features to write this function.

One of the reasons for that is the way Plutus compilation works. We have seen how, in order for the compilation to Plutus to succeed, all the code used by the validation function must be available within the Oxford Brackets. This means that all the functions relied on by the *mkValidator* function must use the `INLINABLE` pragma.

```
{-# INLINABLE mkValidator #-}  
mkValidator :: Data -> Data -> Data -> ()  
mkValidator _ _ _ = ()  
  
$(PlutusTx.compile [| mkValidator |])
```

And recall, that because the standard Haskell functions don't have this `INLINABLE` pragma, there is a new Plutus Prelude module that is similar to the standard Haskell Prelude, but with the functions defined with the `INLINABLE` pragma.

But, of course, there are hundreds of Haskell libraries out there and most of them weren't written with Plutus in mind, so we can't use them inside validation. And, that has the effect that the Haskell inside validation will be relatively simple and won't have many dependencies.

## 4.2 Monads

In the off-chain part of Plutus, the situation is reversed. We don't have to worry about compilation to Plutus script - it is just plain Haskell. But, the flip side is that, the way this is implemented, it uses much more sophisticated Haskell features - e.g. so-called effect systems, streaming and, in particular, monads.

All the off-chain code (the wallet code), is written in a special monad - the Contract Monad.

Monads are infamous in the Haskell world. It is normally the first stumbling block for beginning Haskell coders.

There are a lot of tutorials out there that try to explain Monads. Monads get compared to burritos, and all sorts of metaphors are employed to try to explain the concept. But here, let's at least try to give a crash course in monads for those who are new to Haskell.

Before we get to general monads, we will start with *IO*, which is how IO side-effects are handled in Haskell. But, before we get to Haskell, let's look at a mainstream language like Java.

Let's look at the following Java method.

```
public static int foo() {  
    ...  
}
```

This function takes no arguments, and it returns an `int`. Let's imagine it gets called twice in the code.

```
...  
final int a = foo();  
...  
final int b = foo();
```

Now, we note that, so long as we don't know what is going on inside the `foo()` function, the return value of the following expression is unknown.

```
a == b; // true or false? at compile time, we don't know
```

We do not know if `a` is the same as `b` because, in Java, it is perfectly possible that some IO happens inside `foo`. For example, there could be code that asks the user to enter input on the console and uses this to compute the return value.

This means that, in order to reason about the code, we need to look inside `foo`, which makes testing, for example, more difficult. And it means that, if the first call to `foo` returns, for example, 13 - we cannot just replace all other calls to `foo` with the known return value of 13.

In Haskell the situation is very different because Haskell is a pure functional language. The equivalent signature in Haskell would be something like:

```
foo :: Int  
foo = ...
```

Now, if we have a situation where we call `foo` twice, even though we don't know what the value of `foo` is, we know for sure that the two return values will be the same.

This is a very important feature that is called *referential transparency*. There are, in fact, some escape hatches to get around this, but we can ignore this.

This makes tasks such as refactoring and testing much easier.

This is all very well, but you need side-effects in order to have an effect on the world. Otherwise, all your program does is heat up the processor.

You need input and output. You must be able to write output to the screen, or read input from the keyboard, or a network connection, or a file, for example.

There is a famous [video by Simon Peyton-Jones called Haskell Is Useless](#) which explains that it is beautiful mathematically to have a pure, side effect-free language, but in the end you do need side effects to make anything happen.

And Haskell does have a way to handle side effects and that is the IO Monad. But, don't worry about the monad part just yet.

Here is how we do it in Haskell.

```
foo :: IO Int
foo = ...
```

`IO` is a type constructor that takes one argument, like some other examples of type constructors such as `Maybe` and `List`. However, unlike those examples, `IO` is special, in the sense that you can't implement it in the language itself. It is a built-in primitive.

The return value `IO Int` tells us that this is a recipe to compute an `Int`, and this recipe can cause side effects. A list of instructions telling the computer what to do in order to end up with an `Int`.

It is important to notice that referential transparency is not broken here. The result of the evaluation of `foo` is the recipe itself, not the `Int` value. And as the recipe is always the same, referential transparency is maintained.

The only way to actually execute such a recipe in a Haskell program is from the main entry point of the program - the `main` function. You can also execute `IO` actions in the REPL.

## 4.2.1 Hello World

Hello World in Haskell looks like this:

```
main :: IO ()
main = putStrLn "Hello, world!"
```

Here, `main` is a recipe that performs some side effects and returns `Unit` - nothing of interest.

Let's look at `putStrLn` in the REPL. We see that it is an `IO` action that takes a `String` and returns no interesting result.

```
Prelude Week04.Contract> :t putStrLn
putStrLn :: String -> IO ()

Prelude Week04.Contract> :t putStrLn "Hello, world!"
putStrLn "Hello, world!" :: IO ()
```

We can also run this. Open up the `app/Main.sh` file and edit the `main` function so it reads:

```
main :: IO ()
main = putStrLn "Hello, world!"
```

Then run

```
cabal run hello
```

We will take a quick look at the cabal file now.

In previous lectures we only needed the *library* section in the *plutus-pioneer-program-week04.cabal* file as we were dealing only with library functions. Now, we need to add an *executable* stanza.

```
executable hello
hs-source-dirs:      app
main-is:             hello.hs
build-depends:       base ^>=4.14.1.0
default-language:    Haskell2010
ghc-options:         -Wall -O2
```

This specifies the source directory and which file holds the main function. Normally the file name must match the module name, but the *main* is an exception.

Rather than just asking for the type of *putStrLn*, we can run it in the REPL. As mentioned, the REPL allows us to execute IO actions.

```
Prelude Week04.Contract> putStrLn "Hello, world!"
Hello, world!
```

### 4.2.2 getLine

Let's look at *getLine*

```
Prelude Week04.Contract> :t getLine
getLine :: IO String
```

This shows that it is a recipe, possibly producing side-effects, that, when executed will produce a *String*. In the case of *getLine*, the side-effect in question is that it will wait for user input from the keyboard.

If we execute *getLine* in the REPL.

```
Prelude Week04.Contract> getLine
```

It waits for keyboard input. Then, if we enter something, it returns the result.

```
Haskell
"Haskell"
```

There are a variety of IO actions defined in Haskell to do all sorts of things like reading files, writing files, reading from and writing to sockets.

But no matter how many predefined actions you have, that will never be enough to achieve something complex, so there must be a way to combine these primitive, provided IO actions into bigger, more complex recipes.

One thing we can do is make use of the *Functor* type instance of IO. Let's look at the type instances of *IO* in the REPL.

```
Prelude Week04.Contract> :i IO
type IO :: * -> *
newtype IO a
= ghc-prim-0.6.1:GHC.Types.IO (ghc-prim-0.6.1:GHC.Prim.State#
```

(continues on next page)

(continued from previous page)

```

ghc-prim-0.6.1:GHC.Prim.RealWorld
-> (# ghc-prim-0.6.1:GHC.Prim.State#
    ghc-prim-0.6.1:GHC.Prim.RealWorld,
    a #))
-- Defined in 'ghc-prim-0.6.1:GHC.Types'
instance Applicative IO -- Defined in 'GHC.Base'
instance Functor IO -- Defined in 'GHC.Base'
instance Monad IO -- Defined in 'GHC.Base'
instance Monoid a => Monoid (IO a) -- Defined in 'GHC.Base'
instance Semigroup a => Semigroup (IO a) -- Defined in 'GHC.Base'
instance MonadFail IO -- Defined in 'Control.Monad.Fail'

```

We see the dreaded *Monad* instance, but we also see a *Functor* instance. *Functor* is a very important type class in Haskell. If we look at it in the REPL:

```

Prelude Week04.Contract> :i Functor
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
{-# MINIMAL fmap #-}
-- Defined in 'GHC.Base'
instance Functor (Either a) -- Defined in 'Data.Either'
instance Functor [] -- Defined in 'GHC.Base'
instance Functor Maybe -- Defined in 'GHC.Base'
instance Functor IO -- Defined in 'GHC.Base'
instance Functor ((->) r) -- Defined in 'GHC.Base'
instance Functor ((,,,) a b c) -- Defined in 'GHC.Base'
instance Functor ((,,) a b) -- Defined in 'GHC.Base'
instance Functor ((,) a) -- Defined in 'GHC.Base'

```

The important method here is *fmap*. The second function (*<\$*) is a convenience function.

```
fmap :: (a -> b) -> f a -> f b
```

This function, *fmap*, that all *Functors* have tells us that, if we give it has access to a function that can turn an *a* into a *b*, then it can turn an *f a* into an *f b* for us. Here, we are interested in the case where *f* is *IO*.

If we specialized the function for *IO*, we would have a function like:

```
fmap' :: (a -> b) -> IO a -> IO b
```

How does that work. Well, *IO a* is a recipe that has side effects and produces an *a*. So, how do we get a *b* out of that? We perform the recipe, but, before return the *a*, we apply the *(a -> b)* function to to *a* and return the result, which is the *b*.

In the REPL, let's look at the *toUpper* function.

```

Prelude Week04.Contract> import Data.Char
Prelude Data.Char Week04.Contract> :t toUpper
toUpper :: Char -> Char
Prelude Data.Char Week04.Contract> toUpper 'q'
'Q'

```

If we want to apply that to a *String* rather than a *Char* we can use the *map* function. *Strings* in Haskell are just lists of *Chars*.

```
Prelude Data.Char Week04.Contract> map toUpper "Haskell"  
"HASKELL"
```

The *map toUpper* function is a function from *String* to *String*.

```
Prelude Data.Char Week04.Contract> :t map toUpper  
map toUpper :: [Char] -> [Char]
```

And we can use this in combination with *fmap*. If we use *map toUpper* as our function to convert an *a* to a *b*, we can see what the type of output of *fmap* would be when applied to an *IO a*.

```
Prelude Data.Char Week04.Contract> :t fmap (map toUpper) getLine  
fmap (map toUpper) getLine :: IO [Char]
```

Let's see it in action.

```
Prelude Data.Char Week04.Contract> fmap (map toUpper) getLine  
haskell  
"HASKELL"
```

We can also use the *>>* operator. This chains two *IO* actions together, ignoring the result of the first. In the following example, both actions will be performed in sequence.

```
Prelude Week04.Contract> putStrLn "Hello" >> putStrLn "World"  
Hello  
World
```

Here, there is no result from *putStrLn*, but if there were, it would have been ignored. Its side effects would have been performed, its result ignored, then the second *putStrLn* side effects would have been performed before returning the result of the second call.

Then, there is an important operator that does not ignore the result of the first *IO* action, and that is called *bind*. It is written as the *>=>* symbol.

```
Prelude Week04.Contract> :t (>=>)  
(>=>) :: Monad m => m a -> (a -> m b) -> m b
```

We see the *Monad* constraint, but we can ignore that for now and just think of *IO*.

What this says is that if I have a recipe that performs side effects then gives me a result *a*, and given that I have a function that takes an *a* and gives me back a recipe that returns a *b*, then I can combine the recipe *m a* with the recipe *m b* by taking the value *a* and using it in the recipe that results in the value *b*.

An example will make this clear.

```
Prelude Week04.Contract> getLine >=> putStrLn  
Haskell  
Haskell
```

Here, the function *getLine* is of type *IO String*. The return value *a* is passed to the function *(a -> m b)* which then generates a recipe *putStrLn* with an input value of *a* and an output of type *IO ()*. Then, *putStrLn* performs its side effects and returns *Unit*.

There is another, very important, way to create *IO* actions, and that is to create recipes that immediately return results without performing any side effects.



That is done with a function called *return*.

```
Prelude Week04.Contract> :t return
return :: Monad m => a -> m a
```

Again, it is general for any Monad, we only need to think about *IO* right now.

It takes a value *a* and returns a recipe that produces the value *a*. In the case of *return*, the recipe does not actually create any side effects.

For example:

```
Prelude Week04.Contract> return "Haskell" :: IO String
"Haskell"
```

We needed to specify the return type so that the REPL knows which Monad we are using:

```
Prelude Week04.Contract> :t return "Haskell" :: IO String
return "Haskell" :: IO String :: IO String

Prelude Week04.Contract> :t return "Haskell"
return "Haskell" :: Monad m => m [Char]
```

If we now go back to our *main* program, we can now write relatively complex *IO* actions. For example, we can define an *IO* action that will ask for two strings and print result of concatenating those two strings to the console.

```
main :: IO ()
main = bar

bar :: IO ()
bar = getLine >=> \s ->
      getLine >=> \t ->
      putStrLn (s ++ t)
```

And then, when we run it, the program will wait for two inputs and then output the concatenated result.

```
cabal run hello
one
two
onetwo
```

This is enough now for our purposes, although we won't need the *IO* Monad until perhaps later in the course when we talk about actually deploying Plutus contracts. However, the *IO* Monad is an important example, and a good one to start with.

So, for now, let's completely forget about *IO* and just write pure, functional Haskell, using the *Maybe* type.

### 4.2.3 Maybe

The *Maybe* type is one of the most useful types in Haskell.

```
Prelude Week04.Contract> :i Maybe
type Maybe :: * -> *
data Maybe a = Nothing | Just a
    -- Defined in 'GHC.Maybe'
instance Applicative Maybe -- Defined in 'GHC.Base'
instance Eq a => Eq (Maybe a) -- Defined in 'GHC.Maybe'
instance Functor Maybe -- Defined in 'GHC.Base'
instance Monad Maybe -- Defined in 'GHC.Base'
instance Semigroup a => Monoid (Maybe a) -- Defined in 'GHC.Base'
instance Ord a => Ord (Maybe a) -- Defined in 'GHC.Maybe'
instance Semigroup a => Semigroup (Maybe a)
    -- Defined in 'GHC.Base'
instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
instance Read a => Read (Maybe a) -- Defined in 'GHC.Read'
instance Foldable Maybe -- Defined in 'Data.Foldable'
instance Traversable Maybe -- Defined in 'Data.Traversable'
instance MonadFail Maybe -- Defined in 'Control.Monad.Fail'
```

It is often called something like *Optional* in other programming languages.

It has two constructors - *Nothing*, which takes no arguments, and *Just*, which takes one argument.

```
data Maybe a = Nothing | Just a
```

Let's look at an example.

In Haskell, if you want to pass a *String* to a value that has a *read* instance, you will normally do this with the *read* function.

```
Week04.Maybe> read "42" :: Int
42
```

But, *read* is a bit unpleasant, because if we have something that can't be parsed as an *Int*, then we get an error.

```
Week04.Maybe> read "42+u" :: Int
*** Exception: Prelude.read: no parse
```

Let's import *readMaybe* to do it in a better way.

```
Prelude Week04.Maybe> import Text.Read (readMaybe)
Prelude Text.Read Week04.Contract>
```

The function *readMaybe* does the same as *read*, but it returns a *Maybe*, and in the case where it cannot parse, it will return a *Maybe* created with the *Nothing* constructor.

```
Prelude Text.Read Week04.Contract> readMaybe "42" :: Maybe Int
Just 42

Prelude Text.Read Week04.Contract> readMaybe "42+u" :: Maybe Int
Nothing
```

Let's say we want to create a new function that returns a *Maybe*.

```
foo :: String -> String -> String -> Maybe Int
```

The idea is that the function should try to parse all three *Strings* as *Ints*. If all the *Strings* can be successfully parsed as *Ints*, then we want to add those three *Ints* to get a sum. If one of the parses fails, we want to return *Nothing*.

One way to do that would be:

```
foo :: String -> String -> String -> Maybe Int
foo x y z = case readMaybe x of
  Nothing -> Nothing
  Just k   -> case readMaybe y of
    Nothing -> Nothing
    Just l   -> case readMaybe z of
      Nothing -> Nothing
      Just m   -> Just (k + l + m)
```

Let's see if it works. First, the case where it succeeds:

```
Prelude Week04.Contract> :l Week04.Maybe
Prelude Week04.Maybe> foo "1" "2" "3"
Just 6
```

But, if one of the values can't be parsed, we get *Nothing*:

```
Prelude Week04.Maybe> foo "" "2" "3"
Nothing
```

The code is not ideal because we repeat the same pattern three times. Each time we have to consider the two cases - whether the result of the read is a *Just* or a *Nothing*.

Ask Haskellers, we hate repetition like this.

The thing we want to do is very simple. We want to pass the three *Strings* and add the result, but with all those cases it is very noisy and very ugly. We want to abstract away this pattern.

One way to do that would be to define something like:

```
bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bindMaybe Nothing = Nothing
bindMaybe (Just x) f = f x
```

Let's write the same function again using *bindMaybe*.

```
foo' :: String -> String -> String -> Maybe Int
foo' x y z = readMaybe x `bindMaybe` \k ->
  readMaybe y `bindMaybe` \l ->
  readMaybe z `bindMaybe` \m ->
  Just (k + l + m)
```

And then, in the REPL, we get the same results for *foo'* as we got for *foo*.

```
Prelude Week04.Maybe> foo "1" "2" "3"
Just 6

Prelude Week04.Maybe> foo "" "2" "3"
Nothing
```

This does exactly the same as *foo*, but it is much more compact, there is far less noise, and the business logic is much clearer.

It may, or may not, help to view the function with it not being used with infix notation:

```
Prelude Text.Read Week04.Maybe> bindMaybe (readMaybe "42" :: Maybe Int) (\x -> Just x)
Just 42
```

Here you can see the function clearly taking the *Maybe* and then the function that takes the *a* from the *Maybe* and uses it as the input to a function that returns a new *Maybe*.

This produces nothing useful, until we add the second *readMaybe*

```
Prelude Text.Read Week04.Maybe> bindMaybe (readMaybe "42" :: Maybe Int) (\x -> bindMaybe
↳ (readMaybe "5" :: Maybe Int) (\y -> Just (y + x)))
Just 47
```

In some ways *Nothing* is a bit like an exception in other languages. If any of the computations returns *Nothing*, the remainder of the computations in the block are not performed and *Nothing* is returned.

### 4.2.4 Either

Another very useful type in Haskell is the *Either* type.

```
Prelude Week04.Contract> :i Either
type Either :: * -> * -> *
data Either a b = Left a | Right b
    -- Defined in 'Data.Either'
instance Applicative (Either e) -- Defined in 'Data.Either'
instance (Eq a, Eq b) => Eq (Either a b)
    -- Defined in 'Data.Either'
instance Functor (Either a) -- Defined in 'Data.Either'
instance Monad (Either e) -- Defined in 'Data.Either'
instance (Ord a, Ord b) => Ord (Either a b)
    -- Defined in 'Data.Either'
instance Semigroup (Either a b) -- Defined in 'Data.Either'
instance (Show a, Show b) => Show (Either a b)
    -- Defined in 'Data.Either'
instance (Read a, Read b) => Read (Either a b)
    -- Defined in 'Data.Either'
instance Foldable (Either a) -- Defined in 'Data.Foldable'
instance Traversable (Either a) -- Defined in 'Data.Traversable'
```

*Either* takes two parameters, *a* and *b*. Like *Maybe* it has two constructors, but unlike *Maybe* both take a value. It can *Either* be an *a* or a *b*. The two constructors are *Left* and *Right*.

For example:

```
Prelude Week04.Contract> Left "Haskell" :: Either String Int
Left "Haskell"
```

Or

```
Prelude Week04.Contract> Right 7 :: Either String Int
Right 7
```

If we take the exception analogy a little further, then one issue with *Maybe* is that if we return *Nothing*, there is no error message. But, if we want something that gives a message, we can replace *Maybe* with an *Either* type.

In that case, *Right* can correspond to *Just* and *Left* can correspond to an error, as *Nothing* did. But, depending on what type we choose for *a*, we can give appropriate error messages.

Let's define something called *readEither* and see what it does when it can and when it cannot parse its input.

```
readEither :: Read a => String -> Either String a
readEither s = case readMaybe s of
  Nothing -> Left $ "can't parse: " ++ s
  Just a   -> Right a
```

```
Prelude Week04.Either> readEither "42" :: Either String Int
Right 42
```

```
Prelude Week04.Either> readEither "42+u" :: Either String Int
Left "can't parse: 42+u"
```

Using this, we can now rewrite *foo* in terms of *Either*. First, using the long-winded method:

```
foo :: String -> String -> String -> Either String Int
foo x y z = case readEither x of
  Left err -> Left err
  Right k   -> case readEither y of
    Left err -> Left err
    Right l   -> case readEither z of
      Left err -> Left err
      Right m   -> Right (k + l + m)
```

Let's try it. First, the happy path:

```
Prelude Week04.Either> foo "1" "2" "3"
Right 6
```

Then, when we have a problem:

```
Prelude Week04.Either> foo "ays" "2" "3"
Left "can't parse: ays"
```

But, we have the same problem that we had with *Maybe*; we have a lot of repetition.

The solution is similar.

```
bindEither :: Either String a -> (a -> Either String b) -> Either String b
bindEither (Left err) _ = Left err
bindEither (Right x)   f = f x

foo' :: String -> String -> String -> Either String Int
foo' x y z = readEither x `bindEither` \k ->
  readEither y `bindEither` \l ->
  readEither z `bindEither` \m ->
  Right (k + l + m)
```

You can run this again in the REPL and it will behave in the same way as its long-winded version.

### 4.2.5 Writer

So far we have looked at three examples: *IO a*, *Maybe a* and *Either String a*. *IO a* represents plans that can involve side effects and, when executed, produce an *a*. *Maybe a* and *Either String a* represent computations that can produce an *a* but can also fail. The difference between *Maybe* and *Either* is just that *Maybe* does not produce any error message, but *Either* does.

Now let's look at a completely different example that captures the idea of computations that can also produce log output.

We can represent that with a type.

```
data Writer a = Writer a [String]
  deriving Show
```

As an example, let's write a function that returns a *Writer* for an *Int* and writes a log message.

```
number :: Int -> Writer Int
number n = Writer n $ ["number: " ++ show n]
```

In the REPL:

```
Prelude Week04.Writer> number 42
Writer 42 ["number: 42"]
```

Now, let's do something similar to that which we have done with *Maybe* and *Either*.

Let's write a function that takes three logging computations that each produce an *Int* and we want to return a single computation that produces the sum of those *Ints*.

```
foo :: Writer Int -> Writer Int -> Writer Int -> Writer Int
foo (Writer k xs) (Writer l ys) (Writer m zs) =
  Writer (K + l + m) $ xs ++ ys ++ zs
```

In the REPL:

```
Prelude Week04.Writer> foo (number 1) (number 2) (number 3)
Writer 6 ["number: 1","number: 2","number: 3"]
```

Now, let's write another useful function that takes a list of message and produces a *Writer* with no useful result.

```
tell :: [String] -> Writer ()
tell = Writer ()
```

Now, we can update *foo* to add an extra log message showing the sum of the numbers.

```
foo :: Writer Int -> Writer Int -> Writer Int -> Writer Int
foo (Writer k xs) (Writer l ys) (Writer m zs) =
  let
    s = k + l + m
    Writer _ us = tell ["sum: " ++ show s]
  in
    Writer s $ xs ++ ys ++ zs ++ us
```

In the REPL:

```
Prelude Week04.Writer> foo (number 1) (number 2) (number 3)
Writer 6 ["number: 1","number: 2","number: 3","sum: 6"]
```

As before, we can write a bind function:

```
bindWriter :: Writer a -> (a -> Writer b) -> Writer b
bindWriter (Writer a xs) f =
let
    Writer b ys = f a
in
    Writer b $ xs ++ ys
```

Here, the *bindWriter* function is returning the *Writer b* and producing log messages which are a concatenation of the *xs* that we pattern matched on input, and the *ys* that we pattern matched when calling *f a* in order to produce the *Writer b*.

Now, we can rewrite *foo* using *bindWriter* and make it much nicer.

```
foo' :: Writer Int -> Writer Int -> Writer Int -> Writer Int
foo' x y z = x `bindWriter` \k ->
    y `bindWriter` \l ->
    z `bindWriter` \m ->
    let s = k + l + m
    in tell ["sum: " ++ show s] `bindWriter` \_ ->
        Writer s []
```

What we did with *foo* before, we can now do with *foo'*, and we get the same result.

```
Prelude Week04.Writer> foo' (number 1) (number 2) (number 3)
Writer 6 ["number: 1","number: 2","number: 3","sum: 6"]
```

Admittedly, it is longer than it was before, but it is much nicer. We no longer need to do the pattern matching to extract the messages. We don't have to explicitly combine the log messages, where we could make a mistake and forget one, or get the order wrong. Instead, we abstract all that away and can just concentrate on the business logic.

Although the pattern is the same as with *Maybe* and *Either*, note that the special aspect of these computations is completely different. With *Maybe* and *Either* we dealt with the notion of failure, whereas here, with the *Writer*, there is no failure, but we instead have additional output.

## 4.2.6 What is a Monad?

Now, we are in a position to explain what a Monad is.

Looking back at the four examples, what did they have in common? In all four cases, We had a type constructor with one type parameter - *IO*, *Maybe*, *Either String* and *Writer* all take a type parameter.

And, for all four of these examples, we had a bind function. For *IO*, we had the *>>=* function and for the others we had the bind functions that we wrote ourselves.

```
bindWriter :: Writer a -> (a -> Writer b) -> Writer b
bindEither :: Either String a -> (a -> Either String b) -> Either String b
bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
```

How the bind works depends on the case. In the case of *IO* it is built-in magic, but you can think of it as just combining the two plans describing the actions to take during computation. For *bindMaybe* and *bindEither* the logic is for the whole plan to fail if any part of it fails, and for *bindWriter*, the logic was to combine the list of log messages.

And that is the main idea of Monads. It's a concept of computation with some additional side effects, and the ability to bind two such computations together.

There is another aspect that we briefly mentioned in the case of *IO* but not for the other examples - another thing that we can always do.

Whenever we have such a concept of computation with side effects, we also always have the ability to produce a computation of this kind that *doesn't* have any side effects.

In the example of *IO*, this was done with *return*. Given an *a*, you can create an *IO a* which is the recipe that always simply returns the *a* with no side effects. Each of the other example has this ability as well, as shown below.

```
return      :: a -> IO a
Just        :: a -> Maybe a
Right       :: a -> Either String a
(\a -> Writer a []) :: a -> Writer a
```

And it is the combination of these two features that defines a Monad.

- the ability to bind two computations together
- the possibility to construct a computation from a pure value without making use of any of the potential side effects

If we look in the REPL:

```
Prelude Week04.Contract> :i Monad
type Monad :: (* -> *) -> Constraint
class Applicative m => Monad m where
(>>=) :: m a -> (a -> m b) -> m b
(>>)  :: m a -> m b -> m b
return :: a -> m a
{-# MINIMAL (>>=) #-}
-- Defined in 'GHC.Base'
instance Monad (Either e) -- Defined in 'Data.Either'
instance Monad [] -- Defined in 'GHC.Base'
instance Monad Maybe -- Defined in 'GHC.Base'
instance Monad IO -- Defined in 'GHC.Base'
instance Monad ((->) r) -- Defined in 'GHC.Base'
instance (Monoid a, Monoid b, Monoid c) => Monad ((,,) a b c)
-- Defined in 'GHC.Base'
instance (Monoid a, Monoid b) => Monad ((,,) a b)
-- Defined in 'GHC.Base'
instance Monoid a => Monad ((,) a) -- Defined in 'GHC.Base'
```

We see the bind function

```
(>>=) :: m a -> (a -> m b) -> m b
```

And the *return* function that takes a pure value and turns it into a computation that has potential for side effects, but does not use them.

```
return :: a -> m a
```

The other function *>>* can easily be defined in terms of *>>=*, but is provided for convenience.

```
(>>) :: m a -> m b -> m b
```

What this function does is to throw away the result of the first computation, so you could define it in terms of *>>=* by just ignoring the argument to the function parameter.



There's another technical computation. We see that *Monad* has the super class *Applicative*, so every *Monad* is *Applicative*.

```
Prelude Week04.Contract> :i Applicative
type Applicative :: (* -> *) -> Constraint
class Functor f => Applicative f where
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
GHC.Base.liftA2 :: (a -> b -> c) -> f a -> f b -> f c
(<*>) :: f a -> f b -> f b
(<*>) :: f a -> f b -> f a
{-# MINIMAL pure, ((<*>) | liftA2) #-}
-- Defined in 'GHC.Base'
instance Applicative (Either e) -- Defined in 'Data.Either'
instance Applicative [] -- Defined in 'GHC.Base'
instance Applicative Maybe -- Defined in 'GHC.Base'
instance Applicative IO -- Defined in 'GHC.Base'
instance Applicative ((->) r) -- Defined in 'GHC.Base'
instance (Monoid a, Monoid b, Monoid c) =>
  Applicative ((,,) a b c)
-- Defined in 'GHC.Base'
instance (Monoid a, Monoid b) => Applicative ((,,) a b)
-- Defined in 'GHC.Base'
instance Monoid a => Applicative ((,) a) -- Defined in 'GHC.Base'
```

We see it has a bunch of functions, but we only need the first two.

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

The function *pure* has the same type signature as *return*. Then there is *<\*>* (pronounced 'ap') which looks a bit more complicated. But, the truth is that, once you have *return* and *>=>* in a *Monad*, we can easily define both *pure* and *<\*>*.

We see that *Applicative* also has a superclass *Functor*.

```
Prelude Week04.Contract> :i Functor
type Functor :: (* -> *) -> Constraint
class Functor f where
fmap :: (a -> b) -> f a -> f b
(<$) :: a -> f b -> f a
{-# MINIMAL fmap #-}
-- Defined in 'GHC.Base'
instance Functor (Either a) -- Defined in 'Data.Either'
instance Functor [] -- Defined in 'GHC.Base'
instance Functor Maybe -- Defined in 'GHC.Base'
instance Functor IO -- Defined in 'GHC.Base'
instance Functor ((->) r) -- Defined in 'GHC.Base'
instance Functor ((,,) a b c) -- Defined in 'GHC.Base'
instance Functor ((,,) a b) -- Defined in 'GHC.Base'
instance Functor ((,) a) -- Defined in 'GHC.Base'
```

As we mentioned in the context of *IO*, *Functor* has the *fmap* function which, given a function from *a* to *b* will turn an *f a* into an *f b*.

The prototypical example for *fmap* is lists where *fmap* is just *map*. Given a function from *a* to *b*, you can create a list of type *b* from a list of type *a* by applying the *map* function to each of the elements of the list.

Again, once you have *return* and *>>=*, it is easy to define *fmap*.

So, whenever you want to define a Monad, you just define *return* and *>>=*, and to make the compiler happy and to give instances for *Functor* and *Applicative*, there's always a standard way of doing it.

We can do this in the example of *Writer*.

```
import Control.Monad

instance Functor Writer where
    fmap = liftM

instance Applicative Writer where
    pure = return
    (<*>) = ap

instance Monad Writer where
    return a = Writer a []
    (>>=) = bindWriter
```

We don't have to do the same for *Maybe*, *Either* or *IO* because they are already Monads defined by the Prelude.

### 4.2.7 Why Is This useful?

It is always useful, in general, to identify a common pattern and give it a name.

But, maybe the most important advantage is that there are lots of functions that don't care which Monad we are dealing with - they will work with all Monads.

Let's generalize the example where we compute the sum of three integers. We use a *let* in the example below for reasons that will become clear in moment.

```
threeInts :: Monad m => m Int -> m Int -> m Int -> m Int
threeInts mx my mz =
    mx >>= \k ->
    my >>= \l ->
    mz >>= \m ->
    let s = k + l + m in return s
```

Now we have this function, we can return to the *Maybe* example and rewrite it.

```
foo'' :: String -> String -> String -> Maybe Int
foo'' x y z = threeInts (readMaybe x) (readMaybe y) (readMaybe z)
```

We can do the same for the *Either* example.

```
foo'' :: String -> String -> String -> Either String Int
foo'' x y z = threeInts (readEither x) (readEither y) (readEither z)
```

The *Writer* example is not exactly the same.

If we are happy not to have the log message for the sum, it is very simple as it is already an instance of *threeInts*.

```
foo'' :: Writer Int -> Writer Int -> Writer Int -> Writer Int
foo'' x y z = threeInts
```

However, if we want the final log message, it becomes a little more complicated.

```
foo' :: Writer Int -> Writer Int -> Writer Int -> Writer Int
foo' x y z = do
  s <- threeInts x y z
  tell ["sum: " ++ show s]
  return s
```

If you look into the `Control.Monad` module in the standard Haskell Prelude, you will see that there are many useful functions that you can use for all Monads.

One way to think about a Monad is as a computation with a super power.

In the case of *IO*, the super power would be having real-world side-effects. In the case of *Maybe*, the super power is being able to fail. The super power of *Either* is to fail with an error message. And in the case of *Writer*, the super power is to log messages.

There is a saying in the Haskell community that Haskell has an overloaded semi-colon. The explanation for this is that in many imperative programming languages, you have semi-colons that end with a semi-colon - each statement is executed one after the other, each separated by a semi-colon. But, what exactly the semi-colon means depends on the language. For example, there could be an exception, in which case computation would stop and wouldn't continue with the next lines.

In a sense, *bind* is like a semi-colon. And the cool thing about Haskell is that it is a programmable semi-colon. We get to say what the logic is for combining two computations together.

Each Monad comes with its own “semi-colon”.

#### 4.2.8 ‘do’ notation

Because this pattern is so common and monadic computations are all over the place, there is a special notation for this in Haskell, called *do* notation.

It is syntactic sugar. Let's rewrite *threeInts* using *do* notation.

```
threeInts' :: Monad m => m Int -> m Int -> m Int -> m Int
threeInts' mx my mz = do
  k <- mx
  l <- my
  m <- mz
  let s = k + l + m
  return s
```

This does exactly the same thing as the non-*do* version, but it has less noise.

Note that the *let* statement does not use an *in* part. It does not need to inside a *do* block.

And that's Monads. There is a lot more to say about them but hopefully you now have an idea of what Monads are and how they work.

Often you are in a situation where you want several effects at once - for example you may want optional failure *and* log messages. There are ways to do that in Haskell. For example there are Monad Transformers where one can basically build custom Monads with the features that you want.

There are other approaches. One is called Effect Systems, which has a similar objective. And this is incidentally what Plutus uses for important Monads. In particular the Contact Monad in the wallet, and the Trace Monad which is used to test Plutus code.

The good news is that you don't need to understand Effect Systems to work with these Monads. You just need to know that you are working with a Monad, and what super powers it has.

## 4.3 Plutus Monads

Now that we have seen how to write monadic code, either by using `bind` and `return` or by using `do` notation, we can look at a very important Monad, namely the Contract Monad, which you may have already noticed in previous code examples.

The Contract Monad defines code that will run in the wallet, which is the off-chain part of Plutus.

But, before we go into details, we will talk about a second Monad, the `EmulatorTrace` monad.

### 4.3.1 The `EmulatorTrace` Monad

You may have wondered if there is a way to execute Plutus code for testing purposes without using the Plutus Playground. There is indeed, and this is done using the `EmulatorTrace` Monad.

You can think of a program in this monad as what we do manually in the *simulator* tab of the playground. That is, we define the initial conditions, we define the actions such as which wallets invoke which endpoints with which parameters and we define the waiting periods between actions.

The relevant definitions are in the package `plutus-contract` in module `Plutus.Trace.Emulator`.

```
module Plutus.Trace.Emulator
```

The most basic function is called `runEmulatorTrace`.

```
-- | Run an emulator trace to completion, returning a tuple of the final state
-- of the emulator, the events, and any error, if any.
runEmulatorTrace
  :: EmulatorConfig
  -> EmulatorTrace ()
  -> ([EmulatorEvent], Maybe EmulatorError, EmulatorState)
runEmulatorTrace cfg trace =
  (\(xs :> (y, z)) -> (xs, y, z))
  $ run
  $ runReader ((initialDist . _initialChainState) cfg)
  $ foldEmulatorStreamM (generalize list)
  $ runEmulatorStream cfg trace
```

It gets something called an `EmulatorConfig` and an `EmulatorTrace ()`, which is a pure computation where no real-world side effects are involved. It is a pure function that executes the trace on an emulated blockchain, and then gives a result as a list of `EmulatorEvent`s, maybe an error, if there was one, and then finally the final `*EmulatorState`.

`EmulatorConfig` is defined in a different module in the same package:

```
module Wallet.Emulator.Stream

data EmulatorConfig =
  EmulatorConfig
    { _initialChainState      :: InitialChainState -- ^ State of the blockchain at the
    ↪ beginning of the simulation. Can be given as a map of funds to wallets, or as a block,
    ↪ of transactions.
    } deriving (Eq, Show)

type InitialChainState = Either InitialDistribution Block
```

We see it only has one field, which is of type `InitialChainState` and it is either `InitialDistribution` or `Block`.

*InitialDistribution* is defined in another module in the same package, and it is a type synonym for a map of key value pairs of *Wallet\*s* to *\*Value\*s*, as you would expect. *\*Value* can be either lovelace or native tokens.

```
module Plutus.Contract.Trace

type InitialDistribution = Map Wallet Value
```

In the same module, we see something called *defaultDist* which returns a default distribution for all wallets. It does this by passing the 10 wallets defined by *allWallets* to *defaultDistFor* which takes a list of wallets.

```
-- | The wallets used in mockchain simulations by default. There are
-- ten wallets because the emulator comes with ten private keys.
allWallets :: [EM.Wallet]
allWallets = EM.Wallet <$> [1 .. 10]

defaultDist :: InitialDistribution
defaultDist = defaultDistFor allWallets

defaultDistFor :: [EM.Wallet] -> InitialDistribution
defaultDistFor wallets = Map.fromList $ zip wallets (repeat (Ada.lovelaceValueOf 100_000_
->000))
```

We can try this out in the REPL:

```
Prelude Week04.Contract> import Plutus.Trace.Emulator
Prelude Plutus.Trace.Emulator Week04.Contract> import Plutus.Contract.Trace
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Week04.Contract> defaultDist
fromList [(Wallet 1,Value (Map [(,Map [("",100000000)])])),(Wallet 2,Value (Map [(,Map [(
->"",100000000)])])),(Wallet 3,Value (Map [(,Map [("",100000000)])])),(Wallet 4,Value_
->(Map [(,Map [("",100000000)])])),(Wallet 5,Value (Map [(,Map [("",100000000)])])),
->(Wallet 6,Value (Map [(,Map [("",100000000)])])),(Wallet 7,Value (Map [(,Map [("",
->100000000)])])),(Wallet 8,Value (Map [(,Map [("",100000000)])])),(Wallet 9,Value (Map_
->[(,Map [("",100000000)])])),(Wallet 10,Value (Map [(,Map [("",100000000)])]))]
```

We can see that each of the 10 wallets has been given an initial distribution of 100,000,000 lovelace.

We can also get the balances for a specific wallet or wallets:

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Week04.Contract> defaultDistFor_
->[Wallet 1]
fromList [(Wallet 1,Value (Map [(,Map [("",100000000)])]))]
```

If you want different initial values, or if you want native tokens, then you have to specify that manually.

Let's see what we need to run our first trace:

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Week04.Contract> :t runEmulatorTrace
runEmulatorTrace
:: EmulatorConfig
-> EmulatorTrace ()
-> ([Wallet.Emulator.MultiAgent.EmulatorEvent], Maybe EmulatorErr,
    Wallet.Emulator.MultiAgent.EmulatorState)
```

So, we need an *EmulatorConfig* which we know takes an *InitialChainState*.

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Week04.Contract> import Wallet.
↳ Emulator.Stream
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator.Stream Week04.
↳ Contract> :i InitialChainState
type InitialChainState :: *
type InitialChainState =
Either InitialDistribution Ledger.Blockchain.Block
    -- Defined in 'Wallet.Emulator.Stream'
```

If we take the *Left* of the *defaultDist* will will get an *InitialDistribution*.

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator.Stream Week04.
↳ Contract> :t Left defaultDist
Left defaultDist :: Either InitialDistribution b
```

Which we can then use to construct an *EmulatorConfig*.

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator.Stream Week04.
↳ Contract> EmulatorConfig $ Left defaultDist
EmulatorConfig {_initialChainState = Left (fromList [(Wallet 1,Value (Map [(Map [("",
↳ 1000000000))])),(Wallet 2,Value (Map [(Map [("",1000000000))])),(Wallet 3,Value (Map_
↳ [(Map [("",1000000000))])),(Wallet 4,Value (Map [(Map [("",1000000000))])),(Wallet 5,
↳ Value (Map [(Map [("",1000000000))])),(Wallet 6,Value (Map [(Map [("",
↳ 1000000000))])),(Wallet 7,Value (Map [(Map [("",1000000000))])),(Wallet 8,Value (Map_
↳ [(Map [("",1000000000))])),(Wallet 9,Value (Map [(Map [("",1000000000))])),(Wallet_
↳ 10,Value (Map [(Map [("",1000000000))]))]))])}
```

So, let's try out *runEmulatorTrace*. Recall that, as well as and *EmulatorConfig*, we also need to pass in an *EmulatorTrace*, and the most simple one we can create is simply one that returns Unit - *return ()*.

```
runEmulatorTrace (EmulatorConfig $ Left defaultDist) $ return ()
```

If you run this in the REPL you will get a crazy amount of data output to the console, even though we are not doing anything with the trace. If you want to make it useful, you must somehow filter all this data down to something that sensible, and aggregate it in some way.

Luckily, there are other functions as well as *runEmulatorTrace*. One of them is *runEmulatorTraceIO* which runs the emulation then outputs the trace in a nice form on the screen.

```
runEmulatorTraceIO
:: EmulatorTrace ()
-> IO ()
runEmulatorTraceIO = runEmulatorTraceIO' def def
```

To use this function, we don't need to specify an *EmulatorConfig* like we did before, because by default will will just use the default distribution.

In the REPL:

```
Prelude...> runEmulatorTraceIO $ return ()
```

```
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
Slot 00000: SlotAdd Slot 1
Slot 00001: SlotAdd Slot 2
```

(continues on next page)

(continued from previous page)

```

Final balances
Wallet 1:
{, ""}: 1000000000
Wallet 2:
{, ""}: 1000000000
Wallet 3:
{, ""}: 1000000000
Wallet 4:
{, ""}: 1000000000
Wallet 5:
{, ""}: 1000000000
Wallet 6:
{, ""}: 1000000000
Wallet 7:
{, ""}: 1000000000
Wallet 8:
{, ""}: 1000000000
Wallet 9:
{, ""}: 1000000000
Wallet 10:
{, ""}: 1000000000

```

And we see a much more manageable, concise output. Nothing happens, but we see the Genesis transaction and then the final balances for each wallet.

If you want more control, there is also `runEmulatorTraceIO`, which does take an `EmulatorConfig`, so we could specify a different distribution.

```

runEmulatorTraceIO'
:: TraceConfig
-> EmulatorConfig
-> EmulatorTrace ()
-> IO ()
runEmulatorTraceIO' tcfg cfg trace
= runPrintEffect (outputHandle tcfg) $ runEmulatorTraceEff tcfg cfg trace

```

It also takes a `TraceConfig`, which has two fields.

```

data TraceConfig = TraceConfig
{ showEvent    :: EmulatorEvent' -> Maybe String
-- ^ Function to decide how to print the particular events.
, outputHandle :: Handle
-- ^ Where to print the outputs to. Default: 'System.IO.stdout'
}

```

The first field, `showEvent` is a function that specifies which `EmulatorEvent*s` are displayed and how they are displayed. It takes an `*EmulatorEvent` as an argument and can return `Nothing` if the event should not be displayed, or a `Just` with a `String` showing how the event will be displayed.

Here is the default `TraceConfig` used by `runEmulatorTraceIO`. We can see that most events are ignored and that we only get output for some of the events.

```

instance Default TraceConfig where
def = TraceConfig

```

(continues on next page)

(continued from previous page)

```

    { showEvent      = defaultShowEvent
    , outputHandle   = stdout
    }

defaultShowEvent :: EmulatorEvent' -> Maybe String
defaultShowEvent = \case
  UserThreadEvent (UserLog msg)                -> Just $ "**** USER_
  LOG: " <> msg
  InstanceEvent (ContractInstanceLog (ContractLog (A.String msg)) _ _) -> Just $ "****_
  CONTRACT LOG: " <> show msg
  InstanceEvent (ContractInstanceLog (StoppedWithError err) _ _) -> Just $ "****_
  CONTRACT STOPPED WITH ERROR: " <> show err
  InstanceEvent (ContractInstanceLog NoRequestsHandled _ _) -> Nothing
  InstanceEvent (ContractInstanceLog (HandledRequest _) _ _) -> Nothing
  InstanceEvent (ContractInstanceLog (CurrentRequests _) _ _) -> Nothing
  SchedulerEvent _ -> Nothing
  ChainIndexEvent _ _ -> Nothing
  WalletEvent _ _ -> Nothing
  ev -> Just ._
  renderString . layoutPretty defaultLayoutOptions . pretty $ ev

```

The second field is a handle which defaults to *stdout*, but we could also specify a file here.

Now let's look at a more interesting trace, using the *Vesting* contract from the last lecture.

First, we define a *Trace*.

```

myTrace :: EmulatorTrace ()
myTrace = do
  h1 <- activateContractWallet (Wallet 1) endpoints
  h2 <- activateContractWallet (Wallet 2) endpoints
  callEndpoint @"give" h1 $ GiveParams
    { gpBeneficiary = pubKeyHash $ walletPubKey $ Wallet 2
    , gpDeadline    = Slot 20
    , gpAmount      = 1000
    }
  void $ waitUntilSlot 20
  callEndpoint @"grab" h2 ()
  void $ waitNSlots 1

```

The first thing we have to do is to activate the wallets using the monadic function *activateContractWallet*. We bind the result of this function to *h1*, and then bind the result of a second call (for Wallet 2) to *h2*. Those two values - *h1* and *h2* are handles to their respective wallets.

Next, we use *callEndpoint* to simulate Wallet 1 calling the *give* endpoint, with the shown parameters. We then wait for 20 slots. The function *waitUntilSlot* actually returns a value representing the slot that was reached, but, as we are not interested in that value here, we use *void* to ignore it. We then simulate the call to the *grab* endpoint by Wallet 2.

Now, we can write a function to call *runEmulatorTraceIO* with out *Trace*.

```

test :: IO ()
test = runEmulatorTraceIO myTrace

```

And, we can then run this in the REPL:



```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator Week04.Trace Wallet.
↳ Emulator.Stream Week04.Contract> test
```

```
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
Slot 00000: SlotAdd Slot 1
Slot 00001: 000000000-0000-4000-8000-0000000000000 {Contract instance for wallet 1}:
  Contract instance started
Slot 00001: 000000000-0000-4000-8000-0000000000001 {Contract instance for wallet 2}:
  Contract instance started
Slot 00001: 000000000-0000-4000-8000-0000000000000 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "give"),("value",Object
↳ (fromList [("unEndpointValue",Object (fromList [("gpAmount",Number 1000.0),
↳ ("gpBeneficiary",Object (fromList [("getPubKeyHash",String
↳ "39f713d0a644253f04529421b9f51b9b08979d08295959c4f3990ee617f5139f"))]),("gpDeadline",
↳ Object (fromList [("getSlot",Number 20.0))]))))])))
Slot 00001: W1: TxSubmit:
↳ 49f326a21c09ba52eddee46b65bdb5fb33b3444745e9af1510a68f9043696eba
Slot 00001: TxnValidate 49f326a21c09ba52eddee46b65bdb5fb33b3444745e9af1510a68f9043696eba
Slot 00001: SlotAdd Slot 2
Slot 00002: *** CONTRACT LOG: "made a gift of 1000 lovelace to
↳ 39f713d0a644253f04529421b9f51b9b08979d08295959c4f3990ee617f5139f with deadline Slot
↳ {getSlot = 20}"
Slot 00002: SlotAdd Slot 3
Slot 00003: SlotAdd Slot 4
Slot 00004: SlotAdd Slot 5
Slot 00005: SlotAdd Slot 6
Slot 00006: SlotAdd Slot 7
Slot 00007: SlotAdd Slot 8
Slot 00008: SlotAdd Slot 9
Slot 00009: SlotAdd Slot 10
Slot 00010: SlotAdd Slot 11
Slot 00011: SlotAdd Slot 12
Slot 00012: SlotAdd Slot 13
Slot 00013: SlotAdd Slot 14
Slot 00014: SlotAdd Slot 15
Slot 00015: SlotAdd Slot 16
Slot 00016: SlotAdd Slot 17
Slot 00017: SlotAdd Slot 18
Slot 00018: SlotAdd Slot 19
Slot 00019: SlotAdd Slot 20
Slot 00020: 000000000-0000-4000-8000-0000000000001 {Contract instance for wallet 2}:
  Receive endpoint call: Object (fromList [("tag",String "grab"),("value",Object
↳ (fromList [("unEndpointValue",Array [])]))))]
Slot 00020: W2: TxSubmit:
↳ d9a2028384b4472242371f27cb51727f5c7c04327972e4278d1f69f606019a8b
Slot 00020: TxnValidate d9a2028384b4472242371f27cb51727f5c7c04327972e4278d1f69f606019a8b
Slot 00020: SlotAdd Slot 21
Slot 00021: *** CONTRACT LOG: "collected gifts"
Slot 00021: SlotAdd Slot 22
Final balances
Wallet 1:
  {, ""}: 99998990
```

(continues on next page)

(continued from previous page)

```

Wallet 2:
  {, ""}: 100000990
Wallet 3:
  {, ""}: 100000000
Wallet 4:
  {, ""}: 100000000
Wallet 5:
  {, ""}: 100000000
Wallet 6:
  {, ""}: 100000000
Wallet 7:
  {, ""}: 100000000
Wallet 8:
  {, ""}: 100000000
Wallet 9:
  {, ""}: 100000000
Wallet 10:
  {, ""}: 100000000

```

This output is very similar to the output we see in the playground. We can see the Genesis transaction as well as both the *give* and *grab* transactions from the *Trace*. We can also see some log output from the contract itself, prefixed with *CONTRACT LOG*.

We can also log from inside the *Trace* monad. We could, for example, log the result of the final *waitNSlots* call:

```

myTrace :: EmulatorTrace ()
myTrace = do
  ...
  ...
  s <- waitNSlots 1
  Extras.logInfo $ "reached slot " ++ show s

```

We would then see this output when we run the emulation:

```

...
Slot 00020: SlotAdd Slot 21
Slot 00021: *** USER LOG: reached slot Slot {getSlot = 21}
Slot 00021: *** CONTRACT LOG: "collected gifts"
Slot 00021: SlotAdd Slot 22
...

```

Now let's look at the Contract Monad.

### 4.3.2 The Contract Monad

The purpose of the Contract Monad is to define off-chain code that runs in the wallet. It has four type parameters:

```

newtype Contract w s e a = Contract { unContract :: Eff (ContractEffs w s e) a }
    deriving newtype (Functor, Applicative, Monad)

```

The *a* is the same as in every Monad - it denotes the result type of the computation.

We will go into the other three in more detail later but just briefly:

- *w* is like our *Writer* monad example, it allows us to write log messages of type *w*.
- *s* describes the blockchain capabilities, e.g. waiting for a slot, submitting transactions, getting the wallet's public key. It can also contain specific endpoints.
- *e* describes the type of error messages that this monad can throw.

Let's write an example.

```
myContract1 :: Contract () BlockchainActions Text ()
myContract1 = Contract.logInfo @String "Hello from the contract!"
```

Here, we pass a *Contract* constructed with *Unit* as the *w* type and *BlockchainActions* as the second argument, *s*. This gives us access to all the blockchain actions - the only thing we can't do is to call specific endpoints.

For *e* - the error message type, we use *Text*. *Text* is a Haskell type which is like *String*, but it is much more efficient.

We don't want a specific result, so we use *Unit* for the type *a*.

For the function body, we write a log message. We use *@String* because, we have imported the type *Data.Text* and we have used the *OverloadedStrings* GHC compiler option, so the compiler needs to know what type we are referencing - a *Text* or a *String*. We can use *@String* if we also use the compiler option *TypeApplications*.

Let's now define a *Trace* that starts the contract in the wallet, and a *test* function to run it.

```
myTrace1 :: EmulatorTrace ()
myTrace1 = void $ activateContractWallet (Wallet 1) myContract1

test1 :: IO ()
test1 = runEmulatorTraceIO myTrace1
```

If we run this in the REPL, we will see our log message from the contract.

Now, let's throw an exception.

```
myContract1 :: Contract () BlockchainActions Text ()
myContract1 = do
void $ Contract.throwError "BOOM!"
Contract.logInfo @String "Hello from the contract!"
```

Recall that we chose the type *Text* as the error message.

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator Week04.Trace Wallet.
↳Emulator.Stream Week04.Contract> test1
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
Slot 00000: SlotAdd Slot 1
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Contract instance started
Slot 00001: *** CONTRACT STOPPED WITH ERROR: "\"BOOM!\""
Slot 00001: SlotAdd Slot 2
Final balances
Wallet 1:
{, ""}: 1000000000
Wallet 2:
{, ""}: 1000000000
Wallet 3:
{, ""}: 1000000000
Wallet 4:
```

(continues on next page)

(continued from previous page)

```
{, ""}: 1000000000
Wallet 5:
{, ""}: 1000000000
Wallet 6:
{, ""}: 1000000000
Wallet 7:
{, ""}: 1000000000
Wallet 8:
{, ""}: 1000000000
Wallet 9:
{, ""}: 1000000000
Wallet 10:
{, ""}: 1000000000
```

Now, we don't get the log message, but we do get told that the contract stopped with an error and we see our exception message.

Another thing you can do is to handle exceptions. We will use the *handleError* function from module *Plutus.Contract.Types*.

```
handleError ::
  forall w s e e' a.
  (e -> Contract w s e' a)
-> Contract w s e a
-> Contract w s e' a
handleError f (Contract c) = Contract c' where
  c' = E.handleError @e (raiseUnderN @[E.Error e'] c) (fmap unContract f)
```

The *handleError* function takes an error handler and a *Contract* instance. The error handler takes an argument of type *e* from our contract, and returns a new *Contract* with the same type parameters as the first, but we can change the type of the *e* argument - the error type, which is expressed in the return *Contract* argument list as *e'*.

```
myContract2 :: Contract () BlockchainActions Void ()
myContract2 = Contract.handleError
  (\err -> Contract.logError $ "Caught error: " ++ unpack err)
  myContract1

myTrace2 :: EmulatorTrace ()
myTrace2 = void $ activateContractWallet (Wallet 1) myContract2

test2 :: IO ()
test2 = runEmulatorTraceIO myTrace2
```

We use the type *Void* as the error type. *Void* is a type that can hold no value, so, by using this type we are saying that there cannot be any errors for this contract.

**Note:** The function *unpack* is defined in the *Data.Text* module. It converts a value of type *Text* to a value of type *String*.

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator Week04.Trace Wallet.
  ↳Emulator.Stream Week04.Contract> test2
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
```

(continues on next page)

(continued from previous page)

```

Slot 00000: SlotAdd Slot 1
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Contract instance started
Slot 00001: *** CONTRACT LOG: "Caught error: BOOM!"
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Contract instance stopped (no errors)
Slot 00001: SlotAdd Slot 2
Final balances
...

```

We no longer get the error message, but, instead we get a message from the error handler showing the exception that was thrown by `Contract1`. Note that we still do not get the message “Hello from the contract!”. Contract 1 still stopped processing after its error, but there was no overall contract error due to the exception being caught and handled.

Of course, exceptions can also happen even if they are not explicitly thrown by your contract code. There are operations, such as submitting a transaction where there are insufficient inputs to make a payment for an output, where Plutus will throw an exception.

Next, let’s look at the `s` parameter, the second parameter to `Contract`, that determines the available blockchain actions.

In the first two examples we just used the `BlockChainActions` type which has all the standard functionality but without support for specific endpoints. If we want support for specific endpoints, we must use a different type.

The way that is usually done is by using a type synonym. The following example will create a type synonym `MySchema` that has all the capabilities of `BlockChainActions` but with the addition of being able to call endpoint `foo` with an argument of type `Int`.

```
type MySchema = BlockchainActions .\ Endpoint "foo" Int
```

**Note:** The operator `.\` is a type operator - it operates on types, not values. In order to use this we need to use the `TypeOperators` and `DataKinds` compiler options.

Now, we can use the `MySchema` type to define our contract.

```

myContract3 :: Contract () MySchema Text ()
myContract3 = do
  n <- endpoint @"foo"
  Contract.logInfo n

```

This contract will block until the endpoint `foo` is called with, in our case, an `Int`. Then the value of the `Int` parameter will be bound to `n`. Because of this, it is no longer enough for us to just activate the contract to test it. Now, we must invoke the endpoint as well.

In order to do this, we now need to handle from `activateContractWallet`, which we can then use to call the endpoint.

```

myTrace3 :: EmulatorTrace ()
myTrace3 = do
  h <- activateContractWallet (Wallet 1) myContract3
  callEndpoint @"foo" h 42

test3 :: IO ()
test3 = runEmulatorTraceIO myTrace3

```

Running this in the REPL:

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator Week04.Trace Wallet.
↳Emulator.Stream Week04.Contract> test3
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
...
Receive endpoint call: Object (fromList [("tag",String "foo"),("value",Object (fromList_
↳[("unEndpointValue",Number 42.0)]))])
Slot 00001: 000000000-0000-4000-8000-0000000000000 {Contract instance for wallet 1}:
Contract log: Number 42.0
...
Final balances
...
Wallet 10:
{, ""}: 1000000000
```

Finally, let's look at the first type parameter, the writer. The  $w$  cannot be an arbitrary type, it must be an instance of the type class *Monoid*.

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator Week04.Trace Wallet.
↳Emulator.Stream Week04.Contract> :i Monoid
type Monoid :: * -> Constraint
class Semigroup a => Monoid a where
empty :: a
mappend :: a -> a -> a
mconcat :: [a] -> a
{-# MINIMAL empty #-}
-- Defined in 'GHC.Base'
instance Monoid [a] -- Defined in 'GHC.Base'
instance Monoid Ordering -- Defined in 'GHC.Base'
instance Semigroup a => Monoid (Maybe a) -- Defined in 'GHC.Base'
instance Monoid a => Monoid (IO a) -- Defined in 'GHC.Base'
instance Monoid b => Monoid (a -> b) -- Defined in 'GHC.Base'
instance (Monoid a, Monoid b, Monoid c, Monoid d, Monoid e) =>
Monoid (a, b, c, d, e)
-- Defined in 'GHC.Base'
instance (Monoid a, Monoid b, Monoid c, Monoid d) =>
Monoid (a, b, c, d)
-- Defined in 'GHC.Base'
instance (Monoid a, Monoid b, Monoid c) => Monoid (a, b, c)
-- Defined in 'GHC.Base'
instance (Monoid a, Monoid b) => Monoid (a, b)
-- Defined in 'GHC.Base'
instance Monoid () -- Defined in 'GHC.Base'
```

This is a very important and very common type class in Haskell. It defines *empty* and *mappend*.

The function *empty* is like the neutral element, and *mappend* combines two elements of this type to create a new element of the same type.

The prime example of a *Monoid* is *List*, when *empty* is the empty list `[]`, and *mappend* is concatenation `++`.

For example:

```
Prelude> empty :: [Int]
[]
```

(continues on next page)

(continued from previous page)

```
Prelude> mappend [1, 2, 3 :: Int] [4, 5, 6]
[1,2,3,4,5,6]
```

There are many, many other examples of the *Monoid* type, and we will see other instances in this course.

But for now, let's stick with lists and write our last example.

```
myContract4 :: Contract [Int] BlockchainActions Text ()
myContract4 = do
  void $ Contract.waitNSlots 10
  tell [1]
  void $ Contract.waitNSlots 10
  tell [2]
  void $ Contract.waitNSlots 10
```

Rather than using *Unit* as our *w* type, we are using *[Int]*. This allows us to use the *tell* function as shown.

This now gives us access to those messages during the trace, using the *observableState* function.

```
myTrace4 :: EmulatorTrace ()
myTrace4 = do
  h <- activateContractWallet (Wallet 1) myContract4

  void $ Emulator.waitNSlots 5
  xs <- observableState h
  Extras.logInfo $ show xs

  void $ Emulator.waitNSlots 10
  ys <- observableState h
  Extras.logInfo $ show ys

  void $ Emulator.waitNSlots 10
  zs <- observableState h
  Extras.logInfo $ show zs

test4 :: IO ()
test4 = runEmulatorTraceIO myTrace4
```

If we run this in the REPL, we can see the *USER LOG* messages created using the *tell* function.

```
Prelude Plutus.Trace.Emulator Plutus.Contract.Trace Wallet.Emulator Week04.Trace Wallet.
↳Emulator.Stream Week04.Contract> test4
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
Slot 00000: SlotAdd Slot 1
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Contract instance started
Slot 00001: SlotAdd Slot 2
...
Slot 00005: SlotAdd Slot 6
Slot 00006: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Sending contract state to Thread 0
Slot 00006: SlotAdd Slot 7
Slot 00007: *** USER LOG: []
Slot 00007: SlotAdd Slot 8
```

(continues on next page)

(continued from previous page)

```
...
Slot 00015: SlotAdd Slot 16
Slot 00016: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Sending contract state to Thread 0
Slot 00016: SlotAdd Slot 17
Slot 00017: *** USER LOG: [1]
Slot 00017: SlotAdd Slot 18
...
Slot 00025: SlotAdd Slot 26
Slot 00026: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Sending contract state to Thread 0
Slot 00026: SlotAdd Slot 27
Slot 00027: *** USER LOG: [1,2]
Final balances
Wallet 1:
  {, ""}: 1000000000
Wallet 2:
  {, ""}: 1000000000
...
Wallet 10:
  {, ""}: 1000000000
```

Using this mechanism, it is possible to pass information from the contract running in the wallet to the outside world. Using endpoints we can pass information into a contract. And using the *tell* mechanism we can get information out of the wallet.



## WEEK 05 - NATIVE TOKENS

---

**Note:** These is a written version of [Lecture #5](#).

In this lecture we learn about native tokens, minting policies and NFTs.

These notes use Plutus commit 0c3c310cab61dbff8cbc1998a3678b367be6815a

---

### 5.1 Overview

We are going to talk about how Plutus supports native tokens and how to define under which conditions native tokens can be minted and burned. But before we get to that, let's explore what *value* means in Cardano.

When we talked about the (E)UTxO model, we learned that each UTxO (unspent transaction) has an address and a value. And, we saw that, as a result of being extended to the (E)UTxO model, each UTxO also has a *Datum*. We have seen examples of such UTxOs in previous lectures.

In almost all the examples we have seen so far, the value was simply an Ada value, denominated in lovelace. The exception was the first example, from lecture 1, namely the *English Auction* example. In that example we auctioned away an NFT. However, the NFT was just created out of thin air in the playground.

In the real Cardano blockchain, however, in the beginning there are only Ada, there are no other native tokens. So, you have to do something to create new native tokens, or to burn existing ones. In this lecture we will see how to do that.

But let's first talk about values.

### 5.2 Value

The relevant types are defined in package *plutus-ledger-api*. The modules of interest are

```
module Plutus.V1.Ledger.Value
module Plutus.V1.Ledger.Ada
```

### 5.2.1 The Value Type

*Value* is defined as a map from *CurrencySymbol\*s* to maps from *\*TokenName\*s* to *\*Integers*, which sounds a bit weird and complicated.

```
newtype Value = Value { getValue :: Map.Map CurrencySymbol (Map.Map TokenName Integer) }
  deriving stock (Generic)
  deriving anyclass (ToJSON, FromJSON, Hashable, NFData)
  deriving newtype (Serialise, PlutusTx.IsData)
  deriving Pretty via (PrettyShow Value)
```

The first thing to note is that each native token, including Ada, is identified by two pieces of data - the *CurrencySymbol* and the *TokenName*.

A *CurrencySymbol* is a *newtype* wrapper around a *ByteString*.

```
newtype CurrencySymbol = CurrencySymbol { unCurrencySymbol :: Builtins.ByteString }
  deriving (IsString, Show, Serialise, Pretty) via LedgerBytes
  deriving stock (Generic)
  deriving newtype (Haskell.Eq, Haskell.Ord, Eq, Ord, PlutusTx.IsData)
  deriving anyclass (Hashable, ToJSONKey, FromJSONKey, NFData)
```

And the same is true for *TokenName*.

```
newtype TokenName = TokenName { unTokenName :: Builtins.ByteString }
  deriving (Serialise) via LedgerBytes
  deriving stock (Generic)
  deriving newtype (Haskell.Eq, Haskell.Ord, Eq, Ord, PlutusTx.IsData)
  deriving anyclass (Hashable, NFData)
  deriving Pretty via (PrettyShow TokenName)
```

So we have these two *ByteStrings* that define a coin, or, as it is also called, an *asset class*.

```
assetClass :: CurrencySymbol -> TokenName -> AssetClass
assetClass s t = AssetClass (s, t)
```

Ada is one asset class, and custom native tokens will be other asset classes.

A *Value* simply shows how many units exist for a given asset class.

Let's start the REPL and import the two relevant modules.

```
cabal repl
Prelude Week05.Free> import Plutus.V1.Ledger.Ada
Prelude Plutus.V1.Ledger.Ada Week05.Free> import Plutus.V1.Ledger.Value
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free>
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free> :set -XOverloadedStrings
```

---

**Note:** We have also activated the *OverloadedStrings* extension so that we can enter *\*ByteString\*s* as literal strings.

---

Now let's look at some values. Let's start with lovelace values. In the *Ledger.Ada* module there is a function called *adaSymbol*.

```
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free> :t adaSymbol
adaSymbol :: CurrencySymbol
```

This gives us the currency symbol of the Ada asset class, which is just the empty *ByteString*. Similarly, there is a function *adaToken*, which will give us the token name.

```
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free> :t adaToken
adaToken :: TokenName
```

Again, this is also the empty *ByteString*.

We have seen before in the examples how to construct a *Value* containing just lovelace. There is a function *lovelaceValueOf* that, given an *Integer*, gives us a *Value*.

```
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free> :t lovelaceValueOf
lovelaceValueOf :: Integer -> Value
```

So, for example to have 123 lovelace, we can do:

```
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free> lovelaceValueOf 123
Value (Map [(Map [("",123)])])
```

You will always use a helper function such as *lovelaceValueOf* to construct the value maps - you would never need to construct one directly.

Here we see the map. The out map of currency symbols has one key, which is the empty symbol for Ada, and the inner map of token names has one key, the empty string for Ada, and a value of 123.

One thing we can do with values is combine them. The *Value* class is an instance of *Monoid*, so we can use *mappend*, which we can write as *<>*, which comes from a super class of *Monoid* called *Semigroup*.

```
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free> lovelaceValueOf 123 <>
↳lovelaceValueOf 10
Value (Map [(Map [("",133)])])
```

So, how do we create *\*Value\*s* containing native tokens?

There is a very useful function called *singleton*.

```
Prelude Plutus.V1.Ledger.Ada Plutus.V1.Ledger.Value Week05.Free> :t singleton
singleton :: CurrencySymbol -> TokenName -> Integer -> Value
```

This will create a *Value* for a token specified by the *CurrencySymbol* and the *TokenName*, and for a given *Integer* amount.

```
Week05.Free> singleton "a8ff" "ABC" 7
Value (Map [(a8ff,Map [("ABC",7)])])
```

The first argument, “a8ff” for *CurrencySymbol*” has to be a string representing a hexadecimal value, for reasons that will soon become clear. The second argument, “ABC” for *\*TokenName* can be an arbitrary string.

And, we can combine, as before, with the *mappend* operator. We can now create a somewhat more interesting map.

```
Week05.Free> singleton "a8ff" "ABC" 7 <> lovelaceValueOf 42 <> singleton "a8ff" "XYZ" 100
Value (Map [(Map [("",42)]),(a8ff,Map [("ABC",7),("XYZ",100)])])
```

Now, we see a map representing 42 lovelace as well as two tokens *ABC* and *XYZ* both belonging to the *CurrencySymbol* “a8ff”, and each with their respective integer amounts.

Let’s give this value a name:

```
Week05.Free> let v = singleton "a8ff" "ABC" 7 <> lovelaceValueOf 42 <> singleton "a8ff"
↳ "XYZ" 100
Week05.Free> v
Value (Map [(Map [("",42)]),(a8ff,Map [("ABC",7),("XYZ",100)])])
```

Another useful function is *valueOf* which allows us to get the value of a given currency symbol and token name.

```
Week05.Free> :t valueOf
valueOf :: Value -> CurrencySymbol -> TokenName -> Integer

Week05.Free> valueOf v "a8ff" "XYZ"
100

Week05.Free> valueOf v "a8ff" "ABC"
7

Week05.Free> valueOf v "a8ff" "abc"
0
```

Another useful function is *flattenValue*. As the name suggests, it flattens the map of maps into a flat list of triples.

```
Week05.Free> :t flattenValue
flattenValue :: Value -> [(CurrencySymbol, TokenName, Integer)]

Week05.Free> flattenValue v
[(a8ff,"ABC",7),(a8ff,"XYZ",100),("",42)]
```

## 5.3 Minting Policies

Now the question is why? Why do we need both a currency symbol and a token name? Why don't we just use one identifier for an asset class? And why does the currency symbol have to be in hexadecimal digits?

This is where so-called minting policies come in.

The rule is that, in general, a transaction can't create or delete tokens. Everything that goes in also comes out, with the exception of the fees. There is always a lovelace fee that has to be paid with each transaction. The fee depends on the size of the transaction and the number of steps that the validation script takes to execute, and the memory consumption of the script.

But, if that was the whole story then we could never create native tokens. And this is where minting policies come in, and the relevance of the currency symbol comes in.

The reason that the currency symbol has to consist of hexadecimal digits is that it is actually the hash of a script. And this script is called the minting policy, and if we have a transaction where we want to create native or burn native tokens then, for each native token that we try to create or burn, the currency symbol is looked up. So, the corresponding script must also be contained in the transaction. And that script is executed along with the other validation scripts.

And, similar to the validation scripts that we have seen so far to validate input, the purpose of these minting scripts is to decide whether this transaction has the right to mint or burn tokens. Ada also fits into this scheme. Remember the currency symbol of Ada is just an empty string, which is not the hash of any scripts. So there is no script that hashes to the empty string, so there is no script that would allow the minting or burning of Ada, which means that Ada can never be minted or burned.

All the Ada that exists comes from the Genesis transaction and the total amount of Ada in the system is fixed and can never change. Only custom native tokens can have custom minting policies.

So we'll look at an example of a minting policy next and will see that it is very similar to a validation script, but not identical.

Before we write out first minting policy, let's briefly recall how validation works.

When we don't have a public key address, but a script address, and a UTxO that sits at that address, then for any transaction that tries to consume that UTxO, a validation script is run.

That validation script gets, as input, the datum, which comes from the UTxO, the redeemer, which comes from the input, and the context.

Recall that the *ScriptContext* has two fields.

```
data ScriptContext = ScriptContext{scriptContextTxInfo :: TxInfo, scriptContextPurpose_
↳ :: ScriptPurpose }
```

One of those fields is *ScriptPurpose*, and, for this field, everything we have seen until now has been of type *Spending*.

```
data ScriptPurpose
  = Minting CurrencySymbol
  | Spending TxOutRef
  | Rewarding StakingCredential
  | Certifying DCert
```

The other field is of type *TxInfo* which contains all the context information about the transaction.

```
-- | A pending transaction. This is the view as seen by validator scripts, so some_
↳ details are stripped out.
data TxInfo = TxInfo
  { txInfoInputs      :: [TxInInfo] -- ^ Transaction inputs
  , txInfoInputsFees  :: [TxInInfo] -- ^ Transaction inputs designated to pay fees
  , txInfoOutputs     :: [TxOut]   -- ^ Transaction outputs
  , txInfoFee         :: Value     -- ^ The fee paid by this transaction.
  , txInfoForge       :: Value     -- ^ The 'Value' forged by this transaction.
  , txInfoDCert       :: [DCert]   -- ^ Digests of certificates included in this_
↳ transaction
  , txInfoWdrl        :: [(StakingCredential, Integer)] -- ^ Withdrawals
  , txInfoValidRange  :: SlotRange -- ^ The valid range for the transaction.
  , txInfoSignatories :: [PubKeyHash] -- ^ Signatures provided with the transaction,_
↳ attested that they all signed the tx
  , txInfoData        :: [(DatumHash, Datum)]
  , txInfoId          :: TxId
  -- ^ Hash of the pending transaction (excluding witnesses)
  } deriving (Generic)
```

For minting policies, this is triggered if the *txInfoForge* field of the transaction contains a non-zero value. In all of the transactions we have seen so far, this field value has been zero - we have never created or destroyed any tokens.

If it is non-zero, then for each currency symbol contained in the *Value*, the corresponding minting policy script is run.

Whereas the validation scripts had three inputs - the datum, the redeemer and the context, these minting policy scripts only have one input - the context. And it is the same context as we had before - the *ScriptContext*. It would make no sense to have the datum, as it belongs to the UTxO, and it would make no sense to have the redeemer as it belongs to the validation script. The minting policy belongs to the transaction itself, not to a specific input or output.

As for the *ScriptPurpose*, this will not be *Spending* as it has been until now, but will be *Minting*.

## 5.4 Example 1 - Free

Let's write a simple minting policy.

### 5.4.1 On chain

When we wrote a validator we had a function such as the following:

```
mkValidator :: Datum -> Redeemer -> ScriptContext -> Bool
```

We also saw the low-level version where we had three *Data* arguments and returned *Unit*. And we saw that there can be additional arguments before the datum, if we write a parameterized script.

We can also have parameterized minting policy scripts and we will see that in a later example. But first we will look at one that is not parameterized.

First, let's rename the function to *mkPolicy*, remove the datum and redeemer, and write the simplest minting policy that we can.

```
mkPolicy :: ScriptContext -> Bool
mkPolicy _ = True
```

This policy ignores the context and always returns *True*. This will allow arbitrary minting and burning of tokens for and token name that belongs to the currency symbol associated with this policy.

Remember that, when we were writing a validator, we needed to use Template Haskell to compile this function to Plutus code. We need to do something similar for our minting policy.

```
policy :: Scripts.MonetaryPolicy
policy = mkMonetaryPolicyScript $(PlutusTx.compile [| | Scripts.wrapMonetaryPolicy_
↳mkPolicy | |])
```

And, as before, we need to make the *mkPolicy* function *INLINABLE*, as everything within the Oxford brackets needs to be available at compile time.

```
{-# INLINABLE mkPolicy #-}
mkPolicy :: ScriptContext -> Bool
mkPolicy _ = True
```

Now that we have a policy, we can get a currency symbol from the policy.

```
curSymbol :: CurrencySymbol
curSymbol = scriptCurrencySymbol policy
```

And, we can look at this in the REPL:

```
Prelude Week05.Free> curSymbol
e01824b4319351c40b5ec727fff328a82076b1474a6bad6c8e8a2cd835cc6aaf
```

And this completes the on-chain part, for this simple minting policy. But in order to try it out and interact with it, we need an off-chain part.

### 5.4.2 Off chain

What should the off-chain part do? Well, it should allow arbitrary wallets to mint and burn tokens of this currency symbol.

We have the currency symbol, so what is missing is the token name and the amount we want to mint or burn. And for this, we will define a data type *MintParams*.

```
data MintParams = MintParams
  { mpTokenName :: !TokenName
  , mpAmount    :: !Integer
  } deriving (Generic, ToJSON, FromJSON, ToSchema)
```

We see two fields - *mpTokenName* and *mpAmount*. The idea is that if the *mpAmount* is positive, we should create tokens, and if it is negative, we should burn tokens.

The next step is to define the schema. Recall that one of the parameters of the *Contact* monad was the schema that defined the available actions that we can take.

```
type FreeSchema =
  BlockchainActions
    .\ Endpoint "mint" MintParams
```

As always, we have *BlockchainActions* that give us access generic things like getting your own public key. And here, we have added an endpoint *mint* using the type-level operator we have seen previously.

So, now we can look at the contract itself.

```
mint :: MintParams -> Contract w FreeSchema Text ()
```

In the past, we have not gone into detail with the off-chain part of the contract. But, as we now know about the *Contract* monad from the last lecture, we are ready to go into it in much more detail.

Recall that the *Contract* monad takes four type parameters.

The first is the writer monad which allows us to use a *tell* function. By leaving this parametric with a small *w*, we indicate that we will not be making use of this parameter - we won't *tell* any state.

The next parameter is the schema that we just discussed. As noted above, by using *FreeSchema* we have access to the regular block chain actions, as well as the *mint* endpoint.

The third parameter is the type of error message, and as we have seen, *Text* is usually a good choice.

Finally the last parameter is the return type, and our contract will just have the Unit return type.

Now the function body. As *Contact* is a monad, we can use *do* notation.

```
mint mp = do
  let val      = Value.singleton curSymbol (mpTokenName mp) (mpAmount mp)
      lookups  = Constraints.monetaryPolicy policy
      tx       = Constraints.mustForgeValue val
  ledgerTx <- submitTxConstraintsWith @Void lookups tx
  void $ awaitTxConfirmed $ txId ledgerTx
  Contract.logInfo @String $ printf "forged %s" (show val)
```

The first thing that we define is the value that we want to forge. For this we are using the *singleton* function that we tried out in the REPL earlier.

The arguments to the *singleton* function are the currency symbol that represents the hash of the minting policy, plus the token name and amount extracted from the *MintParams*.

We'll skip the *lookups* assignment for the moment, and move onto the *tx* assignment.

One of the main purposes of the *Contract* monad is to construct and submit transactions. The path that the Plutus team has taken to do that is provide a way to specify the constraints of the transaction you are defining. The Plutus libraries then take care of constructing the correct transaction (if possible). This is as opposed to being required to specify all the inputs and outputs manually, which would be tedious as many requirements, such as sending change back to the sending wallet, are often the same.

These conditions all have names that start with *must*. There are things like *mustSpendScriptOutput*, *mustPayToPublicKey* and all sorts of conditions that can be put on a condition.

In our example, we are using *mustForgeValue* and we pass it the previously-defined *val*. The result of forging the tokens specified by *val* is that they will end up in our own wallet.

Once the conditions are defined, you then need to call a function to submit the transaction. There are a variety of such functions, but in this case, the appropriate one is *submitTxConstraintsWith*.

These *submitTx* functions all take these declarative conditions that the transaction must satisfy, and then they try to construct a transaction that fulfils those conditions. In our case, the only condition is that we want to forge the value.

So what must the *submitTxConstraintsWith* do in order to create a valid transaction? It must, for example balance the inputs and outputs. In this case, because we always have transaction fees, we need an input that covers the transactions fees. So, to create the transaction, the function will look at our own UTXOs and find one, or more, that can cover the transaction fees, and use them as an input to the transaction.

Furthermore, if we are forging value (if *mpAmount* is positive), that must go somewhere. In this case, *submitTxConstraintsWith*, will create an output that sends the newly-minted value to our own wallet.

If, on the other hand, we were burning tokens (if *mpAmount* is negative), then those tokens must come from somewhere. In that case, the *submitTxConstraintsWith* function would find an input in our own wallet from which to take the tokens.

The submit function can also fail. For example, if we want to pay someone, but we do not have enough funds in our wallet, it would fail. Or, if we are asking to burn tokens that we don't have, it will also fail. On failure, an exception would be thrown, with an error message of type *Text*.

Now, back to the *lookups*. In order to fulfil the conditions in the *mustForgeValue* function, and to construct the transaction, sometimes the library needs additional information. In this case, in order to validate a transaction that forges value, the nodes that validate the transaction have to run the policy script.

But, the currency symbol is only the hash of the policy script. In order to run the script itself, it must be included in the transaction. Which means that, in the construction step of the transaction, when the algorithm see the *mustForgeValue* constraint, it knows it has to attach the corresponding policy script to the transaction.

In order to tell the algorithm where the policy script is, we can give it hints, and these are the lookups. There are a variety of lookups that can be used - you can give UTXOs, validator scripts, and, as we do here, you can give monetary policy scripts.

In our case, the only thing we need to supply as a lookup is the policy that we defined earlier in the script.

There are variants of *submitTxConstraintsWith* without the *with* that do not take lookups, as we have seen in previous lectures.

Finally, the *@Void* on the line:

```
ledgerTx <- submitTxConstraintsWith @Void lookups tx
```

Most of the constraint functions are geared towards using a specific validator script. Normally you have the situation that you are working on one specific smart contract. And that specific smart contract has a datum and a redeemer type, and most of the constraints functions are parametric in the datum and redeemer type. In that case you can directly use the datum type without first having to convert it to the Plutus *Datum* type.



But in this case, we are not making use of that. We don't have any validator script. Which means that *submitTxConstraintsWith* wouldn't know which type to use for datum and redeemer because we don't have them in this example. So, in that case we must tell the compiler which type to use. We don't care, as there is no datum and redeemer, so we use the *Void* type.

Also, in the same line, we see a monadic bind, so we know that this is a monadic action happening within the *Contract* monad. The reason for this is that, in order to lookup, for example, our UTxOs, the *submitTxConstraintsWith* function must make use of the super power of the *Contract* monad, which is to access the *BlockchainActions*.

Now, *ledgerTx* is basically a handle to the transaction to we just submitted.

Then we wait for the transaction to be confirmed.

```
void $ awaitTxConfirmed $ txId ledgerTx
```

Currently, if the transaction validation fails, the await for confirmation line will block forever. However, this will soon change in an upcoming Plutus release to allow us to listen for status changes, so you could detect if validation failed.

Once confirmed, we simply write a log message.

Finally, we need some more boilerplate to define our endpoint, to be able to actually execute the *mint* function, for example, in the playground.

```
endpoints :: Contract () FreeSchema Text ()
endpoints = mint' >> endpoints
where
  mint' = endpoint @"mint" >=> mint
```

We define another contract, *endpoints*, and that is always the name of the contract that the playground will run. So, if you want to test something in the playground, you always need something called *endpoints*.

Here we just define a function called *mint'* and then recursively call *endpoints*, so once it has executed, it will be available to be executed again.

For *mint'* we must somehow get the *MintParams* and for that we use the *endpoint* function. The *endpoint* function blocks until someone provides a parameter. Once the parameter of *MintParams* is provided, we use the monadic bind to call the *mint* function with those arguments.

The final two lines, as we have seen before, are just needed for the playground UI.

```
mkSchemaDefinitions ''FreeSchema
mkKnownCurrencies []
```

### 5.4.3 In The Playground

We have set up a scenario where Wallet 1 mints 555 ABC tokens, and Wallet 2 mints 444 ABC tokens. Then, after waiting for 1 slot, Wallet 1 burns 222 ABC tokens. Finally, we wait for 1 slot at the end.

Now, if we evaluate this, first we see the genesis transaction where the wallets are given 1000 lovelace each.

Next, we see two transactions at Slot 1. The first is the transaction from Wallet 2, where 444 ABC tokens are minted, and a 10 lovelace fee is paid. The UTxO to pay the fees was automatically found by the function that created the transaction *submitTxConstraintsWith*, as discussed previously.

We see something here that we have not seen before - the *Forge* part of a transaction, where the native tokens are actually created. The box contains the currency symbol (the policy hash) and the token name.

We also see the two outputs - once with the 990 lovelace change, and another with the newly-minted tokens. These outputs could, in fact, be combined, but here they are shown as two separate UTxOs.

Simulation 1 +

## Wallets

Add some initial wallets, then click one of your function calls inside the wallet to begin a chain of actions.

**Wallet 1** ×  
 Opening Balances  
 Lovelace 1000  
 Available functions  
 mint +  
 Pay to Wallet +

**Wallet 2** ×  
 Opening Balances  
 Lovelace 1000  
 Available functions  
 mint +  
 Pay to Wallet +

+  
Add Wallet

## Actions

This is your action sequence. Click "Evaluate" to run these actions against a simulated blockchain.

**1** **Wallet 1: mint** ×  
 mpTokenName  
 unTokenName  
 ABC ✓  
 mpAmount  
 555 ✓

**2** **Wallet 2: mint** ×  
 mpTokenName  
 unTokenName  
 ABC ✓  
 mpAmount  
 444 ✓

**3** **Wait** ×  
 Wait For... Wait Until...  
 Blocks 1

**4** **Wallet 1: mint** ×  
 mpTokenName  
 unTokenName  
 ABC ✓  
 mpAmount  
 -222 ✓

**5** **Wait** ×  
 Wait For... Wait Until...  
 Blocks 1

+  
Add Wait Action

Evaluate Transactions

Simulation 1 +

## Transactions

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 1, Tx 1

### Inputs

### Transaction

Slot 0, Tx 0

Tx: 23fdb80eaf4a806e5cc4026092c162c5d7a106aef41d87f3384c67be4857

Validity: All time  
Signatures: None

Forge

Ada Lovelace 2,000

### Outputs

**Wallet 2**  
 PubKeyHash 39f713d0a644253f04529421b9f51b9b08979d...  
 Ada Lovelace 1,000  
 Spent in: Slot 1, Tx 0

**Wallet 1**  
 PubKeyHash 21fe31dfa154a261626b854046f2271b7be4b...  
 Ada Lovelace 1,000  
 Spent in: Slot 1, Tx 1

Simulation 1 +

## Transactions

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 1, Tx 1

### Inputs

**Wallet 2**  
 PubKeyHash 39f713d0a644253f04529421b9f51b9b08979d...  
 Ada Lovelace 1,000  
 Created by: Slot 0, Tx 0

### Transaction

Slot 1, Tx 0

Tx: 120b10471e7a7935cc92b74be9d10608270ee508e06d52f328ac1ab8c26c1a5

Validity: All time  
Signatures:  
• PubKey 3d4017c3e843895a92b70aa74d1b7ebc9c982cf2ec4968cc0d55f12a4660c

Forge

e01824b4319351c40b5ec727ff328a82076b1474a6bad6c8e8a2cd835cc6aaf  
ABC 444

### Outputs

**Fee**  
 Ada Lovelace 10

**Wallet 2**  
 PubKeyHash 39f713d0a644253f04529421b9f51b9b08979d...  
 Ada Lovelace 990  
 e01824b4319351c40b5ec727ff328a82076b1474a6bad6c8e8a2c...  
 Unspent

**Wallet 2**  
 PubKeyHash 39f713d0a644253f04529421b9f51b9b08979d...  
 Ada Lovelace 444  
 ABC  
 Unspent

Then, we see the transaction from Wallet 1, where 555 ABC tokens are minted, and a 10 lovelace fee is paid.

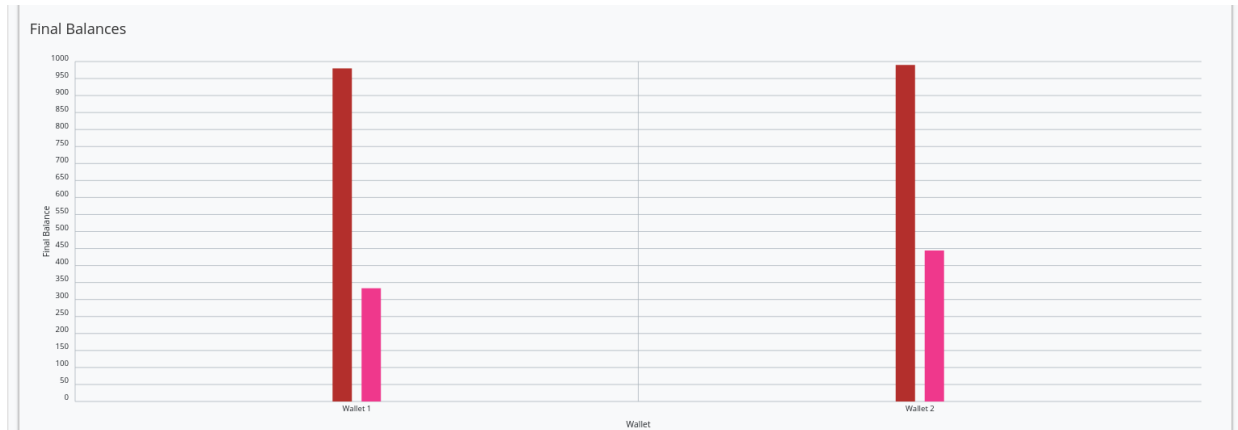
The screenshot displays the 'Transactions' window in the Plutus Pioneer Program simulation. It shows a transaction from Slot 1, Tx 1. The transaction details include a Tx hash, validity, and a signature. The inputs section shows a wallet with 1,000 Lovelace and a forged ABC token. The outputs section shows a fee of 10 Lovelace and a wallet with 990 Lovelace and 555 ABC tokens.

Finally, we see the burning of 222 tokens by Wallet 1. Here we see that the algorithm did something slightly different. When it notices that a burn is taking place, it has found the ABC tokens UTxO in Wallet 1 and used them as an input. We also note here that the output UTxO is combined, which, as we mentioned above, can be done instead of using two output UTxOs.

The screenshot displays the 'Transactions' window in the Plutus Pioneer Program simulation. It shows a transaction from Slot 2, Tx 0. The transaction details include a Tx hash, validity, and a signature. The inputs section shows a wallet with 990 Lovelace and a forged ABC token. The outputs section shows a fee of 10 Lovelace and a wallet with 980 Lovelace and 333 ABC tokens.

And we can also view the final balances to double check that all went according to plan.

With our monetary policy, we can create arbitrary forging and burning transactions by any wallet. So, this is probably not a very good monetary policy. The purpose of a token is to represent value, but if anybody at any time can mint new tokens, this token will not make much sense. There might be some exotic use case for it, but realistically this policy is rather useless.



### 5.4.4 Testing with EmulatorTrace

Let's also test this from the command line, rather than in the playground.

```
test :: IO ()
test = runEmulatorTraceIO $ do
  let tn = "ABC"
  h1 <- activateContractWallet (Wallet 1) endpoints
  h2 <- activateContractWallet (Wallet 2) endpoints
  callEndpoint @"mint" h1 $ MintParams
    { mpTokenName = tn
    , mpAmount    = 555
    }
  callEndpoint @"mint" h2 $ MintParams
    { mpTokenName = tn
    , mpAmount    = 444
    }
  void $ Emulator.waitNSlots 1
  callEndpoint @"mint" h1 $ MintParams
    { mpTokenName = tn
    , mpAmount    = -222
    }
  void $ Emulator.waitNSlots 1
```

If we run this in the REPL, we see what we saw in the playground, but instead on the console. It's not as pretty, but it is quicker.

```
Prelude Week05.Free> test
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
Slot 00000: SlotAdd Slot 1
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList
↳ [("unEndpointValue",Object (fromList [("mpAmount",Number 555.0),("mpTokenName",Object
↳ (fromList [("unTokenName",String "ABC"))]))]))]))))
```

(continues on next page)

(continued from previous page)

```

Slot 00001: W1: TxSubmit:
  ↳ 7c01d39fc031815eaf05d97709e4973a24dfa38e9dd68a4fd1ec92bb80cf76e4
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList
  ↳ [("unEndpointValue",Object (fromList [("mpAmount",Number 444.0),("mpTokenName",Object
  ↳ (fromList [("unTokenName",String "ABC"))]))]))]))))
Slot 00001: W2: TxSubmit:
  ↳ 6ba7eb4441992284e687d184080d4a8693e7b188fc45150d6e7ccd1243968f53
Slot 00001: TxnValidate 6ba7eb4441992284e687d184080d4a8693e7b188fc45150d6e7ccd1243968f53
Slot 00001: TxnValidate 7c01d39fc031815eaf05d97709e4973a24dfa38e9dd68a4fd1ec92bb80cf76e4
Slot 00001: SlotAdd Slot 2
Slot 00002: *** CONTRACT LOG: "forged Value (Map
  ↳ [(e01824b4319351c40b5ec727fff328a82076b1474a6bad6c8e8a2cd835cc6aaf,Map [(\\"ABC\\",
  ↳ 555))]))"
Slot 00002: *** CONTRACT LOG: "forged Value (Map
  ↳ [(e01824b4319351c40b5ec727fff328a82076b1474a6bad6c8e8a2cd835cc6aaf,Map [(\\"ABC\\",
  ↳ 444))]))"
Slot 00002: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList
  ↳ [("unEndpointValue",Object (fromList [("mpAmount",Number -222.0),("mpTokenName",Object
  ↳ (fromList [("unTokenName",String "ABC"))]))]))]))))
Slot 00002: W1: TxSubmit:
  ↳ 95d42e93ee41ab5bed7857b176be5a4e16602323eaacaa90f3bb807a9fd235c0
Slot 00002: TxnValidate 95d42e93ee41ab5bed7857b176be5a4e16602323eaacaa90f3bb807a9fd235c0
Slot 00002: SlotAdd Slot 3
Slot 00003: *** CONTRACT LOG: "forged Value (Map
  ↳ [(e01824b4319351c40b5ec727fff328a82076b1474a6bad6c8e8a2cd835cc6aaf,Map [(\\"ABC\\",-
  ↳ 222))]))"
Slot 00003: SlotAdd Slot 4
Final balances
Wallet 1:
  {, ""}: 99999980
  {e01824b4319351c40b5ec727fff328a82076b1474a6bad6c8e8a2cd835cc6aaf, "ABC"}: 333
Wallet 2:
  {e01824b4319351c40b5ec727fff328a82076b1474a6bad6c8e8a2cd835cc6aaf, "ABC"}: 444
  {, ""}: 99999990
...
Wallet 10:
  {, ""}: 1000000000

```

## 5.5 Example 2 - Signed

### 5.5.1 On-chain

Let's look at a more realistic example.

We'll take a copy of the Free module, and call it Signed.

Probably the easiest example of a realistic minting policy is one where the minting and burning of tokens is restricted to transactions that are signed by a specific public key hash. That is similar to a central bank, in fiat currencies.

This means that our policies is no longer without parameters. We need the public key hash. In addition, we are going to need to look at the context, so we can't just ignore it like last time.

We recall that *scriptContextTxInfo* from the context contains a list of all the signatories of the transaction. So, we can use this to see if the required signatory is one of them.

```
mkPolicy :: PubKeyHash -> ScriptContext -> Bool
mkPolicy pkh ctx = txSignedBy (scriptContextTxInfo ctx) pkh
```

The *txSignedBy* function is a convenient way of checking this. In previous examples, we used the *elem* function to check that it existed in the list.

```
Prelude Week05.Free> import Ledger
Prelude Ledger Week05.Free> :t txSignedBy
txSignedBy :: TxInfo -> PubKeyHash -> Bool
```

Now, we need to update the part of the code that compiles our *mkPolicy* function into Plutus code. We will use the same techniques that we have used when writing validator scripts. Specifically, we use the *applyCode* function to allows us to reference *pkh*, whose value is only known at runtime.

```
policy :: PubKeyHash -> Scripts.MonetaryPolicy
policy pkh = mkMonetaryPolicyScript $
  $(PlutusTx.compile [| | Scripts.wrapMonetaryPolicy . mkPolicy |])
  `PlutusTx.applyCode`
  PlutusTx.liftCode pkh
```

We also need to update the *curSymbol* function, as it now depends on the public key hash. It depends on it so that it can pass it to the *policy* function.

```
curSymbol :: PubKeyHash -> CurrencySymbol
curSymbol = scriptCurrencySymbol . policy
```

Note, the second line here, the body, is a shorter way of writing:

```
curSymbol pkh = scriptCurrencySymbol $ policy pkh
```

This is clear, when you consider something like the following, where *timesSix* is just another way of writing the results of combining the functions *timesTwo* and *timesThree*.

```
timesSix x = timesTwo $ timesThree x
```

is exactly the same as...

```
timesSix = timesTwo . timesThree
```

This process of simplification is called ETA reduction, so if you ever see your IDE hinting that you can ETA reduce, this is what it's talking about.

Now for the off-chain code.

### 5.5.2 Off-chain

We don't need to extend the *MintParams* data type for the off-chain code. A wallet that wants to mint or burn a currency can sign with its own public key hash. This is the only signature that a wallet can provide, and it has the ability to look it up for itself.

We will make a change to the name of the schema for clarity. We'll also, of course, update this name wherever it appears in the contract script.

```
type SignedSchema =
  BlockchainActions
    .\ Endpoint "mint" MintParams
```

Now, for the *mint* function, we need to pass the public key hash to the *curSymbol* function. Getting hold of the public key is something that is provided by *BlockchainActions*. So, we will get this from *Contract* and apply the *pubKeyHash* function to it.

One way to do this would be

```
pk <- Contract.ownPubKey
let pkh = pubKeyHash pk
```

However, as *Contract* is a monad, and therefore an instance of *Functor*, we have the *fmap* function available, which will turn a *Contract a* into a *Contract b*. In this case we can take advantage of that by using the *pubKeyHash* function as the (a -> b) function of *fmap* and this will turn *Contract pubKey* into *Contract pubKeyHash*, and then we can grab this value instead.

```
pkh <- fmap pubKeyHash Contract.ownPubKey
```

There is one more thing we can do to improve this. There is an operator for *fmap*.

```
pkh <- pubKeyHash <$> Contract.ownPubKey
```

Ok, now let's update the lookups line to pass in the public key hash.

```
lookups = Constraints.monetaryPolicy $ policy pkh
```

And now we have finished modifying the *mint* function.

```
mint :: MintParams -> Contract w SignedSchema Text ()
mint mp = do
  pkh <- pubKeyHash <$> Contract.ownPubKey
  let val      = Value.singleton (curSymbol pkh) (mpTokenName mp) (mpAmount mp)
      lookups  = Constraints.monetaryPolicy $ policy pkh
      tx       = Constraints.mustForgeValue val
  ledgerTx <- submitTxConstraintsWith @Void lookups tx
  void $ awaitTxConfirmed $ txId ledgerTx
  Contract.logInfo @String $ printf "forged %s" (show val)
```

So, let's try it out using the *test* function.

```
Prelude Ledger Week05.Signed> Week05.Signed.test
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
Slot 00000: SlotAdd Slot 1
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
```

(continues on next page)

(continued from previous page)

```

Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList
↳[("unEndpointValue",Object (fromList [("mpAmount",Number 555.0),("mpTokenName",Object
↳(fromList [("unTokenName",String "ABC"))]))]))]))))
Slot 00001: W1: TxSubmit:
↳20289e7b1bb6692b35e24e0f9293327f9169d843ae0ea431186fdefae6092a44
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList
↳[("unEndpointValue",Object (fromList [("mpAmount",Number 444.0),("mpTokenName",Object
↳(fromList [("unTokenName",String "ABC"))]))]))]))))
Slot 00001: W2: TxSubmit:
↳1c367cf81dd2da478abb96235ee16facf9f7d47374c9455d5fdd516aaf04d0c2
Slot 00001: TxnValidate 1c367cf81dd2da478abb96235ee16facf9f7d47374c9455d5fdd516aaf04d0c2
Slot 00001: TxnValidate 20289e7b1bb6692b35e24e0f9293327f9169d843ae0ea431186fdefae6092a44
Slot 00001: SlotAdd Slot 2
Slot 00002: *** CONTRACT LOG: "forged Value (Map
↳[(7183b1cf81e44b26c558ddf442c4a7161a1b504b61136a8773dc2e4960323521,Map [(\\"ABC\\",
↳555))]))"
Slot 00002: *** CONTRACT LOG: "forged Value (Map
↳[(2a964fa6314803cf1b61165aeb1d758e355aae9480a29e282b58e76983f101ba,Map [(\\"ABC\\",
↳444))]))"
Slot 00002: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList
↳[("unEndpointValue",Object (fromList [("mpAmount",Number -222.0),("mpTokenName",Object
↳(fromList [("unTokenName",String "ABC"))]))]))]))))
Slot 00002: W1: TxSubmit:
↳6e20d243447d7f49de509ef6b52c6d947769d95a6451c9cda53e42a0ba02fa69
Slot 00002: TxnValidate 6e20d243447d7f49de509ef6b52c6d947769d95a6451c9cda53e42a0ba02fa69
Slot 00002: SlotAdd Slot 3
Slot 00003: *** CONTRACT LOG: "forged Value (Map
↳[(7183b1cf81e44b26c558ddf442c4a7161a1b504b61136a8773dc2e4960323521,Map [(\\"ABC\\",-
↳222))]))"
Slot 00003: SlotAdd Slot 4
Final balances
Wallet 1:
  {, ""}: 99999980
  {7183b1cf81e44b26c558ddf442c4a7161a1b504b61136a8773dc2e4960323521, "ABC": 333
Wallet 2:
  {2a964fa6314803cf1b61165aeb1d758e355aae9480a29e282b58e76983f101ba, "ABC": 444
  {, ""}: 99999990
...
Wallet 10:
  {, ""}: 100000000

```

This looks very similar to before, but this time, notice that, while the token names are the same, the currency symbols are different for each wallet.



## 5.6 NFTs

Let's now talk about NFTs - Non-Fungible Tokens. NFTs are tokens that have a quantity of exactly 1.

The examples of native tokens that we have studied so far are definitely not NFTs because we could easily mint as many as we wanted. This is true not only in the first example where anyone could mint tokens, but also in the second example, where, so long as you are the owner of the correct public key hash, you could mint unlimited tokens for the associated currency symbol and token name.

In order to produce an NFT, perhaps the first naive idea would be to look at forge field in the policy and enforce a policy where the amount is one.

But that wouldn't help us. That would only mean that during one transaction you can mint only one token. But nobody could stop us from submitting as many of those transactions as we like.

The second option is actually in use already on the Cardano blockchain. NFTs have been available since the Mary fork, which predates Plutus, and to do this, they are implemented using deadlines.

We saw in previous examples how time can be incorporated in validation scripts, and the same can be done in policy scripts.

The idea here is to only allow minting before a given deadline has passed. Using this method, if you want to mint an NFT, you mint one token before the deadline, then allow the deadline to pass. This guarantees that, after the deadline, no new tokens will ever be minted.

But, in order to check that you only minted one token before the deadline, you need something like a blockchain explorer. So, in this sense, they are not true NFTs, insofar as the currency symbol itself guarantees that they are unique.

Using Plutus, it is possible to mint true NFTs. If you know the policy script that corresponds to the currency symbol, you can be sure that only one token is in existence without having to resort to something like a blockchain explorer.

And, thinking about how to do that, there must be a way to prevent there ever being more than one minting transaction for the token in question. Whatever you write in your policy script, it must only return true for one transaction, so that it is impossible to do the same again in another transaction.

At first, this sounds impossible. Why can't you just run the same transaction again and have validation succeed again? Even considering deadlines, what stops a second transaction in the same slot from passing validation?

The key here is that we need something unique. Something that can only exist in one transaction and never again. This is an important trick, and it is something to keep in mind.

The idea is to use UTxOs. A UTxO is unique. A UTxO is the output of a transaction and its unique identifier is the transaction ID and its index in the list of outputs from that transaction.

The reason that transactions are unique is a bit subtle. They would not necessarily be unique if it were not for fees. Without fees, you could have a transaction that has zero inputs and only with outputs without value. Such a transaction would have the exact same hash each time it was run, and therefore the exact same transaction id. But with fees, such a transaction cannot exist, as you always need an input that provides fees, and the fees can never come from the same UTxO as input.

So, to create an NFT, we are going to provide a specific UTxO as a parameter to the minting policy and, in the policy, we are going to check that the transaction consumes this UTxO. And, as we have just noted, once that UTxO is consumed, it can never be consumed again.

### 5.6.1 Example 3 - NFT

We start with a copy of the previous example, *Signed* and we will call it *NFT*.

So let's turn the signed policy into a true NFT policy.

#### On-chain

First, we will no longer use the public key hash as an input, as if we were a central bank, but will use a UTxO instead. So, what type corresponds to a UTxO?

Let's look in the REPL and remind ourselves about *TxInfo*.

```
Prelude Week05.Signed Week05.Free> import Ledger
Prelude Week05.Signed Ledger Week05.Free> :i TxInfo
type TxInfo :: *
data TxInfo
  = TxInfo {txInfoInputs :: [TxInInfo],
            txInfoInputsFees :: [TxInInfo],
            txInfoOutputs :: [TxOut],
            txInfoFee :: Value,
            txInfoForge :: Value,
            txInfoDCert :: [Plutus.V1.Ledger.DCert.DCert],
            txInfoWdrl :: [(Plutus.V1.Ledger.Credential.StakingCredential,
                           Integer)],
            txInfoValidRange :: SlotRange,
            txInfoSignatories :: [PubKeyHash],
            txInfoData :: [(DatumHash, Datum)],
            txInfoId :: TxId}
```

We are interested in this field:

```
txInfoInputs :: [TxInInfo]
```

Let's look at the type *TxInInfo*

```
Prelude Week05.Signed Ledger Week05.Free> :i TxInInfo
type TxInInfo :: *
data TxInInfo
  = TxInInfo {txInInfoOutRef :: TxOutRef, txInInfoResolved :: TxOut}
```

We see that it is a record with two fields. The first is of type *TxOutRef*, and this references a UTxO, which is exactly what we need. So, let's use it.

```
mkPolicy :: TxOutRef -> ScriptContext -> Bool
```

Now, we are ready to write the logic. We must check that the script contains the specified UTxO as input. We will delegate this to a helper function. This function, which we will call *hasUTxO* uses the *any* function, which is a standard Prelude function, but also has a Plutus version, for reasons we have addressed previously.

The *any* function takes a predicate (a function that returns a boolean) and applies it to an input collection of the type *Foldable* (a list, for example), and will return true if the predicate is true for any of the inputs.

Here, we use the *any* function to see if any of the *txInInfoOutRef*'s from the *\*txInfoInputs* from the *TxInfo* field of the context matches the UTxO for which we are validating.

For clarity, we will also provide a helper function to get the list of *txInfoInputs*.

```
info :: TxInfo
info = scriptContextTxInfo ctx

hasUTx0 :: Bool
hasUTx0 = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs info
```

So, do we have enough to finish writing our policy? Let's see what we have.

```
mkPolicy :: TxOutRef -> ScriptContext -> Bool
mkPolicy oref ctx = traceIfFalse "UTx0 not consumed" hasUTx0
where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    hasUTx0 :: Bool
    hasUTx0 = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs info
```

Right now, we have a policy that can only mint or burn once. But, of course, in that single transaction, we can still mint as many tokens as we like.

Now, we think about what we actually want. Maybe we want a policy that allows us to mint just one token for the currency symbol. Or perhaps, we would like to be able to mint many NFTs at once, each with a different token name.

It's up to us. But, let's say we go with the first option. We just want to mint one token.

So, it makes sense to pass the token name as a parameter.

```
mkPolicy :: TxOutRef -> TokenName -> ScriptContext -> Bool
```

And we need a second condition that checks that we mint just this one specific coin.

```
mkPolicy oref tn ctx = traceIfFalse "UTx0 not consumed" hasUTx0 &&
                        traceIfFalse "wrong amount minted" checkMintedAmount
```

And, of course, we need to implement *checkMintedAmount*.

First of all, we need access to the forged value. We get this from the field *txInfoForge* of *TxInfo*.

How do we check that this forged value is exactly 1 token of the name that we require? There are several approaches, but one is to use the *flattenValue* function which, we will recall, returns a list of triples of currency symbol, token name and value. We can then check that the output of *flattenValue* is exactly one triple that matches the symbol, token and value that we expect.

This would look something like this:

```
flattenValue (txInfoForge info) == [(cs, tn, 1)]
```

But we still have a problem to solve - we need to know what the currency symbol is. Given that the currency symbol is a hash of the policy, it seems as if we have a chicken and egg problem.

As luck would have it, there is a function called *ownCurrencySymbol* which exists to solve exactly this problem.

```
flattenValue (txInfoForge info) == [(ownCurrencySymbol ctx, tn, 1)]
```

As it happens, this won't compile, because *Eq* is not defined for triples in the Plutus Prelude. So, we can work around this with a case statement and some pattern matching.

```
case flattenValue (txInfoForge info) of
  [(cs, tn', amt)] -> cs == ownCurrencySymbol ctx && tn' == tn && amt == 1
  _                 -> False
```

Now, we can complete our policy.

```
mkPolicy :: TxOutRef -> TokenName -> ScriptContext -> Bool
mkPolicy oref tn ctx = traceIfFalse "UTxO not consumed" hasUTxO &&
  traceIfFalse "wrong amount minted" checkMintedAmount
where
  info :: TxInfo
  info = scriptContextTxInfo ctx

  hasUTxO :: Bool
  hasUTxO = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs info

  checkMintedAmount :: Bool
  checkMintedAmount = case flattenValue (txInfoForge info) of
    [(cs, tn', amt)] -> cs == ownCurrencySymbol ctx && tn' == tn && amt == 1
    _                 -> False
```

And we will update our boilerplate.

```
policy :: TxOutRef -> TokenName -> Scripts.MonetaryPolicy
policy oref tn = mkMonetaryPolicyScript $
  $(PlutusTx.compile [| | \oref' tn' -> Scripts.wrapMonetaryPolicy $ mkPolicy oref' tn
  -> ' | |])
  `PlutusTx.applyCode`
  PlutusTx.liftCode oref
  `PlutusTx.applyCode`
  PlutusTx.liftCode tn

curSymbol :: TxOutRef -> TokenName -> CurrencySymbol
curSymbol oref tn = scriptCurrencySymbol $ policy oref tn
```

That completes the on-chain part.

## Off-chain

We need to think about the inputs we need for this transaction.

First, we need a UTxO, and we need to provide one of our own. However, we don't need to pass that in because we can look it up directly.

We only need to provide the token name, so we no longer need a special data type, so we can delete *MintParams* and just use *TokenName*.

```
type NFTSchema =
  BlockchainActions
  .\ Endpoint "mint" TokenName
```

Now we will write the off-chain *mint* function.

```
mint :: TokenName -> Contract w NFTSchema Text ()
mint tn = do
```

The first thing to do is to get the list of UTxOs that belong to us.

The *Plutus.Contract* module gives us the *utxoAt* function, which has the signature below, and looks up all the UTxOs at a given address.

```
utxoAt :: Address -> Contract w s e Ledger.AddressMap.UtxoMap
```

An *AddressMap* is a map where the keys are *\*TxOutRef\**s and the values are *\*TxOutTx\**s.

```
Prelude Week05.NFT> :i Ledger.AddressMap.UtxoMap
type Ledger.AddressMap.UtxoMap :: *
type Ledger.AddressMap.UtxoMap = Data.Map.Internal.Map TxOutRef TxOutTx
```

If we pass this function our own address then the keys of this map will be the UTxOs that belong to us. It doesn't matter which one of these we pick. So long as we own at least one UTxO, we are good.

The first step is to find our own address. We know how to find our own public key, and, given this, we can use the function *pubKeyAddress* to get our address.

```
pubKeyAddress :: PubKey -> address
```

Let's get them.

```
import qualified Data.Map as Map

mint :: TokenName -> Contract w NFTSchema Text ()
mint tn = do
    pk    <- Contract.ownPubKey
    utxos <- utxoAt (pubKeyAddress pk)
```

We only need one - we don't care which one. We will write a case statement that will either log an error if we have no UTxO available, or will use the first UTxO in the list continue with the forging code.

The first change is to specify *1* instead of the *mpAmount*, as we want exactly 1 coin minted.

```
case Map.keys utxos of
    []      -> Contract.logError @String "no utxo found"
    oref : _ -> do
        let val      = Value.singleton (curSymbol oref tn) tn 1
```

Secondly, we add the token name argument to the lookups.

```
lookups = Constraints.monetaryPolicy $ policy oref tn
```

Thirdly, we now need an additional constraint which insists that our specific UTxO is consumed.

There's a function for that.

```
Prelude Week05.NFT> import Ledger.Constraints
Prelude Week05.NFT> :t mustSpendPubKeyOutput
mustSpendPubKeyOutput :: TxOutRef -> TxConstraints i o
```

How do we combine the constraints of *mustForgeValue* and *mustSpendPubKeyOutput*? *Constraints* don't form a *Monoid*, but they do form a *Semigroup*, and the difference is just that in *Semigroup* we don't have *mempty*, the neutral element. We can still combine them with the `<>` operator.

```
tx = Constraints.mustForgeValue val <> Constraints.mustSpendPubKeyOutput oref
```

Now, we need to provide a lookup that gives access to where the UTxO *oref* can be found. For that we can use

```
Ledger.Constraints.unspentOutputs :: Data.Map.Internal.Map TxOutRef TxOutTx ->
->ScriptLookups a
```

So, let's update our lookups.

```
lookups = Constraints.monetaryPolicy (policy oref tn) <> Constraints.unspentOutputs utxos
```

Something we need to do before this script will run is to import the operator `<>` for *Semigroup* from the standard Haskell Prelude, as we have explicitly excluded it from the *PlutusTx.Prelude* module.

```
import Prelude (Semigroup (..))
```

Let's take a look at the whole function.

```
mint :: TokenName -> Contract w NFTSchema Text ()
mint tn = do
  pk    <- Contract.ownPubKey
  utxos <- utxoAt (pubKeyAddress pk)
  case Map.keys utxos of
    []    -> Contract.logError @String "no utxo found"
    oref : _ -> do
      let val      = Value.singleton (curSymbol oref tn) tn 1
          lookups  = Constraints.monetaryPolicy (policy oref tn) <> Constraints.
->unspentOutputs utxos
          tx       = Constraints.mustForgeValue val <> Constraints.
->mustSpendPubKeyOutput oref
          ledgerTx <- submitTxConstraintsWith @Void lookups tx
          void $ awaitTxConfirmed $ txId ledgerTx
          Contract.logInfo @String $ printf "forged %s" (show val)
```

For the test script.

```
test :: IO ()
test = runEmulatorTraceIO $ do
  let tn = "ABC"
  h1 <- activateContractWallet (Wallet 1) endpoints
  h2 <- activateContractWallet (Wallet 2) endpoints
  callEndpoint @"mint" h1 tn
  callEndpoint @"mint" h2 tn
  void $ Emulator.waitNSlots 1
```

Let's test.

```
Prelude Week05.Signed Ledger Plutus.Contract Ledger.Constraints Week05.Free> Week05.NFT.
->test
Slot 00000: TxnValidate af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
Slot 00000: SlotAdd Slot 1
```

(continues on next page)

(continued from previous page)

```

Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList_
↳[("unEndpointValue",Object (fromList [("unTokenName",String "ABC"))]))))]
Slot 00001: W1: TxSubmit:_
↳691a5c0725ac09f79c8c45c899d732d26460d18c4c18167be71d55319bcd5669
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
Receive endpoint call: Object (fromList [("tag",String "mint"),("value",Object (fromList_
↳[("unEndpointValue",Object (fromList [("unTokenName",String "ABC"))]))))]
Slot 00001: W2: TxSubmit:_
↳e53519b17bf7d11a148ce17ac0305330f138a684530ba08b1c57f714672b8c68
Slot 00001: TxnValidate e53519b17bf7d11a148ce17ac0305330f138a684530ba08b1c57f714672b8c68
Slot 00001: TxnValidate 691a5c0725ac09f79c8c45c899d732d26460d18c4c18167be71d55319bcd5669
Slot 00001: SlotAdd Slot 2
Slot 00002: *** CONTRACT LOG: "forged Value (Map_
↳[(9d969e597d45fcd1732ce255e12a97599e883f924b4565fc3a2407bc08d34524,Map [(\\"ABC\\",1)])])
↳"
Slot 00002: *** CONTRACT LOG: "forged Value (Map_
↳[(913f220c3b1ba49531bae2fedd9edb138a8b360e7e605bfcf4ff3f2045433069,Map [(\\"ABC\\",1)])])
↳"
Slot 00002: SlotAdd Slot 3
Final balances
Wallet 1:
  {9d969e597d45fcd1732ce255e12a97599e883f924b4565fc3a2407bc08d34524, "ABC"}: 1
  {, ""}: 99999990
Wallet 2:
  {913f220c3b1ba49531bae2fedd9edb138a8b360e7e605bfcf4ff3f2045433069, "ABC"}: 1
  {, ""}: 99999990
...
Wallet 10:
  {, ""}: 100000000

```

And now we have minted some NFTs.





## WEEK 06 - ORACLES

---

**Note:** These is a written version of [Lecture #6](#).

In this lecture we learn about oracles and using the PAB (Plutus Application Backend).

These notes use Plutus commit 476409eae94141e2fe076a7821fc2fcdec5dfcb

---

### 6.1 Overview

In this lecture we are going to look at a case study, to see how what we have learned so far can be turned into an actual application. A collection of executables that even come with a little front end.

It will be a real dApp, apart from the fact that we don't have a real blockchain available yet. This will run on a simulated blockchain - a mockchain.

The example we are going to use for this is to implement a very simple oracle.

---

**Note:** In the blockchain world, an oracle is a way to get real-world information onto the blockchain, in order to make it usable in smart contracts.

---

There are numerous examples of use cases for oracles. We can think of external data sources such as weather data, election results, stock exchange data or randomness. You may have a betting contract that depends on the outcome of a specific sports game, for example.

There are various ways to implement oracles, of varying sophistication.

We are going to use a very simple approach, where we have one trusted data provider. And, as an example of data, we are going to use the ADA/USD exchange rate.

There are lots of problems with this approach, as we have to trust the data source. There are ways to mitigate the risk that the data source is either untrustworthy or unreliable. For example, we could ask the provider to put down some collateral that is lost if data is not provided or is inaccurate. Or, you could combine several oracles into one and only accept the result if they all agree, or take the median, or average value of various sources. You could also come up with more sophisticated mechanisms.

As we know, for anything to happen on the blockchain, there must be a UTxO, so the obvious thing to do is to represent the data feed as a UTxO. The UTxO sits at the script address of the oracle, and its datum field it carries the current value of the oracle data.

And this is where we find our first problem. As we have noted before, validation only happens when you want to consume something from a script address, not when you produce an output at a script address. This means that we can't prevent anybody producing arbitrary outputs at the script address.



Somehow we need to distinguish the true oracle output from other outputs that may be sitting at the same script address. And the way we do this is to put an NFT on the output. Because an NFT can only exist once, there can only be one UTxO at the script address that holds the NFT.



How can such an oracle be used?

Here we come to something we haven't seen before. In all our code writing validators and contracts, we always knew the full API up front. In the case of an oracle, this is different. At the point that an oracle is created, you don't know how people may want to use it. It must be like an open API, able to work with smart contracts that have not yet been designed.

As an example of a use-case that might make use of this specific oracle, let's consider a swap contract where, at the swap address, somebody can deposit ADA, and then somebody else can take those ADA in exchange for USD.

Of course, we don't have USD directly on the blockchain, but we can imagine that they are represented by some native token.

In this example, as the value at the oracle is 1.75, then if someone offers 100 ADA, the price for that should be 175 USD.

In addition to this, we need an incentive for the oracle to provide the data, because in addition to other costs for providing the data, at a minimum they would have to pay fees to create the UTxO.

So, let's say that the oracle provider determines a fee of 1 ADA that has to be paid each time the oracle is used.

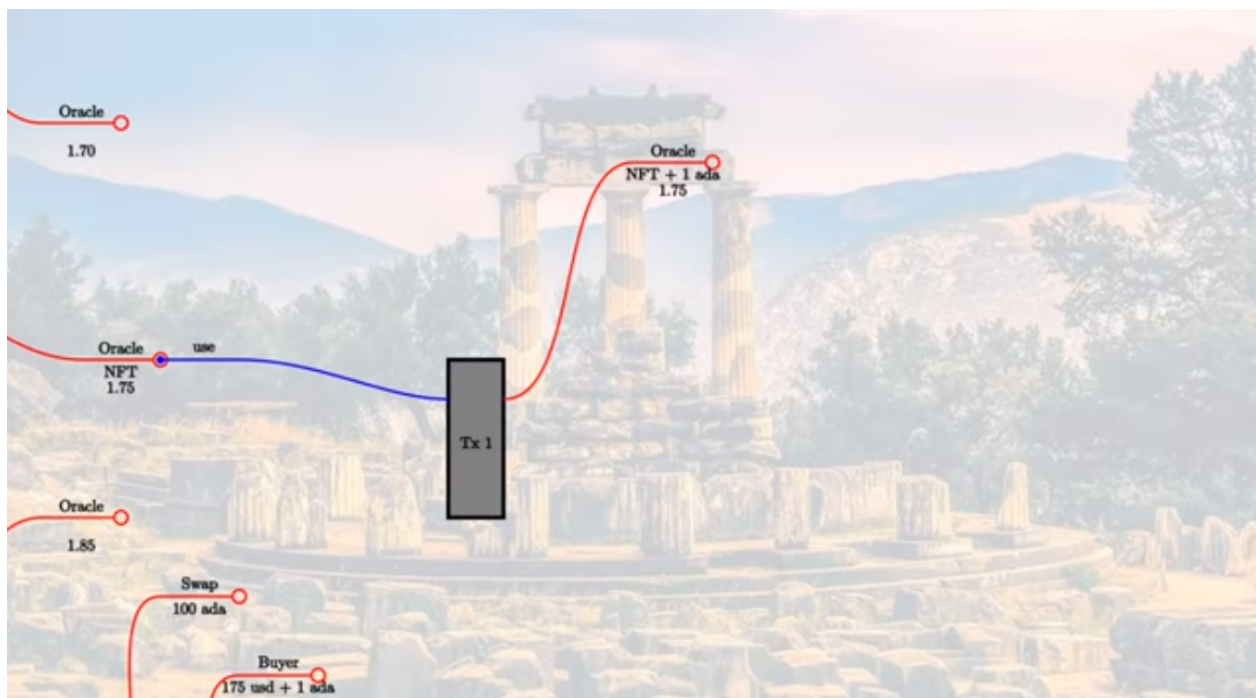
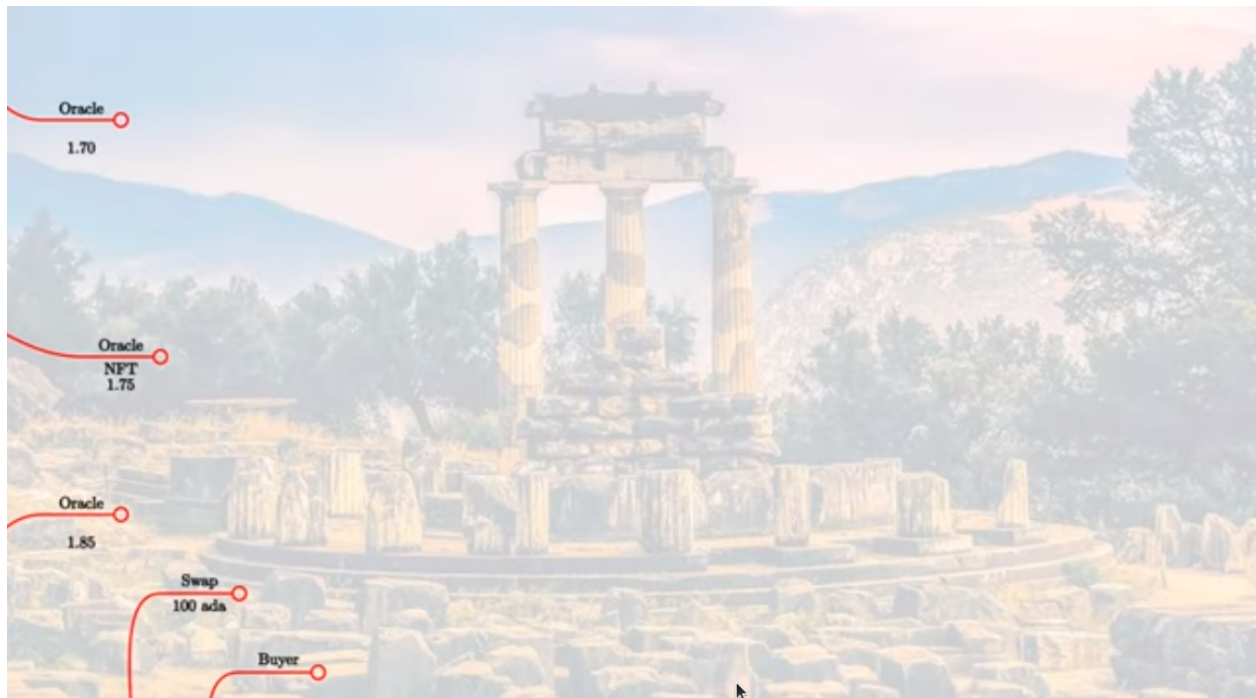
In this example, that would mean that the person wanting to buy the ADA would have to pay 175 USD to the seller of the ADA, and 1 ADA to the oracle.

What will the transaction look like?

First of all, the swap validation logic will need access to the current oracle value, which means that the oracle UTxO must be an input to the transaction.

Then we have the oracle validation logic. In this case we want to use the oracle. So, let's say we have a redeemer called *use*. Now, the oracle validator has to check several things.

1. Is the NFT present in the consumed input?

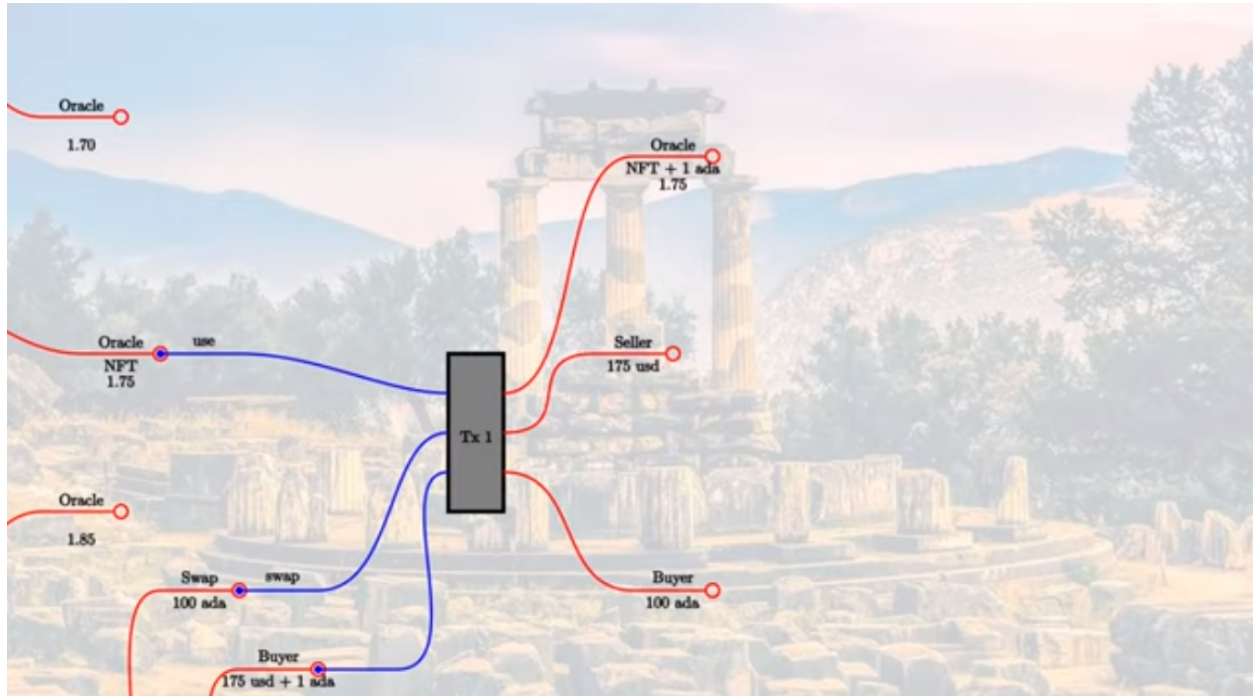




2. Is there an output from the transaction at the same address containing the same NFT?
3. Is the value in the output UTxO the same as the input value?
4. Is the fee present?

Now we can complete the transaction.

We consume two additional inputs - the fee paid by the buyer and the 100 ADA deposited by the seller. Then we have two additional outputs - the 175 USD to the seller, and the 100 ADA to the buyer. And for these new inputs and outputs, it is the responsibility of the swap validator to make sure that it is correct. Whereas, the oracle validator is only interested with making sure that everything concerning the oracle is correct.



Just to emphasize, this swap contract is just an example. The oracle should be capable of working with many different smart contracts that want to make use of its data.

If this was all, then we wouldn't need an oracle. If the value was fixed, so that it was always 1.75 then we could simply hard-code this into our contract. So, the value must be able to change. At least, in an example such as this one where we have an exchange rate that can, of course, change over time. There may be other examples such as the result of a sports match, where it is a singular event in history, but in this case, it is important that it be able to change.

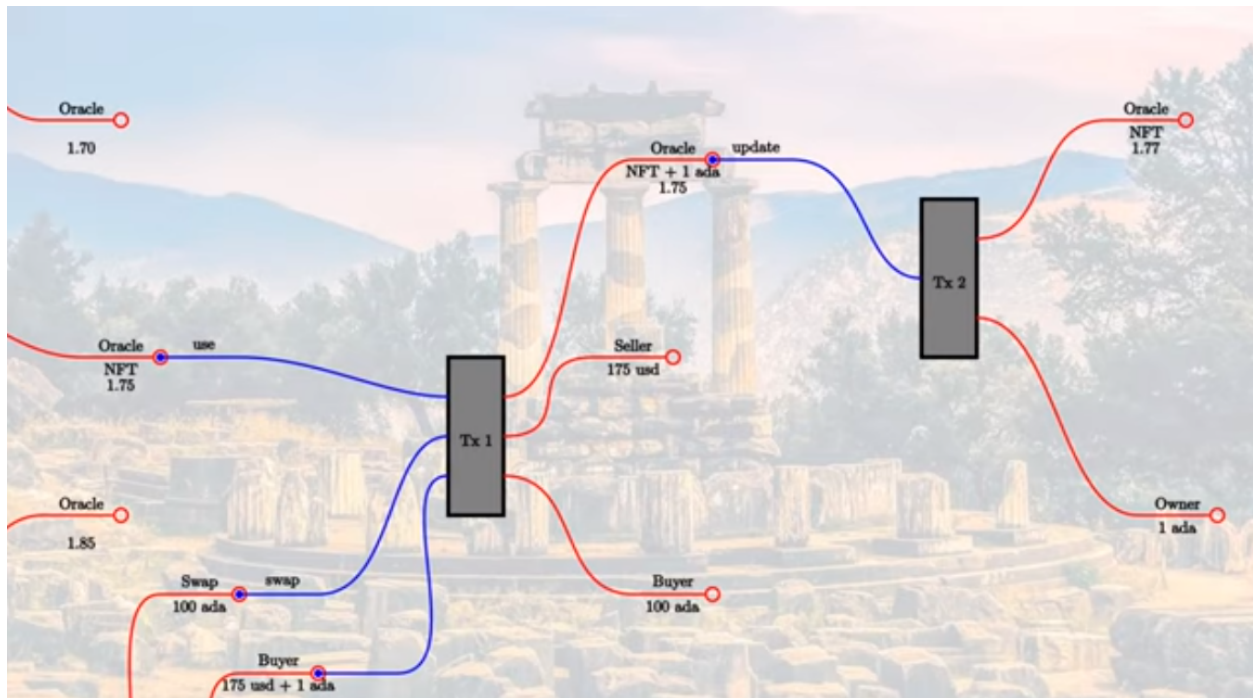
This means that the oracle validator, in addition to the *use* redeemer, must be able to support another operation where the provider of the oracle can actually change the data.

So let's say the the value changes from 1.75 to 1.77.

We know that on a (E)UTxO blockchain, nothing ever changes, so you can't change the datum of an existing UTxO. All you can do is consume UTxOs and produce new ones.

We would have a transaction that uses an *update* redeemer. The validation logic is somewhat different. It is the same as before in that the NFT needs to be present in the consumed oracle input, and also needs to be present in the new output. In addition to that, the transaction must be signed by the oracle provider. And, we can use this update transaction as an opportunity for the oracle provider to collect the fees.

We insist that the NFT be present in the output, but we don't say anything about other values. All the fees that got there by other transactions using this oracle data can be collected during the *update* transaction.



### 6.1.1 Summary

To sum up, we represent the oracle by a UTxO and identify the correct UTxO with an NFT. The oracle value is the datum of the UTxO. We support two operations.

One is *use* which uses the oracle in some arbitrary transaction. The *use* validator will make sure that the consumed oracle input carries the NFT, that there is an output that again carries the NFT, doesn't change the datum, and carries additional fees.

The second operation is *update* which can only be done by the oracle provider. For an *update* transaction, the oracle input must again carry the NFT, there must be an oracle output, also carrying the NFT. There are no further restrictions. The datum can change, and the accumulated fees can be taken out.

## 6.2 Oracle Core

Now that we know how it is supposed to work, let's look at some code.

### 6.2.1 On-chain

First, let's look at the Plutus code that implements the oracle itself.

```
module Week06.Oracle.Core
```

The oracle will be a parameterized contract, and it will depend on four fields.

```
data Oracle = Oracle
  { oSymbol  :: !CurrencySymbol
  , oOperator :: !PubKeyHash
```

(continues on next page)

(continued from previous page)

```
, oFee      :: !Integer
, oAsset    :: !AssetClass
} deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq, Prelude.Ord)
```

- *oSymbol* is the currencySymbol of the NFT that is used to identify the transaction. We don't need the token name as we will just use the empty string as the token name.
- *oOperator* is the owner of the oracle - the hash of the public key owner which can make updates
- *oFee* is the fee in lovelace that is due every time the oracle is used
- *oAsset* represents the asset class that we want to exchange rate for against Ada, which in our case will be some kind of USD token

The redeemer will support two operations.

```
data OracleRedeemer = Update | Use
  deriving Show

PlutusTx.unstableMakeIsData 'OracleRedeemer
```

We need to define the NFT asset class. As mentioned, we are going to use the empty string for the token name.

```
{-# INLINABLE oracleTokenName #-}
oracleTokenName :: TokenName
oracleTokenName = TokenName emptyByteString
```

The *oracleAsset* will be used to identify the NFT - this is not to be confused with *oAsset*, defined above.

```
{-# INLINABLE oracleAsset #-}
oracleAsset :: Oracle -> AssetClass
oracleAsset oracle = AssetClass (oSymbol oracle, oracleTokenName)
```

We create a little helper function called *oracleValue*. This takes an output transaction and a function which looks up the datum, and then returns an *Integer*. The *Integer* represents the exchange rate (e.g. 1.75) multiplied by a million. This avoids potential complications when using real numbers.

```
{-# INLINABLE oracleValue #-}
oracleValue :: TxOut -> (DatumHash -> Maybe Datum) -> Maybe Integer
oracleValue o f = do
  dh    <- txOutDatum o
  Datum d <- f dh
  PlutusTx.fromData d
```

This function is an example of monadic computation in monad that is not *IO* or the *Contract* monad. First we call *txOutDatum*, which can fail if because not every output has a datum. If it succeeds, we get a datum hash which we can reference in *dh*. Next we used the function *f* which is provided as the second argument to maybe turn this datum hash into a datum. This too can fail. If it succeeds we can reference the result in *d*. *Datum* is just a newtype wrapper around *Data*, so we can then use *PlutusTx.fromData* to maybe turn *d* into an *Integer*. Again, this can fail, because even if the datum is there, it may not be convertible to an integer value.

We will see in a moment where we use the *oracleValue* function.

The most important function is *mkOracleValidator*.

```

{-# INLINABLE mkOracleValidator #-}
mkOracleValidator :: Oracle -> Integer -> OracleRedeemer -> ScriptContext -> Bool
mkOracleValidator oracle x r ctx =
  traceIfFalse "token missing from input" inputHasToken &&
  traceIfFalse "token missing from output" outputHasToken &&
  case r of
    Update -> traceIfFalse "operator signature missing" (txSignedBy info $ oOperator_
    ↪ oracle) &&
      traceIfFalse "invalid output datum" validOutputDatum
      Use -> traceIfFalse "oracle value changed" (outputDatum == Just x)
    ↪ &&
      traceIfFalse "fees not paid" feesPaid
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    ownInput :: TxOut
    ownInput = case findOwnInput ctx of
      Nothing -> traceError "oracle input missing"
      Just i -> txInInfoResolved i

    inputHasToken :: Bool
    inputHasToken = assetClassValueOf (txOutValue ownInput) (oracleAsset oracle) == 1

    ownOutput :: TxOut
    ownOutput = case getContinuingOutputs ctx of
      [o] -> o
      _ -> traceError "expected exactly one oracle output"

    outputHasToken :: Bool
    outputHasToken = assetClassValueOf (txOutValue ownOutput) (oracleAsset oracle) == 1

    outputDatum :: Maybe Integer
    outputDatum = oracleValue ownOutput (`findDatum` info)

    validOutputDatum :: Bool
    validOutputDatum = isJust outputDatum

    feesPaid :: Bool
    feesPaid =
      let
        inVal = txOutValue ownInput
        outVal = txOutValue ownOutput
      in
        outVal `geq` (inVal <> Ada.lovelaceValueOf (oFee oracle))

```

The function `mkOracleValidator` takes our parameter `Oracle`, the datum, which, in this example is an `Integer`, the redeemer type `OracleRedeemer` and finally the `ScriptContext`.

There are two cases for this validator - *use* and *update* - but there are similarities. In both cases we want to check that we have the input that holds the NFT and that there is an output that holds the NFT.

As both these checks need to be done regardless of the use case, they are done upfront.



```
...
traceIfFalse "token missing from input" inputHasToken &&
traceIfFalse "token missing from output" outputHasToken &&
...
```

After this, we consider which use case we are dealing with.

```
case r of
  Update -> traceIfFalse "operator signature missing" (txSignedBy info $ oOperator_
    ↪ oracle) &&
    traceIfFalse "invalid output datum"          validOutputDatum
  Use    -> traceIfFalse "oracle value changed"    (outputDatum == Just x)
    ↪ &&
    traceIfFalse "fees not paid"                  feesPaid
```

Before looking at the *inputHasToken* function there is another help function to look at.

```
ownInput :: TxOut
ownInput = case findOwnInput ctx of
  Nothing -> traceError "oracle input missing"
  Just i   -> txInInfoResolved i
```

The *ownInput* function returns the *TxOut* that the script is trying to consume, which in this case is the oracle output. The *Nothing* case here can happen if we are in a different context, such as a minting context, so this eventuality will not occur for us. The *findOwnInput* function is provided by Plutus and will, given the context, find the relevant input. The *txInInfoResolved* function gets the *TxOut* from the *TxInInfo*.

The *inputHashToken* function checks that the token is present. It uses the *assetClassValueOf* function to look for the NFT within the *ownInput* response.

```
inputHasToken :: Bool
inputHasToken = assetClassValueOf (txOutValue ownInput) (oracleAsset oracle) == 1
```

The next helper function, *ownOutput* checks that we have exactly one output and returns that output to us.

```
ownOutput :: TxOut
ownOutput = case getContinuingOutputs ctx of
  [o] -> o
  _    -> traceError "expected exactly one oracle output"
```

We can use this for the *outputHasToken* helper function in the same way as we did for the *inputHashToken* function.

```
outputHasToken :: Bool
outputHasToken = assetClassValueOf (txOutValue ownOutput) (oracleAsset oracle) == 1
```

That covers the code for the common cases. Now, let's look at the code specific to the *update* case.

There are two conditions to check. The first is that the operator actually signed the transaction. This is so simple that we can do it inline without a helper function.

```
traceIfFalse "operator signature missing" (txSignedBy info $ oOperator oracle)
```

The next thing to check is that the output datum. We know that the value can change, but we need to check that it is at least of the correct type.

```
traceIfFalse "invalid output datum" validOutputDatum
```

And for this we have referenced a new helper function *validOutputDatum*, which itself makes use of a helper function *outputDatum*.

```
outputDatum :: Maybe Integer
outputDatum = oracleValue ownOutput (`findDatum` info)

validOutputDatum :: Bool
validOutputDatum = isJust outputDatum
```

---

**Note:** If you look up *findDatum* in the REPL, you will see it has a type of *DatumHash -> TxInfo -> Maybe Datum*. As we are using its infix notation here, we can pass in *info* as the only parameter, and this will result in the whole expression having the type *DatumHash -> Maybe Datum*, which is the type we need to pass into *oracleValue*.

---

This works by trying to get the datum value from the datum hash and then trying to create the oracle value from it. If it succeeds it will return a *Just Integer*, otherwise it will return *Nothing*, so the *validOutputDatum* function just needs to check that the return value is not *Nothing*, in other words, that it is a *Just*.

Note that we are not checking anything about the value of the *Integer*. This could even remain the same as the input value, if the transaction is used just to collect the fees that have accumulated from the use the oracle.

The second case for *mkOracleValidator* is the *use* case. This case can be used by anyone, but it is much more restrictive. First, we don't allow the value to change. So this is the first condition.

```
traceIfFalse "oracle value changed" (outputDatum == Just x)
```

We have already written the *outputDatum* helper function. Instead of checking only that it is an *Integer*, here we also check that its output value is the same as the input value.

And finally, we must check that the fees have been paid. And for this we use a new helper function called *feesPaid*.

```
feesPaid :: Bool
feesPaid =
  let
    inVal  = txOutValue ownInput
    outVal = txOutValue ownOutput
  in
    outVal `geq` (inVal <> Ada.lovelaceValueOf (oFee oracle))
```

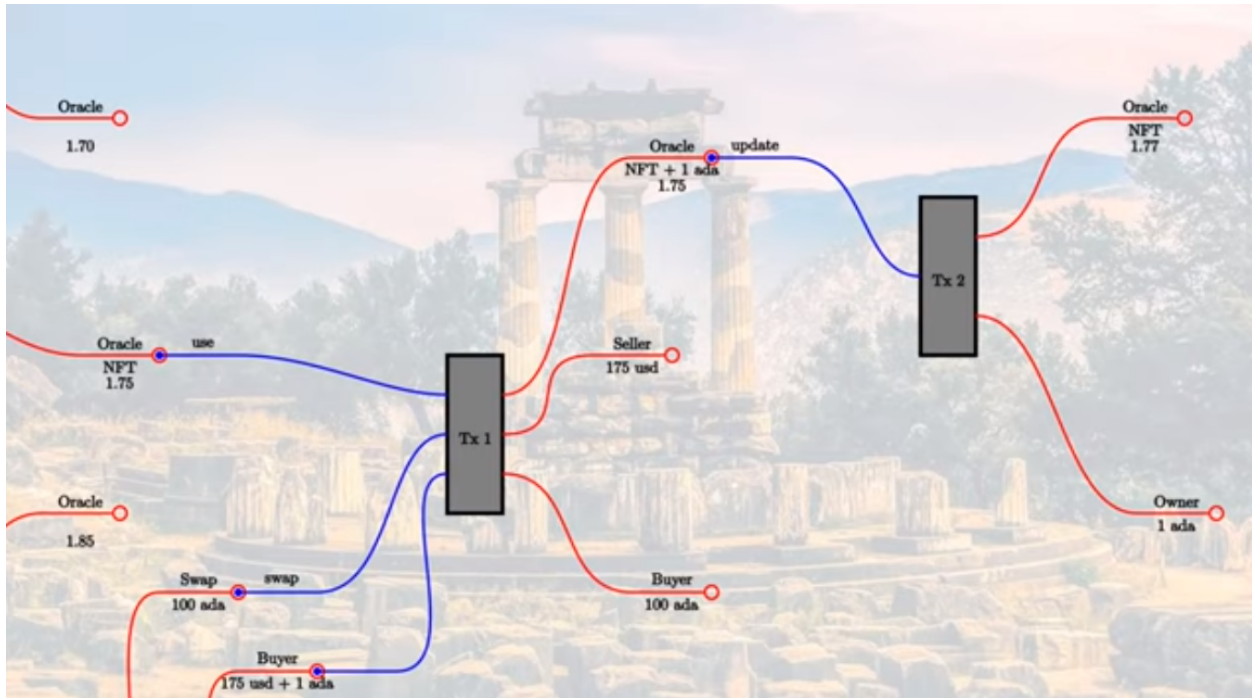
This *feesPaid* function checks that the output value is at least as much as the input value plus the required fee. We again use the semigroup operator *<>* to add the fee value to the input value. We could have used equal (*eq*) instead of greater than or equal (*geq*). Using *geq* allows the user of the oracle to give the oracle provider a tip, if they so wish.

So this now is basically the core business logic of the oracle as shown in the diagrams.

Now we have our usual boilerplate. In particular notice that we use the pattern that we need for a parameterized validator.

```
data Oracling
instance Scripts.ScriptType Oracling where
  type instance DatumType Oracling = Integer
  type instance RedeemerType Oracling = OracleRedeemer
```

(continues on next page)



(continued from previous page)

```

oracleInst :: Oracle -> Scripts.ScriptInstance Oracling
oracleInst oracle = Scripts.validator @Oracling
    ($$(PlutusTx.compile [| mkOracleValidator |]) `PlutusTx.applyCode` PlutusTx.
    ↪ liftCode oracle)
    $$(PlutusTx.compile [| wrap |])
where
    wrap = Scripts.wrapValidator @Integer @OracleRedeemer

oracleValidator :: Oracle -> Validator
oracleValidator = Scripts.validatorScript . oracleInst

oracleAddress :: Oracle -> Ledger.Address
oracleAddress = scriptAddress . oracleValidator

```

And this concludes the on-chain part of the oracle code.

## 6.2.2 Off-chain

We also create some off-chain code, namely to start the oracle, and to update it. However, we don't write off-chain code to *use* the oracle. That is not the responsibility of the author of this contract. That will be the responsibility of the person that wants to use the oracle - they will write the code to create the transaction with the *use* redeemer. This is the first time that we have seen the situation where we have some on-chain code that is not paired with some off-chain code.

## Starting the Oracle

To start the oracle, we need some parameters.

```
data OracleParams = OracleParams
  { opFees    :: !Integer
  , opSymbol  :: !CurrencySymbol
  , opToken   :: !TokenName
  } deriving (Show, Generic, FromJSON, ToJSON)
```

The *opFees* parameter represents the number of lovelace that will be charged to use the oracle.

The *opSymbol* and *opToken* parameters represent the token against which we are providing the Ada exchange rate, in this case a USD token.

First we create a *startOracle* function, whose responsibility is to mint the NFT that will be used to identify the oracle UTxO. The *startOracle* function will not provide an initial value for the oracle, this will be handled by the *updateOracle* function. The reason for this is that, if we provided an initial value, it may be outdated by the time the NFT is minted.

We could have used the same code for minting the NFT as we used in lecture 5. This would have worked perfectly well.

However, this is a currency module provided in *plutus-use-cases* that provides a *forgeContract* function that allows us to mint NFTs.

Here is the type of the *forgeContract* function shown in the REPL.

```
Prelude Week06.Oracle.Core> :t Plutus.Contracts.Currency.forgeContract
Plutus.Contracts.Currency.forgeContract
:: (row-types-1.0.1.0:Data.Row.Internal.AllUniqueLabels
   (Plutus.Contract.Schema.Input s),
   row-types-1.0.1.0:Data.Row.Internal.AllUniqueLabels
   (Plutus.Contract.Schema.Output s),
   Plutus.Contracts.Currency.AsCurrencyError e,
   (Plutus.Contract.Schema.Input s
    row-types-1.0.1.0:Data.Row.Internal...! "tx-confirmation")
  ~ Plutus.Contract.Effects.AwaitTxConfirmed.TxConfirmed,
   (Plutus.Contract.Schema.Input s
    row-types-1.0.1.0:Data.Row.Internal...! "tx")
  ~ Plutus.Contract.Effects.WriteTx.WriteTxResponse,
   (Plutus.Contract.Schema.Output s
    row-types-1.0.1.0:Data.Row.Internal...! "tx")
  ~ Ledger.Constraints.OffChain.UnbalancedTx,
   (Plutus.Contract.Schema.Output s
    row-types-1.0.1.0:Data.Row.Internal...! "tx-confirmation")
  ~ Plutus.V1.Ledger.TxId.TxId) =>
Plutus.V1.Ledger.Crypto.PubKeyHash
-> [(Plutus.V1.Ledger.Value.TokenName, Integer)]
-> Plutus.Contract.Types.Contract
   w s e Plutus.Contracts.Currency.OneShotCurrency
```

The important part starts towards the end, where the first parameter - of type *PubKeyHash* - is defined. This is the hash of the public key of the recipient of the NFT.

The *forgeContract* function provides more general functionality than our previous NFT contract. It allows is to generate multiple NFTs in one go. It will create a currency symbol that can only be used one, similar to our NFT from last time, so there can only be one minting transaction. But for the one currency symbol, you can mint various tokens in the same

transaction, with various token names and in various quantities. The second parameter allows us to define these token names and quantities.

And it gives us a *Contract* that returns a value of the *OneShotCurrency* type. This type is specific to the currency and it doesn't really matter to us what it is. All that matters for us is that we can get the currency symbol out of it again.

There is one slight problem. This is not compatible with what we want. We want this types

```
Contract w s Text Oracle
```

An arbitrary writer type (because we don't make use of it), an arbitrary schema (as long as we have *BlockchainActions* available), *Text* error messages and a return type of *Oracle*.

The problem is that the *Contract* returned by *forgeContract* doesn't allow *Text* error messages. You can see this in the verbose output from the REPL - there is a constraint on the *e* parameter.

```
Plutus.Contracts.Currency.AsCurrencyError e,
```

Unfortunately *Text* doesn't implement *AsCurrencyError*.

Luckily there is a function that can help

```
Plutus.Contract.mapError
:: (e -> e')
-> Plutus.Contract.Types.Contract w s e a
-> Plutus.Contract.Types.Contract w s e' a
```

Given a *Contract*, it allows us to create a new *Contract* with a new type of error message. That is provided we give a function that converts from the first error type to the second error type.

So, let's look at the *startOracle* function.

```
startOracle :: forall w s. HasBlockchainActions s => OracleParams -> Contract w s Text_
  ↳ Oracle
startOracle op = do
  pkh <- pubKeyHash <$> Contract.ownPubKey
  osc <- mapError (pack . show) (forgeContract pkh [(oracleTokenName, 1)] :: Contract_
  ↳ w s CurrencyError OneShotCurrency)
  let cs      = Currency.currencySymbol osc
      oracle = Oracle
        { oSymbol   = cs
        , oOperator = pkh
        , oFee      = opFees op
        , oAsset    = AssetClass (opSymbol op, opToken op)
        }
  logInfo @String $ "started oracle " ++ show oracle
  return oracle
```

Here we see the error conversion function is provided as *pack . show*. The *show* function converts the error to a *String* and the *pack* function converts a *String* to a *Data.Text* type.

At this point, *osc* holds the *OneShotCurrency*, and we can then use the *currencySymbol* function to get the currency symbol as *cs*.

The *currencySymbol* function has type

```
currencySymbol
:: OneShotCurrency -> Plutus.V1.Ledger.Value.CurrencySymbol
```

And is used accordingly

```
let cs = Currency.currencySymbol osc
```

Now we have minted our NFT and it has currency symbol *cs*. And now we can construct our *Oracle* parameter value.

```
oracle = Oracle
  { oSymbol   = cs
  , oOperator = pkh
  , oFee      = opFees op
  , oAsset    = AssetClass (opSymbol op, opToken op)
  }
```

The reason that *opSymbol* and *opToken* are defined separately in the *OracleParams* type *op* is just that this makes is easier when we are using the playground.

### Updating the Oracle

The *updateOracle* function is more complicated.

This function has to deal with two cases. Namely, the case where we have a value that we wish to update, and the case where we have just started the oracle and we want to create a value for the very first time.

It takes our oracle parameters and also the *Integer* value that we wish to have the oracle hold.

First we create a helper function *findOracle*.

```
findOracle :: forall w s. HasBlockchainActions s => Oracle -> Contract w s Text (Maybe_
  ↳ (TxOutRef, TxOutTx, Integer))
findOracle oracle = do
  utxos <- Map.filter f <$> utxoAt (oracleAddress oracle)
  return $ case Map.toList utxos of
    [(oref, o)] -> do
      x <- oracleValue (txOutTxOut o) $ \dh -> Map.lookup dh $ txData $ txOutTxTx o
      return (oref, o, x)
    -           -> Nothing
  where
    f :: TxOutTx -> Bool
    f o = assetClassValueOf (txOutValue $ txOutTxOut o) (oracleAsset oracle) == 1
```

The purpose of *findOracle* is to look up the existing oracle UTxO. This can fail because the oracle might not be there. This will happen if we have just started the oracles and have not yet created a UTxO with the oracle value. But, if we find it, we return a triple containing the UTxO identifier (TxOutRef), the UTxO itself, which contains all the data (TxOutTx) and the oracle value (the current exchange rate held by the oracle). The *Integer* containing the oracle value is encoded also in the TxOutTx value, but we add it to the triple to make it easier to work with.

The first thing we do is to get all the UTxOs sitting at this address. But only one of these will be the one we are looking for - the one that contains the NFT.

We do this by using the *Map.filter* function which takes a function as a parameter which, in this case, returns True for the UTxO where the NFT is present.

```
utxos <- Map.filter f <$> utxoAt (oracleAddress oracle)
...
where
```

(continues on next page)

(continued from previous page)

```
f :: TxOutTx -> Bool
f o = assetClassValueOf (txOutValue $ txOutTxOut o) (oracleAsset oracle) == 1
```

We will end up with a map in *utxos* which is either empty or contains one item. Now, we distinguish between these two cases.

```
return $ case Map.toList utxos of
  [(oref, o)] -> do
    x <- oracleValue (txOutTxOut o) $ \dh -> Map.lookup dh $ txData $ txOutTxTx o
    return (oref, o, x)
  _           -> Nothing
```

We convert the map to a list of tuples representing key value pairs of transaction ids and the transactions themselves.

For the case where there is no element, we use the `_` case to represent all other cases. This could only ever be the empty list, but the compiler doesn't know that.

If, however, we have found the UTxO, then, as we already have its id and transaction, we just need to find its *Integer* value. This part could still go wrong. Even though we have found the correct UTxO, there could be some corrupt data in it for whatever reason.

We use the *oracleValue* function that we used also in validation. This function takes a *TxOut* parameter followed by a second parameter is a function, which, given a datum hash will return the associated datum.

In the off-chain code, we can use the following function parameter

```
\dh -> Map.lookup dh $ txData $ txOutTxTx o
```

Here, *txData* is a field of the transaction and it is a map from datum hashes to datums. We get the transaction from *txOutTxTx o*.

If this all succeeds, when will return the triple (oref, o, x), where x is the *Integer* value of the oracle.

Now that we have written the *findOracle* function we can look at the *updateOracle* function.

```
updateOracle :: forall w s. HasBlockchainActions s => Oracle -> Integer -> Contract w s
  ↳ Text ()
updateOracle oracle x = do
  m <- findOracle oracle
  let c = Constraints.mustPayToTheScript x $ assetClassValue (oracleAsset oracle) 1
  case m of
    Nothing -> do
      ledgerTx <- submitTxConstraints (oracleInst oracle) c
      awaitTxConfirmed $ txId ledgerTx
      logInfo @String $ "set initial oracle value to " ++ show x
    Just (oref, o, _) -> do
      let lookups = Constraints.unspentOutputs (Map.singleton oref o) <>
                    Constraints.scriptInstanceLookups (oracleInst oracle) <>
                    Constraints.otherScript (oracleValidator oracle)
          tx      = c <> Constraints.mustSpendScriptOutput oref (Redeemer $
  ↳ PlutusTx.toData Update)
      ledgerTx <- submitTxConstraintsWith @Oracling lookups tx
      awaitTxConfirmed $ txId ledgerTx
      logInfo @String $ "updated oracle value to " ++ show x
```

After the *findOracle* line there is a helper function definition, as we will need this constraint twice.



```
let c = Constraints.mustPayToTheScript x $ assetClassValue (oracleAsset oracle) 1
```

After looking for the oracle, there are two possibilities - either we found it or we did not.

If we didn't find it, then we have started the oracle but we have not yet provided an initial value. This is the first case. And in this case, all we have to do is to submit a transaction that produces the first value for the oracle.

```
ledgerTx <- submitTxConstraints (oracleInst oracle) c
awaitTxConfirmed $ txId ledgerTx
logInfo @String $ "set initial oracle value to " ++ show x
```

Here is the first usage of the *c* helper function. It provides the constraint *mustPayToTheScript* which ensures that the transaction will have an output that pays to a script address. As arguments it takes the datum *x* and the NFT. The script that it must pay to is always the script that is in focus - here it is the first parameter to *submitTxConstraints* - (*oracleInst oracle*).

We then wait for confirmation and write a log message. And this is all we need to do for this case.

In the other case, where we already have a value, we need to reference the UTxO parts, but we don't care about the current datum, as we are going to update it anyway.

```
Just (oref, o, _) -> do
```

Now it gets a bit more complicated, because now we need two conditions.

The first constraint is the same as in the other case - the constraint referenced by the helper function *c*. But there is now an extra constraint that we must also consume the existing UTxO.

```
tx = c <> Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toData Update)
```

The *mustSpendScriptOutput* function is basically the opposite of *mustPayToTheScript*. It creates an input to this script address. As parameters it takes the reference to the UTxO we want to consume, and it takes a *Redeemer*. In this case the *Redeemer* is *Update* and it is converted to the Plutus *Data* type.

In order for this to work we need to provide some lookups.

In order to find the output *oref* that it wants to spend, we must use the *unspentOutputs* lookup, and in this case, we just provide the lookup with one UTxO.

```
Constraints.unspentOutputs (Map.singleton oref o)
```

Then we must provide the script instances. We need to do this twice, once for the input side, and once for the output side. For this, we provide the oracle instance and the oracle validator.

```
Constraints.scriptInstanceLookups (oracleInst oracle) <>
Constraints.otherScript (oracleValidator oracle)
```

We didn't need to provide the *scriptInstanceLookups* lookup in the first case, as we were able to pass *oracleInst oracle* to the *submitTxConstraints* function. However, with the *submitTxConstraintsWith* function, we don't have that option.

When submitting the transaction, we need to give the compiler a little nudge to let it know the script we are talking about - so that it knows, for example, what The Script is in *mustPayToTheScript*. For this we reference the *Oracling* type.

```
ledgerTx <- submitTxConstraintsWith @Oracling lookups tx
```

Hopefully now we have a valid transaction that gets submitted, and then we wait for it to be confirmed, and write some logging information.



```
awaitTxConfirmed $ txId ledgerTx
logInfo @String $ "updated oracle value to " ++ show x
```

Remember, we talked about fee collecting earlier. This will happen automatically. The function *submitTxConstraintsWith* will send the fees to our own wallet. It does this because there is an imbalance between the value attached to the input, which includes the fees and the NFT, and the value we have said must be paid to the script, which is just the NFT.

This process will also automatically create an extra input from our own input to pay the transaction fees for executing the transaction.

Lastly, we provide a function that combines these two operations, *startOracle* and *updateOracle* into one contract. This will make it possible to use in the playground and the *EmulatorTrace* monad, as well as in the PAB.

```
type OracleSchema = BlockchainActions .\ Endpoint "update" Integer

runOracle :: OracleParams -> Contract (Last Oracle) OracleSchema Text ()
runOracle op = do
  oracle <- startOracle op
  tell $ Last $ Just oracle
  go oracle
  where
    go :: Oracle -> Contract (Last Oracle) OracleSchema Text a
    go oracle = do
      x <- endpoint @"update"
      updateOracle oracle x
      go oracle
```

The function *runOracle* first starts the oracle. Then, for reasons that will become clear later, we use *tell* to write the oracle parameter. We need to be able to communicate the parameter value of the oracle to the outside world, so that people can use the oracle. We will not know until runtime the currency symbol that will be used for the NFT, so we don't know the value of the oracle parameter yet.

Remember that *tell* expects a *Monoid* type. The typical example is a list of strings that get concatenated to one list of all log messages.

But it doesn't have to be lists. In *Data.Monoid* we have this *Last Monoid*.

```
Prelude Week06.Oracle.Core> import Data.Monoid (Last (..))
Prelude Data.Monoid Week06.Oracle.Core> :i Last
type Last :: * -> *
newtype Last a = Last {getLast :: Maybe a}
  -- Defined in 'Data.Monoid'
instance Applicative Last -- Defined in 'Data.Monoid'
instance Eq a => Eq (Last a) -- Defined in 'Data.Monoid'
instance Functor Last -- Defined in 'Data.Monoid'
instance Monad Last -- Defined in 'Data.Monoid'
instance Monoid (Last a) -- Defined in 'Data.Monoid'
instance Ord a => Ord (Last a) -- Defined in 'Data.Monoid'
instance Semigroup (Last a) -- Defined in 'Data.Monoid'
instance Show a => Show (Last a) -- Defined in 'Data.Monoid'
instance Read a => Read (Last a) -- Defined in 'Data.Monoid'
instance Traversable Last -- Defined in 'Data.Traversable'
instance Foldable Last -- Defined in 'Data.Foldable'
```

We see that it is just a *newtype* wrapper around *Maybe*. The point is to provide a specific *Monoid* instance. The idea, as the name suggests, is that it is a monoid operation that always remembers the last *Just* value. For example:

```
Prelude Data.Monoid Week06.Oracle.Core> Last (Just 'x') <> Last (Just 'y')
Last {getLast = Just 'y'}
```

However, if the second *Last* is a *Nothing*, it will return the first one.

```
Prelude Data.Monoid Week06.Oracle.Core> Last (Just 'x') <> Last Nothing
Last {getLast = Just 'x'}
```

If both are *Nothing*, it will be *Nothing*.

*Last* is very useful because it allows us to keep the current state. The value of the log will basically be the last *Just* we told.

In this contract we will only do that once. In the beginning it will be *Last Nothing*. Then we mint the NFT, and then, when we get the oracle value in *runOracle*, and then *tell* it, it will always have that value. If other contracts from the outside query the state, they will always get the *Just oracle*, so they will be able to discover the value of the oracle.

So, next in *runOracle*, we call the helper function *go*. What this does is to block at the update endpoint. As soon as someone provides an *Integer* as the new value, it will call the *updateOracle* function with the new value, and then just loop to go again.

In summary, *runOracle* starts the oracle, *tells* the oracle, then loops to allow others to update the oracle.

And that concludes the code for the oracle itself. What is now missing is an example, a contract that actually uses the oracle - a swap contract. And also using the Plutus Application Backend to run this code in the real world or, in our case, in a simulated blockchain.

## 6.3 Swap Validation

Our example swap contract can be found in

```
module Week06.Oracle.swap
```

The purpose of this contract is for someone to be able to deposit ADA and exchange it for a token, in our case a token that we will call USDT for US Dollar Token.

The idea is that the price, the amount of USDT that will be required to be paid for the ADA, will be determined by the value of the oracle. Remember that we are using an *Integer* to reflect the exchange rate, with a value of one million being equal to one USDT.

We'll start with a helper function called *price*, which, given a number of lovelace and the exchange rate, returns the USDT price.

```
price :: Integer -> Integer -> Integer
price lovelace exchangeRate = (lovelace * exchangeRate) `divide` 1000000
```

The next helper function, *lovelaces*, combines to functions from the Plutus libraries to extract a number of lovelace from a *Value* type.

```
lovelaces :: Value -> Integer
lovelaces = Ada.getLovelace . Ada.fromValue
```

Now we will write *mkSwapValidator*. This is a parameterized validator with two parameters.

The first parameter is the oracle that we are using. To use this, we import the oracle module.

```
import Week06.Oracle.core
```

The second parameter is the address of the oracle. Normally, given the oracle, we would be able to compute the address from it. In the core module we saw a function `oracleAddress` which does this for us. But this is a function that we can't use in the validator, because it can't be compiled to Plutus script. So, here, we explicitly hand the address to the validator.

For the datum, we use the public key hash of the seller. We don't use a redeemer, so we give it a type of Unit.

We recall from the diagram, the swap transaction should have three inputs and three outputs.

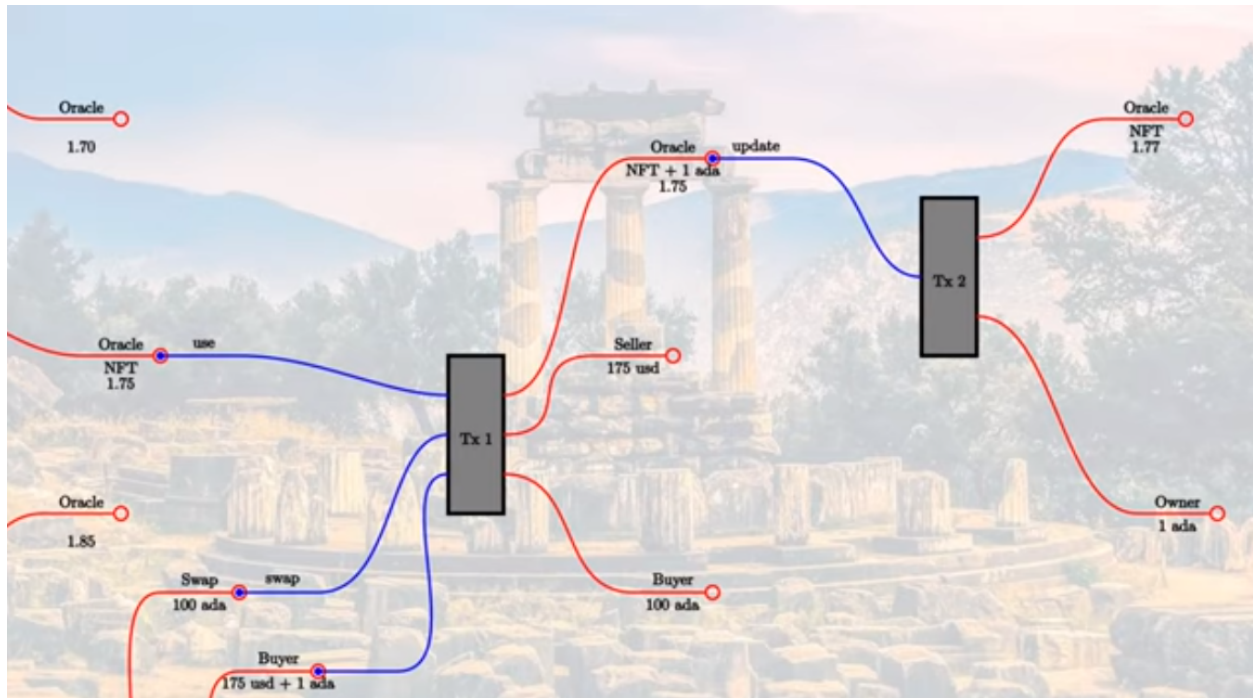


Table 1: Swap Transaction Inputs and Outputs

Inputs	Outputs
The oracle, to check the current exchange rate	The oracle, which we don't need to look at in the swap validation
The swap UTxO that holds the lovelace	The tokens for the seller
The source of the buyer's funds	The lovelace for the buyer

Note that we don't need to worry about the oracle as an output. The oracle validator takes care of ensuring that the value is not changed and that the fees are added.

We also want to support the second use case, the case where the seller can retrieve the ADA tokens in the case that they no longer want to do the swap. If we don't support this case, the ADA could be locked there forever, if nobody ever decides to make the swap.

This second case is the condition we check in the validator. If the seller themselves signs the transaction, there are no further constraints - we don't need to check the oracle or anything else - the seller can just get back their lovelace.

```
mkSwapValidator :: Oracle -> Address -> PubKeyHash -> () -> ScriptContext -> Bool
mkSwapValidator oracle addr pkh () ctx =
```

(continues on next page)

(continued from previous page)

```
txSignedBy info pkh ||  
...
```

The more interesting case is the second one, where we check two conditions.

Firstly, there must be two inputs - the oracle and the swap UTxO. All additional inputs (the buyer's funds) must be public key inputs. This is because we don't want to worry about interference with other smart contracts.

Secondly, we want to check that the seller gets paid.

```
(traceIfFalse "expected exactly two script inputs" hasTwoScriptInputs &&  
  traceIfFalse "price not paid" sellerPaid)
```

Now, we have our helper function definitions.

First, the usual.

```
info :: TxInfo  
info = scriptContextTxInfo ctx
```

Then, we have *oracleInput* to get the UTxO from the oracle.

```
oracleInput :: TxOut  
oracleInput =  
  let  
    ins = [ o  
            | i <- txInfoInputs info  
            , let o = txInInfoResolved i  
            , txOutAddress o == addr  
            ]  
  in  
    case ins of  
      [o] -> o  
      _    -> traceError "expected exactly one oracle input"
```

We do this by getting a list of all the inputs. For this we use list comprehension, which allows us to draw from other lists using a filter. In this case we draw from the list from *txInfoInputs info*, which is a list of *TxInfo*. We use the *txInInfoResolved* function to look at each element as a *TxOut* type, which we then compare with the *addr* parameter. The resulting list will either be empty, or will have the *TxOut* that matches the oracle UTxO.

We then check that there is exactly one element in the resulting list, and, if there is, we return it. We don't return the list, just the *TxOut*.

This has now given us the oracle output that we are consuming as an input.

Now, we want to check the actual exchange rate. For that, we use the *oracleValue* function that we defined in the core module. Here again, it may succeed, or it may fail. If it succeeds we return the value.

```
oracleValue' = case oracleValue oracleInput (`findDatum` info) of  
  Nothing -> traceError "oracle value not found"  
  Just x   -> x
```

We do not need to check whether the oracle contains the NFT. Due to the way validation works for the oracle, we know that it is present.

Now, let's look at the *hasTwoScriptInputs* helper function.

```
hasTwoScriptInputs :: Bool
hasTwoScriptInputs =
  let
    xs = filter (isJust . toValidatorHash . txOutAddress . txInInfoResolved) $
    ↪ txInfoInputs info
  in
    length xs == 2
```

First, we filter, using the composite function

```
(isJust . toValidatorHash . txOutAddress . txInInfoResolved)
```

Reading right to left, we get the UTxO from the input, then we get the address for this UTxO, then we get the validator hash for that address. Then, finally, we check if it is a script output, by seeing if it is a *Just*. If it is a *Nothing*, then this would show that it is a public key, not a script address.

We then use this composite function as a filter against the list of *TxInInfos*. And then we check that the length of the resulting list is exactly two.

Going back to our validation conditions, we now have to deal with checking that the seller is getting paid. So let's write the *sellerPaid* helper function that we referenced.

For this we will use another helper function to determine the required price.

```
minPrice :: Integer
minPrice =
  let
    lovelaceIn = case findOwnInput ctx of
      Nothing -> traceError "own input not found"
      Just i   -> lovelaces $ txOutValue $ txInInfoResolved i
  in
    price lovelaceIn oracleValue'
```

First we check that we have an input, and if so, we extract the number of lovelaces and assign that number to *lovelaceIn*. Then, we use the *price* helper function to determine the price in USD tokens.

Now, we can define the *sellerPaid* helper function.

```
sellerPaid :: Bool
sellerPaid =
  let
    pricePaid :: Integer
    pricePaid = assetClassValueOf (valuePaidTo info pkh) (oAsset oracle)
  in
    pricePaid >= minPrice
```

The function *valuePaidTo* is from the Plutus libraries. Given *info* and a public key hash, it will add up all the values of all the public key outputs that go to this address. We then use the *assetClassValueOf* function to check the component of the value that is in USD token, and the check that we have at least as many as we require.

That's the end of the main part of the code for the swap validator. We just have our normal boiler plate to write.

```
data Swapping
instance Scripts.ScriptType Swapping where
  type instance DatumType Swapping = PubKeyHash
  type instance RedeemerType Swapping = ()
```

(continues on next page)

(continued from previous page)

```

swapInst :: Oracle -> Scripts.ScriptInstance Swapping
swapInst oracle = Scripts.validator @Swapping
    ($$(PlutusTx.compile [| mkSwapValidator |]))
    `PlutusTx.applyCode` PlutusTx.liftCode oracle
    `PlutusTx.applyCode` PlutusTx.liftCode (oracleAddress oracle))
    $$(PlutusTx.compile [| wrap |])
where
    wrap = Scripts.wrapValidator @PubKeyHash @()

swapValidator :: Oracle -> Validator
swapValidator = Scripts.validatorScript . swapInst

swapAddress :: Oracle -> Ledger.Address
swapAddress = scriptAddress . swapValidator

```

Note that in the *swapInst* function, where we use template haskell to generate the Plutus validator from the *mkSwapValidator* function, we do not need to pass in the oracle address as a parameter. This is because we will compute this inside the function. Remember that we can't use the *oracleAddress* function inside the Plutus validator.

Now to define some contracts.

### 6.3.1 offerSwap

First *offerSwap*. This is for a seller who wants to offer a certain number of lovelace for exchange.

```

offerSwap :: forall w s. HasBlockchainActions s => Oracle -> Integer -> Contract w s
    ↳ Text ()
offerSwap oracle amt = do
    pkh <- pubKeyHash <$> Contract.ownPubKey
    let tx = Constraints.mustPayToTheScript pkh $ Ada.lovelaceValueOf amt
    ledgerTx <- submitTxConstraints (swapInst oracle) tx
    awaitTxConfirmed $ txId ledgerTx
    logInfo @String $ "offered " ++ show amt ++ " lovelace for swap"

```

### 6.3.2 findSwaps

Next, a helper function that will find all swaps that satisfy a given predicate. It takes an oracle plus a predicate based on public key hashes, and returns a list of triples of the UTxOs that satisfy the predicate.

```

findSwaps :: HasBlockchainActions s => Oracle -> (PubKeyHash -> Bool) -> Contract w s
    ↳ Text [(TxOutRef, TxOutTx, PubKeyHash)]
findSwaps oracle p = do
    utxos <- utxoAt $ swapAddress oracle
    return $ mapMaybe g $ Map.toList utxos
where
    f :: TxOutTx -> Maybe PubKeyHash
    f o = do
        dh <- txOutDatumHash $ txOutTxOut o
        (Datum d) <- Map.lookup dh $ txData $ txOutTxTx o
        PlutusTx.fromData d

```

(continues on next page)

(continued from previous page)

```

g :: (TxOutRef, TxOutTx) -> Maybe (TxOutRef, TxOutTx, PubKeyHash)
g (oref, o) = do
  pkh <- f o
  guard $ p pkh
  return (oref, o, pkh)

```

First, we get a list of all the UTxOs sitting at the swap contract address. We then apply *mapMaybe* to this list.

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

This function will apply the  $(a \rightarrow \text{Maybe } b)$  function to each element in a list of *as* and creates a list of *Maybe bs*, which could contain a mixture of *Justs* and *Nothings*. It then throws away the *Nothings* and returns the values contained in the *Justs*.

To clarify this, imagine we have a function that returns as *Just* for even numbers and a *Nothing* for odd numbers.

```
f (n :: Int) = if even n then Just (div n 2) else Nothing
```

We can use this as the first parameter to map Maybe

```

Prelude Week06.Oracle.Core> import Data.Maybe
Prelude Data.Maybe Week06.Oracle.Core> mapMaybe f [2, 4, 10, 11, 13, 100]
[1,2,5,50]

```

We use the *mapMaybe* and the function *g* to filter the list of UTxOs.

```

g :: (TxOutRef, TxOutTx) -> Maybe (TxOutRef, TxOutTx, PubKeyHash)
g (oref, o) = do
  pkh <- f o
  guard $ p pkh
  return (oref, o, pkh)

```

This function takes a key value pair representing the UTxO and returns a *Maybe* triple containing the items from the pair alongside a *PubKeyHash*.

Function *g* is inside the *Maybe* monad and makes use of function *f*, which is also inside the *Maybe* monad. Function *f* gets the public key hash from a UTxO, if it exists. After this, function *g* uses the *guard* function with the predicate function *p* that we passed in as an argument.

The *guard* function is available in some monads, and the *Maybe* monad is one of them. It takes a boolean as a parameter, and, if the boolean is false, the computation fails. In this case, failure means returning *Nothing*. If it is true, it just continues. In this case, that means returning the *Just* of the triple containing the public key hash.

We will see how we use the *findSwaps* function in a moment.



### 6.3.3 retrieveSwaps

The *retrieveSwaps* contract is for the seller if they want to change their mind and get their Ada back.

Here is where we use the *findSwaps* function. We use it with  $(== \text{pkh})$  as the predicate, meaning that we want only those UTxOs sitting at the swap address that belong to the operator.

```
retrieveSwaps :: HasBlockchainActions s => Oracle -> Contract w s Text ()
retrieveSwaps oracle = do
  pkh <- pubKeyHash <$> ownPubKey
  xs <- findSwaps oracle (== pkh)
  case xs of
    [] -> logInfo @String "no swaps found"
    _   -> do
      let lookups = Constraints.unspentOutputs (Map.fromList [(oref, o) | (oref, o,
      ↪ _) <- xs]) <>
          Constraints.otherScript (swapValidator oracle)
      tx      = mconcat [Constraints.mustSpendScriptOutput oref $ Redeemer $ ↪
      ↪ PlutusTx.toData () | (oref, _, _) <- xs]
      ledgerTx <- submitTxConstraintsWith @Swapping lookups tx
      awaitTxConfirmed $ txId ledgerTx
      logInfo @String $ "retrieved " ++ show (length xs) ++ " swap(s)"
```

If there are none, then there is nothing to do. If there are some, we construct a transaction that retrieves all of them.

To do that, we create a list of *mustSpendScriptOutput* constraints.

```
tx = mconcat [Constraints.mustSpendScriptOutput oref $ Redeemer $ PlutusTx.toData () | ↪
↪ (oref, _, _) <- xs]
```

The line looks intimidating, but it is just extracting a list of *orefs* from the *xs* list and using it to construct a constraint for each of them, using *Unit* as the *Redeemer* type. The function *mconcat* applies the *Semigroup* operator *<>* throughout the list in order to combine them.

As lookups, we must provide all the UTxOs and the swap validator.

We have the list of UTxOs in *xs* and we use list comprehension to turn this list into a list of pairs, and we then use *Map.fromList* to turn those pairs into a map, to which we then apply the *unspentOutputs* constraint.

### 6.3.4 useSwaps

And now the most interesting one, *useSwaps*. This is where we actually use the oracle.

```
useSwap :: forall w s. HasBlockchainActions s => Oracle -> Contract w s Text ()
useSwap oracle = do
  funds <- ownFunds
  let amt = assetClassValueOf funds $ oAsset oracle
  logInfo @String $ "available assets: " ++ show amt

  m <- findOracle oracle
  case m of
    Nothing      -> logInfo @String "oracle not found"
    Just (oref, o, x) -> do
      logInfo @String $ "found oracle, exchange rate " ++ show x
      pkh <- pubKeyHash <$> Contract.ownPubKey
```

(continues on next page)



(continued from previous page)

```

swaps <- findSwaps oracle (/= pkh)
case find (f amt x) swaps of
  Nothing          -> logInfo @String "no suitable swap found"
  Just (oref', o', pkh') -> do
    let v          = txOutValue (txOutTxOut o) <> lovelaceValueOf (oFee_
->oracle)
        p          = assetClassValue (oAsset oracle) $ price (lovelaces $_
->txOutValue $ txOutTxOut o') x
        lookups    = Constraints.otherScript (swapValidator oracle)
        <>
        Constraints.otherScript (oracleValidator oracle)
        <>
        Constraints.unspentOutputs (Map.fromList [(oref, o),_
->(oref', o')])
        tx         = Constraints.mustSpendScriptOutput oref (Redeemer $_
->PlutusTx.toData Use) <>
        Constraints.mustSpendScriptOutput oref' (Redeemer $_
->PlutusTx.toData ()) <>
        Constraints.mustPayToOtherScript
          (validatorHash $ oracleValidator oracle)
          (Datum $ PlutusTx.toData x)
        v
        <>
        Constraints.mustPayToPubKey pkh' p
    ledgerTx <- submitTxConstraintsWith @Swapping lookups tx
    awaitTxConfirmed $ txId ledgerTx
    logInfo @String $ "made swap with price " ++ show (Value.
->flattenValue p)
  where
    getPrice :: Integer -> TxOutTx -> Integer
    getPrice x o = price (lovelaces $ txOutValue $ txOutTxOut o) x

    f :: Integer -> Integer -> (TxOutRef, TxOutTx, PubKeyHash) -> Bool
    f amt x (_, o, _) = getPrice x o <= amt

```

First, we use the *ownFunds* function. This is defined in a separate module that we will get to in a bit. All it does is to add up all the money in our own wallet and returns a *Value*. We then find out how many USD Tokens we have.

```

funds <- ownFunds
let amt = assetClassValueOf funds $ oAsset oracle
logInfo @String $ "available assets: " ++ show amt

```

The *findOracle* function is defined in the *Oracle.Core* module from earlier. You will recall that it finds us the oracle UTxO that contains the oracle value.

```
m <- findOracle oracle
```

If we don't find the oracle, we will just log a message to that effect.

```

case m of
  Nothing          -> logInfo @String "oracle not found"

```

If we do find it, we will log a message with the current exchange rate.

```
Just (oref, o, x) -> do
  logInfo @String $ "found oracle, exchange rate " ++ show x
```

Next, we check our own public key and check for all available swaps where we are *not* the owner.

```
pkh <- pubKeyHash <$> Contract.ownPubKey
swaps <- findSwaps oracle (/= pkh)
```

Then, we use a function *find* which is from the Haskell prelude, in module *Data.List*. The *find* function takes a predicate and a list and *Maybe* returns one element of that list that satisfies the predicate.

The function used in the predicate is defined as the helper function *f*.

```
where
  getPrice :: Integer -> TxOutTx -> Integer
  getPrice x o = price (lovelaces $ txOutValue $ txOutTxOut o) x

  f :: Integer -> Integer -> (TxOutRef, TxOutTx, PubKeyHash) -> Bool
  f amt x (_, o, _) = getPrice x o <= amt
```

We give it an amount, the current exchange rate, and a UTxO triple. The function determines if there is a swap that is cheaper to or equal to the amount parameter.

Now, we have searched for a swap that we can afford. If we don't find one, we log a message saying so.

```
case find (f amt x) swaps of
  Nothing -> logInfo @String "no suitable swap found"
```

If we *do* find one, we just take the first one. This isn't very realistic, of course. In a real-world example we would probably specify the exact amount we want to swap. Here, we are just keeping it simple as we are focussed on oracles rather than swapping.

So, now we construct a transaction.

```
let v = txOutValue (txOutTxOut o) <> lovelaceValueOf (oFee oracle)
```

This is the output for the oracle. It is the same as the input, including any fees that have accumulated there, plus the fee in lovelace that we need to pay.

We then create a *Value* representing the USD Tokens that we need to pay.

```
p = assetClassValue (oAsset oracle) $ price (lovelaces $ txOutValue $ txOutTxOut o') x
```

Now, let's look at the constraints.

The first constraint is that we must consume the oracle as an input. And here we see the first use of the *Use* redeemer. We never used this redeemer in the oracle core itself, as the oracle provider is only responsible for updating values, which uses the *Update* redeemer.

```
Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toData Use)
```

The second constraint is to consume the swap input, which just uses a *Unit* redeemer.

```
Constraints.mustSpendScriptOutput oref' (Redeemer $ PlutusTx.toData ())
```

The third constraint is to pay the oracle.

```

Constraints.mustPayToOtherScript
  (validatorHash $ oracleValidator oracle)
  (Datum $ PlutusTx.toData x)
  v

```

Here we use *mustPayToOtherScript*, specifying the oracle script, because now we have two scripts in play - the oracle script and the swap script. As *Datum* we use the exist datum - we mustn't change it - and as *Value* we use the value *v* that we computed earlier.

The final constraint is that we must pay the seller of the lovelace - and the payment is the price *p* that we calculated before.

```

Constraints.mustPayToPubKey pkh' p

```

For lookups, we must provide the validators of the oracle and swap contracts, and we must provide the two UTxOs that we want to consume.

```

lookups = Constraints.otherScript (swapValidator oracle)      <>
          Constraints.otherScript (oracleValidator oracle)    <>
          Constraints.unspentOutputs (Map.fromList [(oref, o), (oref', o')])

```

Now, the usual - we submit it, wait for confirmation, then log a message.

```

ledgerTx <- submitTxConstraintsWith @Swapping lookups tx
awaitTxConfirmed $ txId ledgerTx
logInfo @String $ "made swap with price " ++ show (Value.flattenValue p)

```

### 6.3.5 Contract bundle

That defines the raw contracts. Now, we provide a bundle that contains all of them.

First, we define, as always, a schema, which defines the endpoints.

```

type SwapSchema =
  BlockchainActions
    .\ Endpoint "offer" Integer
    .\ Endpoint "retrieve" ()
    .\ Endpoint "use" ()
    .\ Endpoint "funds" ()

```

Next, we see the *select* operator. This use of this operator will cause our code to wait until one of the endpoints is picked, and then executes the associated code.

The *swap* function recursively calls itself, offering again and again the same choice of endpoints.

```

swap :: Oracle -> Contract (Last Value) SwapSchema Text ()
swap oracle = (offer `select` retrieve `select` use `select` funds) >> swap oracle
  where
    offer :: Contract (Last Value) SwapSchema Text ()
    offer = h $ do
      amt <- endpoint @"offer"
      offerSwap oracle amt

    retrieve :: Contract (Last Value) SwapSchema Text ()

```

(continues on next page)

(continued from previous page)

```

retrieve = h $ do
  endpoint @"retrieve"
  retrieveSwaps oracle

use :: Contract (Last Value) SwapSchema Text ()
use = h $ do
  endpoint @"use"
  useSwap oracle

funds :: Contract (Last Value) SwapSchema Text ()
funds = h $ do
  endpoint @"funds"
  v <- ownFunds
  tell $ Last $ Just v

h :: Contract (Last Value) SwapSchema Text () -> Contract (Last Value)
  ↳ SwapSchema Text ()
h = handleError logError

```

The code for the four endpoints are wrappers for the code we have already written.

For *offer*, for example, we block until we are provided with an *amt* and then we call the *offerSwap* contract.

It is the same for the *retrieve* and *use* endpoints, except that they require no parameters.

For the *funds* endpoint it is a little different. The *ownFunds* function comes from the *Funds* module, which, as we noted earlier, we have not yet looked at. It gives us the *Value* that we own. We then *tell* this value as a way of reporting to the outside world how much we have.

The *h* in each of the endpoints is an error handler. Each of the endpoints is wrapped inside the error handler, which just logs the error, but does not halt execution.

And that concludes the swap example.

## 6.4 Funds Module

Now let's quickly look at the *Funds* module. It's a short module that provides two contracts.

The *ownFunds* function is tasked with summing up all the *Value* in our own UTxOs.

```

ownFunds :: HasBlockchainActions s => Contract w s Text Value
ownFunds = do
  pk    <- ownPubKey
  utxos <- utxoAt $ pubKeyAddress pk
  let v = mconcat $ Map.elims $ txOutValue . txOutTxOut <$> utxos
  logInfo @String $ "own funds: " ++ show (Value.flattenValue v)
  return v

```

It does this by looking up our public key, then getting all the UTxOs at that public key address. The *utxos* are then a map from UTxO references to UTxOs.

As *map* implements *Functor* we can map over the map to change the elements to something else. In this case, we change them to *Values* by applying the composite function *txOutValue . txOutTxOut*.

The *Map.elems* function ignores the keys and just gives us the values. And, as we saw before, *mconcat*, when given a *Semigroup* or *Monoid* type, will combine the list of values into one value.

So *v* is not the sum of all the values of all the UTxOs that we own. Our *ownFunds* function is a contract that has a return type of *Value*, we return *v*.

The function *ownFunds'* is a variation that, instead of returning the value, permanently tells it.

```
ownFunds' :: Contract (Last Value) BlockchainActions Text ()
ownFunds' = do
  handleError logError $ ownFunds >=> tell . Last . Just
  void $ Contract.waitNSlots 1
  ownFunds'
```

This calls the *ownFunds* function, performs a monadic bind to the composite function *tell . Last . Just* which tells the value, then it waits for a slot, and then calls itself. So, every block, it writes the value into the log.

## 6.5 Testing

We will now write code, using the *EmulatorTrace* monad, that tests the contracts we have written.

This code can be found in the following module

```
module Week06.Oracle.Test
```

First, we need to define a token that we can test with. The *assetSymbol* is an arbitrary hash, which is fine for test purposes.

```
assetSymbol :: CurrencySymbol
assetSymbol = "ff"

assetToken :: TokenName
assetToken = "USDT"
```

This time we are going to use the primed version of *runEmulatorTraceIO*, which takes two more arguments and gives more fine-grained over the emulation environment.

```
test :: IO ()
test = runEmulatorTraceIO' def emCfg myTrace
  where
    emCfg :: EmulatorConfig
    emCfg = EmulatorConfig $ Left $ Map.fromList [(Wallet i, v) | i <- [1 .. 10]]

    v :: Value
    v = Ada.lovelaceValueOf 100_000_000 <>
      Value.singleton assetSymbol assetToken 100_000_000
```

The first argument to *runEmulatorTraceIO'* determines how the various log messages are displayed. Using *def*, we have selected the default, which is the same as in the unprimed version.

The reason we are using the primed version is that we want to configure the initial distribution, and we can do that with the second argument, which here we have labelled *emCfg*.

```
emCfg :: EmulatorConfig
emCfg = EmulatorConfig $ Left $ Map.fromList [(Wallet i, v) | i <- [1 .. 10]]
```

We use this with the helper function `v` to give everyone 100 million lovelace (100 Ada) and 100 million USD Tokens to begin with.

```
v :: Value
v = Ada.lovelaceValueOf 100_000_000 <>
    Value.singleton assetSymbol assetToken 100_000_000
```

We define a helper contract, `checkOracle`, that will continually check the oracle value and log it.

```
checkOracle :: Oracle -> Contract () BlockchainActions Text a
checkOracle oracle = do
    m <- findOracle oracle
    case m of
        Nothing      -> return ()
        Just (_, _, x) -> Contract.logInfo $ "Oracle value: " ++ show x
    Contract.waitNSlots 1 >> checkOracle oracle
```

And now we can define our trace.

```
myTrace :: EmulatorTrace ()
myTrace = do
    let op = OracleParams
        { opFees = 1_000_000
        , opSymbol = assetSymbol
        , opToken = assetToken
        }

    h1 <- activateContractWallet (Wallet 1) $ runOracle op
    void $ Emulator.waitNSlots 1
    oracle <- getOracle h1

    void $ activateContractWallet (Wallet 2) $ checkOracle oracle

    callEndpoint @"update" h1 1_500_000
    void $ Emulator.waitNSlots 3

    void $ activateContractWallet (Wallet 1) ownFunds'
    void $ activateContractWallet (Wallet 3) ownFunds'
    void $ activateContractWallet (Wallet 4) ownFunds'
    void $ activateContractWallet (Wallet 5) ownFunds'

    h3 <- activateContractWallet (Wallet 3) $ swap oracle
    h4 <- activateContractWallet (Wallet 4) $ swap oracle
    h5 <- activateContractWallet (Wallet 5) $ swap oracle

    callEndpoint @"offer" h3 10_000_000
    callEndpoint @"offer" h4 20_000_000
    void $ Emulator.waitNSlots 3

    callEndpoint @"use" h5 ()
    void $ Emulator.waitNSlots 3

    callEndpoint @"update" h1 1_700_000
    void $ Emulator.waitNSlots 3
```

(continues on next page)

(continued from previous page)

```

callEndpoint @"use" h5 ()
void $ Emulator.waitNSlots 3

callEndpoint @"update" h1 1_800_000
void $ Emulator.waitNSlots 3

callEndpoint @"retrieve" h3 ()
callEndpoint @"retrieve" h4 ()
void $ Emulator.waitNSlots 3
where
getOracle :: ContractHandle (Last Oracle) OracleSchema Text -> EmulatorTrace Oracle
getOracle h = do
  l <- observableState h
  case l of
    Last Nothing      -> Emulator.waitNSlots 1 >> getOracle h
    Last (Just oracle) -> Extras.logInfo (show oracle) >> return oracle

```

This is all stuff that we have already seen. We define our oracle parameters, setting the oracle fee at 1 million lovelace, and our arbitrary asset class defined earlier.

```

let op = OracleParams
  { opFees = 1_000_000
  , opSymbol = assetSymbol
  , opToken  = assetToken
  }

```

Then, we start the oracle and wait for one slot.

```

h1 <- activateContractWallet (Wallet 1) $ runOracle op
void $ Emulator.waitNSlots 1
oracle <- getOracle h1

```

We have grabbed a handle to the oracle using the helper function defined in the *where* clause.

```

getOracle :: ContractHandle (Last Oracle) OracleSchema Text -> EmulatorTrace Oracle
getOracle h = do
  l <- observableState h
  case l of
    Last Nothing      -> Emulator.waitNSlots 1 >> getOracle h
    Last (Just oracle) -> Extras.logInfo (show oracle) >> return oracle

```

We need this because the swap contract is parameterized with the oracle value. And this is why we used *tell* in the *runOracle* function.

We use the *observableState* function to get hold of this information. If it does not exist, we wait for a slot, and try again. Otherwise, we log it for debugging purposes, and the return it.

Next, we use Wallet 2 to execute the *checkOracle* function which we saw earlier.

```

void $ activateContractWallet (Wallet 2) $ checkOracle oracle

```

We then initialize the oracle to an exchange rate of 1.5 Ada, and wait for 3 slots.

```
callEndpoint @"update" h1 1_500_000
void $ Emulator.waitNSlots 3
```

We now call the *ownFunds*' function on Wallets 1, 3, 4 and 5 to check the initial balances.

```
void $ activateContractWallet (Wallet 1) ownFunds'
void $ activateContractWallet (Wallet 3) ownFunds'
void $ activateContractWallet (Wallet 4) ownFunds'
void $ activateContractWallet (Wallet 5) ownFunds'
```

Then we start the swap contract on Wallets 3, 4, and 5.

```
h3 <- activateContractWallet (Wallet 3) $ swap oracle
h4 <- activateContractWallet (Wallet 4) $ swap oracle
h5 <- activateContractWallet (Wallet 5) $ swap oracle
```

Then we try some scenarios. First, Wallets 3 and 4 offer 10 and 20 Ada for swap respectively.

```
callEndpoint @"offer" h3 10_000_000
callEndpoint @"offer" h4 20_000_000
void $ Emulator.waitNSlots 3
```

And now Wallet 5 uses the swap. It will pick one of the two. It is not obvious which one, whichever it finds first. Remember that we only wrote code that would find the first affordable slot. It will then pay USD Token for it. The amount paid will depend on the current value of the oracle.

```
callEndpoint @"use" h5 ()
void $ Emulator.waitNSlots 3
```

Now, Wallet 1 updates the oracle value to 1.7. This also results in the accumulated fees (1 Ada) being paid to Wallet 1.

```
callEndpoint @"update" h1 1_700_000 void $ Emulator.waitNSlots 3
```

Then, Wallet 5 tries again, grabbing the remaining swap, but now paying a different USD Token price.

```
callEndpoint @"use" h5 ()
void $ Emulator.waitNSlots 3
```

We then set the oracle value to 1.8.

```
callEndpoint @"update" h1 1_800_000
void $ Emulator.waitNSlots 3
```

This will allow Wallet 1 to collect the fees. The oracle value didn't actually need to change for this to happen.

Wallets 3 and 4 now issues *retrieve* requests to get back any funds that have not been bought. This will result in no funds being returned because all the swaps have been used.

```
callEndpoint @"retrieve" h3 ()
callEndpoint @"retrieve" h4 ()
void $ Emulator.waitNSlots 3
```

And that's it. So let's run it in the REPL.



### 6.5.1 Test in the REPL

```
Prelude Week06.Oracle.Core> import Week06.Oracle.Test
Prelude Week06.Oracle.Test Week06.Oracle.Core> test
```

There will be a lot of output.

Let's look at some key parts. First, slot 3, where the oracle is created. Here we get the *oSymbol* value that we can use for everything else.

We also see in slot 3 that *getOracle* is started which will log the value of the oracle, for our information, every slot from now on.

```
Slot 00003: *** CONTRACT LOG: "started oracle Oracle {oSymbol =
↳ 6122edd57c938cda24066f434da9aee55120b4eb362d4a1bd37547ef6e4a6cbb, oOperator =
↳ 21fe31dfa154a261626bf854046fd2271b7bed4b6abe45aa58877ef47f9721b9, oFee = 1000000,
↳ oAsset = (ff,\"USDT\")}"
Slot 00003: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Sending contract state to Thread 0
Slot 00003: *** USER LOG: Oracle {oSymbol =
↳ 6122edd57c938cda24066f434da9aee55120b4eb362d4a1bd37547ef6e4a6cbb, oOperator =
↳ 21fe31dfa154a261626bf854046fd2271b7bed4b6abe45aa58877ef47f9721b9, oFee = 1000000,
↳ oAsset = (ff,\"USDT\")}"
Slot 00003: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Contract instance started
Slot 00003: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [(\"tag\",String \"update\"),(\"value\",Object
↳ (fromList [(\"unEndpointValue\",Number 1500000.0)]))])
Slot 00003: W1: TxSubmit:
↳ 93fab1c0845a5b96863a50d248fa2de68bd6702185e3de92ae0c58b869569909
Slot 00003: TxnValidate 93fab1c0845a5b96863a50d248fa2de68bd6702185e3de92ae0c58b869569909
```

And, in slot 4, we see the first value that *getOracle* finds is 1,500,000.

```
Slot 00004: *** CONTRACT LOG: "Oracle value: 1500000"
Slot 00004: *** CONTRACT LOG: "set initial oracle value to 1500000"
```

Slot 6 is the result of all the *activateContractWallet* calls. These do not necessarily appear in the same order as in the code. Notice that Wallet 1 has slightly fewer Ada than the other wallets. This is because Wallet 1 started the oracle and needed to pay transaction fees for that.

```
Slot 00006: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 1}:
  Contract instance started
Slot 00006: 00000000-0000-4000-8000-000000000003 {Contract instance for wallet 3}:
  Contract instance started
Slot 00006: 00000000-0000-4000-8000-000000000004 {Contract instance for wallet 4}:
  Contract instance started
Slot 00006: *** CONTRACT LOG: "own funds: [(,\"\",99999970),(ff,\"USDT\",100000000)]"
Slot 00006: 00000000-0000-4000-8000-000000000005 {Contract instance for wallet 5}:
  Contract instance started
Slot 00006: *** CONTRACT LOG: "own funds: [(,\"\",100000000),(ff,\"USDT\",100000000)]"
Slot 00006: 00000000-0000-4000-8000-000000000006 {Contract instance for wallet 3}:
  Contract instance started
Slot 00006: *** CONTRACT LOG: "own funds: [(,\"\",100000000),(ff,\"USDT\",100000000)]"
```

(continues on next page)

(continued from previous page)

```

Slot 00006: 00000000-0000-4000-8000-000000000007 {Contract instance for wallet 4}:
  Contract instance started
Slot 00006: *** CONTRACT LOG: "own funds: [(,\"\",100000000),(ff,\"USDT\",100000000)]"
Slot 00006: 00000000-0000-4000-8000-000000000008 {Contract instance for wallet 5}:
  Contract instance started
Slot 00006: *** CONTRACT LOG: "Oracle value: 15000000"
Slot 00006: 00000000-0000-4000-8000-000000000006 {Contract instance for wallet 3}:
  Receive endpoint call: Object (fromList [("tag",String "offer"),("value",Object_
  ↳(fromList [("unEndpointValue",Number 1.0e7)]))])
Slot 00006: W3: TxSubmit:
  ↳8274315b83fb8b4d721146a75772cf39be3f96730557bd6021235864f0f37bc6
Slot 00006: 00000000-0000-4000-8000-000000000007 {Contract instance for wallet 4}:
  Receive endpoint call: Object (fromList [("tag",String "offer"),("value",Object_
  ↳(fromList [("unEndpointValue",Number 2.0e7)]))])
Slot 00006: W4: TxSubmit:
  ↳221f86cc1d6087a5967793aaf9eb078d8ec0677b2d6aca586f985f6f2c57a100
Slot 00006: TxnValidate 221f86cc1d6087a5967793aaf9eb078d8ec0677b2d6aca586f985f6f2c57a100
Slot 00006: TxnValidate 8274315b83fb8b4d721146a75772cf39be3f96730557bd6021235864f0f37bc6

```

In slot 7, the offers of 10 Ada and 20 Ada are made.

```

Slot 00007: *** CONTRACT LOG: "own funds: [(,\"\",99999970),(ff,\"USDT\",100000000)]"
Slot 00007: *** CONTRACT LOG: "offered 100000000 lovelace for swap"
Slot 00007: *** CONTRACT LOG: "own funds: [(,\"\",89999990),(ff,\"USDT\",100000000)]"
Slot 00007: *** CONTRACT LOG: "offered 200000000 lovelace for swap"
Slot 00007: *** CONTRACT LOG: "own funds: [(,\"\",79999990),(ff,\"USDT\",100000000)]"
Slot 00007: *** CONTRACT LOG: "own funds: [(,\"\",100000000),(ff,\"USDT\",100000000)]"
Slot 00007: *** CONTRACT LOG: "Oracle value: 15000000"

```

In slot 9, the first *use* is requested.

And in slot 10, we see the swap happen.

```

Slot 00010: *** CONTRACT LOG: "own funds: [(,\"\",99999970),(ff,\"USDT\",100000000)]"
Slot 00010: *** CONTRACT LOG: "own funds: [(,\"\",89999990),(ff,\"USDT\",100000000)]"
Slot 00010: *** CONTRACT LOG: "own funds: [(,\"\",79999990),(ff,\"USDT\",130000000)]"
Slot 00010: *** CONTRACT LOG: "made swap with price [(ff,\"USDT\",300000000)]"
Slot 00010: *** CONTRACT LOG: "own funds: [(,\"\",118999990),(ff,\"USDT\",700000000)]"

```

The oracle gets and update request in slot 12.

```

Slot 00012: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "update"),("value",Object_
  ↳(fromList [("unEndpointValue",Number 1700000.0)]))])
Slot 00012: W1: TxSubmit:
  ↳ff7e1fbfb51897b100dcfdf551ad9a03886432af0a9fa92ff8dd986a0f7c90fe
Slot 00012: TxnValidate ff7e1fbfb51897b100dcfdf551ad9a03886432af0a9fa92ff8dd986a0f7c90fe

```

And we see it happen in slot 13.

```

Slot 00013: *** CONTRACT LOG: "updated oracle value to 17000000"

```

The second *use* request arrives at slot 15.

```

Slot 00015: 00000000-0000-4000-8000-000000000008 {Contract instance for wallet 5}:
  Receive endpoint call: Object (fromList [("tag",String "use"),("value",Object
  ↳(fromList [("unEndpointValue",Array [])]))))
Slot 00015: *** CONTRACT LOG: "own funds: [(,\"\",118999990),(ff,\"USDT\",700000000)]"
Slot 00015: *** CONTRACT LOG: "available assets: 700000000"
Slot 00015: *** CONTRACT LOG: "found oracle, exchange rate 17000000"
Slot 00015: W5: TxSubmit:
  ↳84bec9a9044eee9c5b40029dca5bbc8346214504e5adb4745bbe6e5d7d96078e
Slot 00015: TxnValidate 84bec9a9044eee9c5b40029dca5bbc8346214504e5adb4745bbe6e5d7d96078e

```

And the swap happens at slot 16.

```

Slot 00016: *** CONTRACT LOG: "made swap with price [(ff,\"USDT\",17000000)]"

```

And at the bottom, we see the final balances.

Wallet 2 still has all its funds. All Wallet 2 has done is the oracle checking, which doesn't cost anything, as it is purely an off-chain matter.

Wallet 1 paid some transaction fees but ends up with roughly 2 Ada more than it started with. This is because it collected the 2 Ada in fees for the use of the oracle.

Wallets 3 and 4 both made offers, and their balances reflect the exchange rates at which their offers were accepted.

Wallet 5 was the one accepting the offers, and so has the additional Ada, but a reduced USD Token balance. Note that Wallet 5 has also had some fees deducted from its Ada balance.

```

Final balances
Wallet 1:
  {, ""}: 101999950
  {ff, "USDT"}: 100000000
Wallet 2:
  {, ""}: 100000000
  {ff, "USDT"}: 100000000
Wallet 3:
  {ff, "USDT"}: 117000000
  {, ""}: 89999990
Wallet 4:
  {ff, "USDT"}: 130000000
  {, ""}: 79999990
Wallet 5:
  {, ""}: 127999980
  {ff, "USDT"}: 53000000
Wallet 6:
  {, ""}: 100000000
  {ff, "USDT"}: 100000000
Wallet 7:
  {, ""}: 100000000
  {ff, "USDT"}: 100000000
Wallet 8:
  {, ""}: 100000000
  {ff, "USDT"}: 100000000
Wallet 9:
  {, ""}: 100000000
  {ff, "USDT"}: 100000000

```

(continues on next page)

(continued from previous page)

```

Wallet 10:
  {, ""}: 1000000000
  {ff, "USDT"}: 1000000000

```

And finally, as well as the wallets, we see that the oracle is still going, and still owns the NFT. Note that, in this log, we don't see the datum value.

```

Script cc6a43073dce46eebc7b309223904c7a8033ffab7d9b239cf013342d4c69a5d6:
  {6122edd57c938cda24066f434da9ae55120b4eb362d4a1bd37547ef6e4a6cbb, ""}: 1

```

## 6.6 Plutus Application Backend

Apart from the idea of how to implement an oracle in Plutus, nothing we have done in this lecture so far is new, except for a few library functions and Haskell techniques. In principle, we are familiar with the way validators are written for off-chain code and how contracts are written for on-chain code, and how we can test our code with the *EmulatorTrace* monad.

But now we will talk about something new - the Plutus Application Backend (PAB), which allows us to take all the stuff we have done and turn it into an executable that runs the contracts.

If the testnet or the mainnet were available with Plutus support, we could deploy such an application, but for now we will need to be happy with a simulated blockchain. But, the process of turning the code into a dApp is practically the same as it would be on a real blockchain.

We need one more tiny module, which is basically just a type definition. It is so small, we can include the entire file contents here.

```

{-# LANGUAGE DeriveAnyClass      #-}
{-# LANGUAGE DeriveGeneric      #-}

module Week06.Oracle.PAB
  ( OracleContracts (..)
  ) where

import           Data.Aeson                (FromJSON, ToJSON)
import           Data.Text.Prettyprint.Doc (Pretty (..), viaShow)
import           GHC.Generics              (Generic)
import           Ledger

import qualified Week06.Oracle.Core         as Oracle

data OracleContracts = Init | Oracle CurrencySymbol | Swap Oracle.Oracle
  deriving (Eq, Ord, Show, Generic, FromJSON, ToJSON)

instance Pretty OracleContracts where
  pretty = viaShow

```

The idea is that it reifies the contract instances that we want to run. We have various contracts, and we want to have a data type where each value of the data type corresponds to a contract that we eventually want to run.

The *Init* constructor will be used to setup an environment where there is a USD Token available and where the wallets have an initial supply of those.

The *Oracle* constructor corresponds to the *runOracle* contract that will start the oracle and provide the *update* endpoint, and the *CurrencySymbol* parameter is going to be used for the USD Token.

Finally, the *Swap*, parameterized by *Oracle* will be used to run the swap contract, which provides various endpoints like *offer*, *retrieve*, *use* and *funds*.

We need to put the *OracleContracts* definition in a separate module because we will use it both from the PAB and also from the front end.

We are going to look at the Cabal file.

```
plutus-pioneer-program-week06.cabal
```

In it we have definitions for various executables.

The *oracle-pab* executable will set up a simulated wallet, initialize all the contracts and setup a web server that allows the outside world to interact with these contracts.

```
executable oracle-pab
main-is: oracle-pab.hs
hs-source-dirs:      app
ghc-options:         -Wall -threaded
build-depends:       aeson
                      , base ^>= 4.14.1.0
                      , freer-extras
                      , freer-simple
                      , plutus-contract
                      , plutus-ledger
                      , plutus-pab
                      , plutus-pioneer-program-week06
                      , plutus-use-cases
                      , text
```

The *oracle-client* executable will be run by the oracle provider, so that will interact with the *runOracle* contract. It will also fetch exchange rates from the internet and feed them into the system.

```
executable oracle-client
main-is: oracle-client.hs
hs-source-dirs:      app
ghc-options:         -Wall
build-depends:       base ^>= 4.14.1.0
                      , bytestring
                      , regex-tdfa ^>= 1.3.1.0
                      , req ^>= 3.9.0
                      , text
                      , uuid
```

Then there is the *swap-client* executable that will be run by the clients who want to make use of the swap contract.

```
executable swap-client
main-is: swap-client.hs
hs-source-dirs:      app
ghc-options:         -Wall
build-depends:       aeson
                      , base ^>= 4.14.1.0
                      , plutus-ledger
```

(continues on next page)

(continued from previous page)

```

    , plutus-pab
    , plutus-pioneer-program-week06
    , req ^>= 3.9.0
    , text
    , uuid

```

We will look at each of these in turn.

The code for each of these apps can be found in the *app* directory.

```

app/oracle-client.hs
app/oracle-pab.hs
app/swap-client.hs

```

### 6.6.1 Oracle PAB

First some boilerplate to hook up the data type we just defined - the reified contract instances - with the schemas and contracts that we defined earlier.

```

handleOracleContracts ::
  ( Member (Error PABError) effs
  , Member (LogMsg (PABMultiAgentMsg (Builtin OracleContracts))) effs
  )
=> ContractEffect (Builtin OracleContracts)
~> Eff effs
handleOracleContracts = handleBuiltin getSchema getContract where
  getSchema = \case
    Init      -> endpointsToSchemas @Empty
    Oracle _  -> endpointsToSchemas @(Oracle.OracleSchema .\\ BlockchainActions)
    Swap _    -> endpointsToSchemas @(Oracle.SwapSchema .\\ BlockchainActions)
  getContract = \case
    Init      -> SomeBuiltin initContract
    Oracle cs -> SomeBuiltin $ Oracle.runOracle $ oracleParams cs
    Swap oracle -> SomeBuiltin $ Oracle.swap oracle

```

*Init* won't have any schema, so it just has *BlockChainActions*. *Oracle* uses the *OracleSchema* and *Swap* uses the *SwapSchema*. No surprise there.

*Init* will run the *initContract*, which we will see in a moment.

*Oracle* will run the *runOracle* contract with *oracleParams* which takes the currency symbol of the USD Token and defines example oracle params.

```

oracleParams :: CurrencySymbol -> Oracle.OracleParams
oracleParams cs = Oracle.OracleParams
  { Oracle.opFees    = 1_000_000
  , Oracle.opSymbol = cs
  , Oracle.opToken  = usdt
  }

```

And finally *Swap* will run our swap contract with an oracle value.

Here is some more copy/paste boilerplate.

```
handlers :: SimulatorEffectHandlers (Builtin OracleContracts)
handlers =
    Simulator.mkSimulatorHandlers @(Builtin OracleContracts) []
    $ interpret handleOracleContracts
```

And here is the `initContract` function we mentioned just a moment ago.

```
initContract :: Contract (Last CurrencySymbol) BlockchainActions Text ()
initContract = do
    ownPK <- pubKeyHash <$> ownPubKey
    cur <-
        mapError (pack . show)
        (Currency.forgeContract ownPK [(usdt, fromIntegral (length wallets) * amount)]
         :: Contract (Last CurrencySymbol) BlockchainActions Currency.CurrencyError_)
    ↪ Currency.OneShotCurrency)
    let cs = Currency.currencySymbol cur
        v = Value.singleton cs usdt amount
    forM_ wallets $ \w -> do
        let pkh = pubKeyHash $ walletPubKey w
        when (pkh /= ownPK) $ do
            tx <- submitTx $ mustPayToPubKey pkh v
            awaitTxConfirmed $ txId tx
    tell $ Last $ Just cs
where
    amount :: Integer
    amount = 100_000_000
```

The `initContract` function mints USD Tokens and distributes them to the wallets, then it *tells* the currency symbol for the USD Token.

The wallets are hardcoded earlier in the code. The number of wallets and the tokens given to them are completely arbitrary.

```
wallets :: [Wallet]
wallets = [Wallet i | i <- [1 .. 5]]
```

Now we can look at the actual PAB code.

For the first time, we see a `main` function, which is the entry point for the executable.

```
main :: IO ()
main = void $ Simulator.runSimulationWith handlers $ do
    Simulator.logString @(Builtin OracleContracts) "Starting Oracle PAB webserver. Press_
    ↪ enter to exit."
    shutdown <- PAB.Server.startServerDebug

    cidInit <- Simulator.activateContract (Wallet 1) Init
    cs <- waitForLast cidInit
    _ <- Simulator.waitUntilFinished cidInit

    cidOracle <- Simulator.activateContract (Wallet 1) $ Oracle cs
    liftIO $ writeFile "oracle.cid" $ show $ unContractInstanceId cidOracle
    oracle <- waitForLast cidOracle
```

(continues on next page)

(continued from previous page)

```

forM_ wallets $ \w ->
  when (w /= Wallet 1) $ do
    cid <- Simulator.activateContract w $ Swap oracle
    liftIO $ writeFile ('W' : show (getWallet w) ++ ".cid") $ show $ _
    unContractInstanceId cid

void $ liftIO getLine
shutdown

```

The *main* function makes use of another monad that we haven't seen before and is specific to the PAB - the *Simulator* monad.

The *Simulator* monad is very similar to the *EmulatorTrace* monad. In principle it has the same capabilities. You can start contracts on wallets, you can inspect the state using the log, you can call endpoints, and so on.

It is a bit unfortunate that there are two monads for this as they are so similar. The Plutus team plan to align them and maybe turn them into one. So it may not be worth learning the intricacies of the *Simulator* monad as it will probably change soon.

Similar to the *runEmulatorTraceIO*, we have *runSimulationWith* to which we pass the *handlers* boilerplate.

One significant difference to the *EmulatorTrace* monad though is that the *EmulatorTrace* monad was pure code - there were no real world side-effects, no IO involved. In particular there is a pure interpreter *runEmulatorTrace* that is a pure Haskell function with no side-effects.

*Simulator* is different - you can do IO. The way it works is using *MonadIO*, which has one method, *liftIO*, which takes an *IO* action and *lifts* it into the monad in question. So, if you have some arbitrary IO action that you can do in Haskell, then by applying *liftIO* to it, you can move it into this *Simulator* monad.

Apart from that, if you squint, it looks very similar to an *EmulatorTrace*.

The first thing we do is use *logString* to log that we are starting the PAB server. We then call the *startServerDebug* function, and the return value of that function which gets bound to *shutdown* can be used later to shut down the server.

Now we use something called *activateContract* which is the equivalent of *activateContractWallet* from the *EmulatorTrace* monad.

```
cidInit <- Simulator.activateContract (Wallet 1) Init
```

It takes a wallet where we want to start that instance, and then a value of the reified contract type. Remember that we associated the *Init* constructor with the *initContract* function.

Now we need the currency symbol. This is an example of how we get information out of a contract using *tell*.

```
cs <- waitForLast cidInit
```

The function *cidInit* uses a function from the *Simulator* monad called *waitForState*, which takes a contract instance and a predicate. The predicate gets a JSON expression and returns a *Maybe a*.

```

waitForLast :: FromJSON a => ContractInstanceId -> Simulator.Simulation t a
waitForLast cid =
  flip Simulator.waitForState cid $ \json -> case fromJSON json of
    Success (Last (Just x)) -> Just x
    _                        -> Nothing

```

The idea is that it will read the state of the contract which we wrote using *tell*. This is serialized as a JSON value, and it applies this JSON value to the provided predicate. If the result is *Nothing*, it simply waits until the state changes again. But, if it is *Just x*, it will return the *x*.



There are two ways it could be *Nothing* - either the JSON parsing could fail, or we could get a *Last Nothing*. So, the end result is that the function waits until the state of the contract has told a *Just* value.

At this point we have the currency symbol bound to *cs*. And then we wait until *initContract* has finished.

```
_ <- Simulator.waitUntilFinished cidInit
```

The next step is to start the oracle on Wallet 1, using the *cs* value we have recently obtained.

```
cidOracle <- Simulator.activateContract (Wallet 1) $ Oracle cs
```

In order to interact with the oracle contract from the outside world, e.g. from the web interface, we need to get our hands on the *cidOracle* handle.

So what we do is write this into a file called *oracle.cid*.

```
liftIO $ writeFile "oracle.cid" $ show $ unContractInstanceId cidOracle
```

This is just quick and dirty for the demonstration. In production code you would use a safer mechanism.

Now we use *waitForLast* again to get the oracle value, which we have provided from the *runOracle* contract via a *tell*. We need this because the swap contract is parameterized by this value.

At this stage the NFT is minted and we know what the oracle value is.

Next, we loop over all the wallets, except wallet 1 which runs the oracle, and we activate the swap contract on each of them. Here we use a similar file-writing method to get hold of the contract handles.

```
forM_ wallets $ \w ->
  when (w /= Wallet 1) $ do
    cid <- Simulator.activateContract w $ Swap oracle
    liftIO $ writeFile ('W' : show (getWallet w) ++ ".cid") $ show $
      unContractInstanceId cid
```

Now we just block until the user presses enter, and then we shutdown the server.

```
void $ liftIO getLine
shutdown
```

It is not actually necessary to do all of this, because you can also start and stop contract instances from the web interface. It was easier for us here to do it in a scripted way for the demo, but in principle you could just start the simulator and then wait until you shutdown.

If you are curious about the API provided by the PAB, you can check that in the *plutus-pab* package, in the module *Plutus.PAB.Webserver.API*. There are several, but the one that we are using here is:

```
-- | PAB client API for contracts of type @t@. Examples of @t@ are
-- * Contract executables that reside in the user's file system
-- * "Builtin" contracts that run in the same process as the PAB (ie. the PAB is
--   compiled & distributed with these contracts)
type NewAPI t walletId -- see note [WalletID type in wallet API]
  = "api"   => "new"   => "contract" =>
    ("activate" => ReqBody '[JSON] (ContractActivationArgs t) => Post '[JSON]
    ContractInstanceId -- start a new instance
    :<|> "instance" =>
      (Capture "contract-instance-id" Text =>
        ( "status" => Get '[JSON] (ContractInstanceClientState t) --
        Current status of contract instance
```

(continues on next page)

(continued from previous page)

```

        :<|> "endpoint" => Capture "endpoint-name" String => ReqBody
    => '[JSON] JSON.Value => Post '[JSON] () -- Call an endpoint. Make
        :<|> "stop" => Put '[JSON] () -- Terminate the instance.
    )
)

:~> "instances" => "wallet" => Capture "wallet-id" walletId => Get '[JSON]_
=> [ContractInstanceClientState t]
:~> "instances" => Get '[JSON] [ContractInstanceClientState t] -- list of_
=> all active contract instances
:~> "definitions" => Get '[JSON] [ContractSignatureResponse t] -- list of_
=> available contracts
)

```

This makes use of the popular Haskell library *Servant* to write type safe web applications, but it should be readable more or less without knowledge of the *Servant* library. For example, you can see the `/api/new/contract/activate` endpoint declared to which you can POST a *ContractActivationArgs* as its body and returns a *ContractInstanceId*.

There is also a web socket API, but we have not used that in this example.

So let's try our executable.

```
cabal run oracle-pab
```

We get log output similar to what we see with *EmulatorTrace*, but this is now a live server.

The output below is reduced to avoid the full verbosity.

```

[INFO] Slot 0: TxnValidate_
=> af5e6d25b5ecb26185289a03d50786b7ac4425b21849143ed7e18bcd70dc4db8
[INFO] Starting Oracle PAB webserver. Press enter to exit.
[INFO] Starting PAB backend server on port: 8080
[INFO] Activated instance dda39c83-5a0f-484f-b49a-9943b1ff5526 on W1
[INFO] Slot 1: W1: Balancing an unbalanced transaction:
      Tx:
      Tx_
=> da767478580878690990b3a22387be9c5b27fabed2c0ca7b9991eb682a6781f8:
      {inputs:
      outputs:
      - Value (Map [(,Map [("",1)])]) addressed to
      addressed to ScriptCredential:_
=> e9827f1a9e43109d1c8d4555913734b8c48a467a31061b312959f270850fc8a0 (no staking_
=> credential)

      forge: Value (Map [])
      fee: Value (Map [(,Map [("",10)])])
      mps:
      signatures:
      validity range: Interval {ivFrom = LowerBound NegInf True, ivTo_
=> = UpperBound PosInf True}
      data:
      <>}
      Requires signatures:
[INFO] Slot 1: TxnValidate_
=> b7d6ba18d02898aa5d0814a306e4a05cf153c199aabb175ef9b00328105ab98f
[INFO] Slot 2: W1: Balancing an unbalanced transaction:

```

(continues on next page)

(continued from previous page)

```

...
[INFO] Slot 6: TxnValidate_
↳ d690122263521c37f308a0d5ae858aa4807cb1e493c2a3374bf65b934c74782a
[INFO] Activated instance deceaa52-f117-46bc-b0f1-eb1f2f529b5a on W1
[INFO] Slot 7: W1: Balancing an unbalanced transaction:
...
[INFO] Slot 7: TxnValidate_
↳ a679948d128735ec1f380e9f733d7f4a5e54c81f39c73a656d61b077111840e1
[INFO] Slot 8: W1: Balancing an unbalanced transaction:
...
[INFO] Slot 8: TxnValidate_
↳ f0125e685edd6de2d09f9547f53b18d1bad11bd7fad570a057ba74737ab3053e
[INFO] deceaa52-f117-46bc-b0f1-eb1f2f529b5a: "started oracle Oracle {oSymbol =_
↳ b8a1d67cd94acf75d7e00f27015ec5e31242adad0967eee473f49c5d1d686169, oOperator =_
↳ 21fe31dfa154a261626bf854046fd2271b7bed4b6abe45aa58877ef47f9721b9, oFee = 1000000,_
↳ oAsset = (9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a,\"USDT\")}"
[INFO] Activated instance 5bac6e67-f956-45ef-b386-f1d045cf5e37 on W2
[INFO] Activated instance 2c3a2794-2592-4c2b-9a7d-9c810cedf886 on W3
[INFO] Activated instance 6f8d611d-a3f6-4794-9048-8ea3201e3c56 on W4
[INFO] Activated instance 387d9651-6024-48c1-a72d-5b3c6d3a6b53 on W5

```

We can see, for example, where we can find instance IDs of contracts if we want to interact with them via the API.

```
[INFO] Activated instance deceaa52-f117-46bc-b0f1-eb1f2f529b5a on W1
```

If we now stop the server and look in the directory, we will see the files where we stored the instance IDs.

```

[nix-shell:~/git/ada/pioneer-fork/code/week06]$ ls
app                dist-newstyle      LICENSE            plutus-pioneer-program-week06.cabal  W2.cid _
↳ W4.cid
cabal.project      hie.yaml           oracle.cid         src                                    W3.cid _
↳ W5.cid

[nix-shell:~/git/ada/pioneer-fork/code/week06]$ cat W5.cid
387d9651-6024-48c1-a72d-5b3c6d3a6b53

```

With this information - either obtained from the web server log, or from the files we have created, we could use any HTTP tool such as Curl or Postman to interact with the contracts when the web server is running. By default it runs on port 8080. We could also write code in any programming language we like to interact with the web server using the HTTP endpoints.

We will now briefly look at the *oracle-client* and the *swap-client*. We won't go into too much detail because we are not so interested in how to write a front end here.

## 6.6.2 Oracle Client

We use the Haskell library *Req* to interact with the web server.

We first read the *oracle.cid* file to get the oracle instance ID. Then we have a recursive function *go*.

The function *go* looks up the current exchange rate on CoinMarketCap, checks whether that has changed, and, if it has change, it calls *updateOracle* which calls the update oracle endpoint on our contract. And, whether or not a change is detected, it waits for an arbitrary five seconds, and then goes again.

The length of the delay would depend on things such as that rate cap imposed by CoinMarketCap. In reality, as blocks on Cardano only appear every twenty seconds, five seconds is probably too short.

```
main :: IO ()
main = do
  uuid <- read <$> readFile "oracle.cid"
  putStrLn $ "oracle contract instance id: " ++ show uuid
  go uuid Nothing
  where
    go :: UUID -> Maybe Integer -> IO a
    go uuid m = do
      x <- getExchangeRate
      let y = Just x
      when (m /= y) $
        updateOracle uuid x
      threadDelay 5_000_000
      go uuid y
```

Now, the *updateOracle* function prepares an POST request using the oracle instance ID and a JSON body containing the exchange rate.

```
updateOracle :: UUID -> Integer -> IO ()
updateOracle uuid x = runReq defaultHttpConfig $ do
  v <- req
    POST
    (http "127.0.0.1" /: "api" /: "new" /: "contract" /: "instance" /: pack (show_
  → uuid) /: "endpoint" /: "update")
    (ReqBodyJson x)
    (Proxy :: Proxy (JsonResponse ()))
    (port 8080)
  liftIO $ putStrLn $ if responseStatusCode v == 200
    then "updated oracle to " ++ show x
    else "error updating oracle"
```

And here is the *getExchangeRate* function which is a quick and dirty way of getting the exchange rate from CoinMarketCap. They provide a proper API, but here we are just doing some screen scraping from the web page and using a regex to extract the value we are interested in. This is, of course, very fragile, and could never be used as production code.

```
getExchangeRate :: IO Integer
getExchangeRate = runReq defaultHttpConfig $ do
  v <- req
    GET
    (https "coinmarketcap.com" /: "currencies" /: "cardano")
    NoReqBody
```

(continues on next page)

(continued from previous page)

```

    bsResponse
    mempty
    let priceRegex      = "priceValue____11gHJ\">\\$([\\$.0-9]*)\" :: ByteString
        (_, _, _, [bs]) = responseBody v =~ priceRegex :: (ByteString, ByteString, ByteString, [ByteString])
        d                = read $ unpack bs :: Double
        x                = round $ 1_000_000 * d
    liftIO $ putStrLn $ "queried exchange rate: " ++ show d
    return x

```

Let us run it.

First we need to make sure the PAB is running.

```
cabal run oracle-pab
```

Then, in another terminal

```

cabal run oracle-client
...
queried exchange rate: 1.54
updated oracle to 1540000
queried exchange rate: 1.54

```

We can see the exchange rate that it has obtained from CoinMarketCap, and its request to update the oracle.

And if we wait long enough, we see

```

queried exchange rate: 1.55
updated oracle to 1550000
queried exchange rate: 1.55

```

And we see that we are mooning.

If you switch back to the PAB, you will also see additional log messages.

```

[INFO] Slot 16: W1: Balancing an unbalanced transaction:
  Tx:
  Tx c5b384f75f93ebc8f1e6b514237aa70d0d982e9b035eececa27af0b3e72568e4:
    {inputs:
    outputs:
      - Value (Map [(b8a1d67cd94acf75d7e00f27015ec5e31242adad0967eee473f49c5d1d686169,
↳ Map [( "", 1) ])) addressed to
        addressed to ScriptCredential:
↳ 04a718132f7ca493a011c40926e191a76bd84cbf8e7c14b6c99bbea8b8bc0bba (no staking
↳ credential)
        forge: Value (Map [])
        fee: Value (Map [( , Map [( "", 10) ] )])
        mps:
        signatures:
        validity range: Interval {ivFrom = LowerBound NegInf True, ivTo = UpperBound
↳ PosInf True}
        data:
        1540000}

```

(continues on next page)

(continued from previous page)

```

Requires signatures:
[INFO] Slot 16: TxnValidate
↳ 40c5dbb5e7c8de390a6943d8f0a84d218cf86dd81af1fd7cfc62612e6b616c2c
[INFO] 5d9d778e-55f9-45ab-89a6-2ba9aa18045e: "set initial oracle value to 15400000"

```

### 6.6.3 Swap Client

The swap client is very similar.

Here, we are just giving a simple console interface, so we didn't bother with graphics or a nice web UI.

```

main :: IO ()
main = do
  [i :: Int] <- map read <$> getArgs
  uuid <- read <$> readFile ('W' : show i ++ ".cid")
  hSetBuffering stdout NoBuffering
  putStrLn $ "swap contract instance id for Wallet " ++ show i ++ ": " ++ show uuid
  go uuid
where
  go :: UUID -> IO a
  go uuid = do
    cmd <- readCommand
    case cmd of
      Offer amt -> offer uuid amt
      Retrieve -> retrieve uuid
      Use -> use uuid
      Funds -> getFunds uuid
    go uuid

  readCommand :: IO Command
  readCommand = do
    putStr "enter command (Offer amt, Retrieve, Use or Funds): "
    s <- getLine
    maybe readCommand return $ readMaybe s

```

The idea is to take a command from the console and then call the appropriate endpoint.

```

case cmd of
  Offer amt -> offer uuid amt
  Retrieve -> retrieve uuid
  Use -> use uuid
  Funds -> getFunds uuid

```

The endpoint calling uses the same method for each endpoint, creating an HTTP call in the same way that we did for the oracle client.

The *getFunds* function is slightly more complicated than the other three as it needs to get information out of the server. For this it needs to make two requests. The second request is to read the state that was *told* by the first call.

```

getFunds :: UUID -> IO ()
getFunds uuid = handle h $ runReq defaultHttpConfig $ do
  v <- req

```

(continues on next page)

(continued from previous page)

```

    POST
    (http "127.0.0.1" /: "api" /: "new" /: "contract" /: "instance" /: pack (show_
↳ uuid) /: "endpoint" /: "funds")
    (ReqBodyJson ())
    (Proxy :: Proxy (JsonResponse ()))
    (port 8080)
    if responseStatusCode v /= 200
    then liftIO $ putStrLn "error getting funds"
    else do
        w <- req
        GET
        (http "127.0.0.1" /: "api" /: "new" /: "contract" /: "instance" /: pack_
↳ (show uuid) /: "status")
        NoReqBody
        (Proxy :: Proxy (JsonResponse (ContractInstanceClientState_
↳ OracleContracts))))
        (port 8080)
        liftIO $ putStrLn $ case fromJSON $ observableState $ cicCurrentState $_
↳ responseBody w of
            Success (Last (Just f)) -> "funds: " ++ show (flattenValue f)
            -                          -> "error decoding state"
    where
        h :: HttpException -> IO ()
        h _ = threadDelay 1_000_000 >> getFunds uuid

```

Let's run the swap client. We will leave the web server and the oracle client running.

When using cabal, we pass parameters in following a -. For the swap client we pass the wallet number in as a parameter.

We will launch the swap client for wallets 2 and 3, each in a separate window, and query their respective funds.

```

cabal run swap-client -- 2

swap contract instance id for Wallet 2: ab65f248-450d-4988-ab2a-651ad5697596
enter command (Offer amt, Retrieve, Use or Funds): Funds
funds: [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a,"USDT",
↳ 1000000000),(",","1000000000")]
enter command (Offer amt, Retrieve, Use or Funds):

```

```

cabal run swap-client -- 3

swap contract instance id for Wallet 3: 2dc4f6f2-142e-40a2-a1b8-c431eb29a3a2
enter command (Offer amt, Retrieve, Use or Funds): Funds
funds: [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a,"USDT",
↳ 1000000000),(",","1000000000")]
enter command (Offer amt, Retrieve, Use or Funds):

```

Wallet 2 now offers 10 Ada as a swap, and we check the funds, and we see that the Ada balance has gone down (by the 10 Ada plus the transaction fee), but the USD Token balance remains the same.

```

enter command (Offer amt, Retrieve, Use or Funds): Offer 100000000
offered swap of 100000000 lovelace
enter command (Offer amt, Retrieve, Use or Funds): Funds

```

(continues on next page)

(continued from previous page)

```
funds: [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a, "USDT",  
↪ 1000000000), (, "", 89999990)]
```

While these commands are running, you can also see the calls being made in the PAB output.

```
INFO] Slot 1662: TxnValidate_  
↪ 17f640f03e4dc7d0a4c246129454aa19daa8d9d674bfebeeee486d6143c6648e  
[INFO] ab65f248-450d-4988-ab2a-651ad5697596: "offered 100000000 lovelace for swap"  
[INFO] ab65f248-450d-4988-ab2a-651ad5697596: "own funds:_  
↪ [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a, \"USDT\", 1000000000),  
↪ (, \"\", 89999990)]"
```

Now, Wallet 3 is going to take up the offer of the swap.

```
enter command (Offer amt, Retrieve, Use or Funds): Use  
used swap  
enter command (Offer amt, Retrieve, Use or Funds): Funds  
funds: [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a, "USDT",  
↪ 1000000000), (, "", 1000000000)]
```

It takes a little while for the funds to update, so let's try the *Funds* command again.

```
enter command (Offer amt, Retrieve, Use or Funds): Funds  
funds: [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a, "USDT",  
↪ 844000000), (, "", 108999990)]
```

That's better. And we can see that wallet 3 has received the 10 Ada, minus the oracle fee of 1 Ada and minus the transaction fees.

In the PAB output, we see something like

```
[INFO] Slot 1868: TxnValidate_  
↪ 00afd25af063d58b4f290e43057f4738483098f26ff0134bc14c9d54b9b94090  
[INFO] 2dc4f6f2-142e-40a2-a1b8-c431eb29a3a2: "made swap with price_  
↪ [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a, \"USDT\", 156000000)]"  
[INFO] 2dc4f6f2-142e-40a2-a1b8-c431eb29a3a2: "own funds:_  
↪ [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a, \"USDT\", 844000000),  
↪ (, \"\", 108999990)]"
```

Let's look at wallet 2's funds.

```
enter command (Offer amt, Retrieve, Use or Funds): Funds  
funds: [(9a91216e55e5369b926acc07c70a11d9ae7fef454e43e3e5c0aa1733f48c798a, "USDT",  
↪ 1156000000), (, "", 89999990)]
```

And we see that wallet 2 has lost some Ada, but gained some USD Tokens. The swap is complete, using the exchange rate as it is live, right now, which was injected into the mock blockchain via the oracle.

So now we have seen an end-to-end example of a Plutus dApp. It has a front end, it talks to the outside world, goes on the internet, gets information and interacts with Plutus smart contracts. The smart contracts submit transactions to the blockchain where the validation logic kicks in and makes sure that everything follows the business rules.

In this example, as we have no real blockchain to play with, all wallets use the same PAB server, which, of course, in real life would be silly. Obviously, different wallets will have different instances of PAB running.

But, apart from that, it is almost exactly, end-to-end, how such a system would work.



## WEEK 07 - STATE MACHINES

---

**Note:** This is a written version of [Lecture #7](#).

It covers commit schemes and state machines.

This week we were using Plutus commit 530cc134364ae186f39fb2b54239fb7c5e2986e9

---

### 7.1 Introduction

In this lecture we will look at state machines. State machines can be very useful for writing shorter and more concise contracts, both on-chain and off-chain. There is higher level support for state machines in the Plutus libraries that builds on top of the lower level mechanisms we have seen so far.

As a running example, we are going to implement a little game, played between Alice and Bob. It's a bit like Rock, Paper, Scissors, but even simpler, because there are only two options.

Alice and Bob both have two options, they can either play 0 or 1.

A hand-drawn payoff matrix for a game between Alice and Bob. The matrix is a 2x2 grid. The columns are labeled 'Bob' with options '0' and '1'. The rows are labeled 'Alice' with options '0' and '1'. The matrix is drawn with a vertical line separating the labels from the grid, and a horizontal line separating the column labels from the grid.

		Bob	
		0	1
Alice	0		
	1		

If there were to play this game while being physically in the same room, they would make their moves at the same time. There would be one gesture for 0 and one gesture for 1, they would raise their hands simultaneously, and, depending on what they play, one of them wins.

If they both play the same number, Alice wins. If they play different numbers, Bob wins.

A hand-drawn payoff matrix for a game between Alice and Bob. The matrix is a 2x2 grid with 'Alice' on the left and 'Bob' on top. The rows are labeled '0' and '1' for Alice, and the columns are labeled '0' and '1' for Bob. The payoffs are written in the cells: (0,0) is 'Alice', (0,1) is 'Bob', (1,0) is 'Bob', and (1,1) is 'Alice'.

		Bob	
		0	1
Alice	0	Alice	Bob
	1	Bob	Alice

Now let's imagine that Alice and Bob can't meet in person but that they still want to play the game. So, they decide to play it via mail - email or snail mail, it doesn't matter. How would that work?

Alice could send her move to Bob.

This, however, gives a very unfair advantage to Bob, because now he opens Alice's mail, see that she has played 0, and he can simply reply with 1, and he wins.

And, if Alice plays 1, Bob can simply respond with 1. So Bob always wins, at least if he is unfair.

What can we do about that?

There's a very clever trick which is often used in cryptographic protocols, and that is commit schemes. The idea is that Alice doesn't reveal her choice to Bob, but she commits to it, so that she cannot later change her mind.

One way to make that work is using hash functions.

Hashes are all over the place in the blockchain world. We have seen that script addresses are just the hash of the Plutus code script, and we have seen lots of examples of public key hashes.

A hash function is a one-way function. Given a hash, it is difficult, or impossible, to reconstruct the original byte string that was hashed.

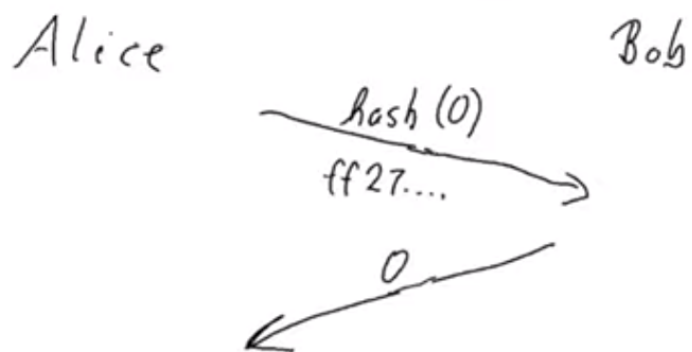
So, one way we could try to make this work is that, instead of Alice sending her choice to Bob, she instead sends the hash of her choice.

Bob then sees this cryptic byte string and he has no idea whether Alice picked 0 or 1.

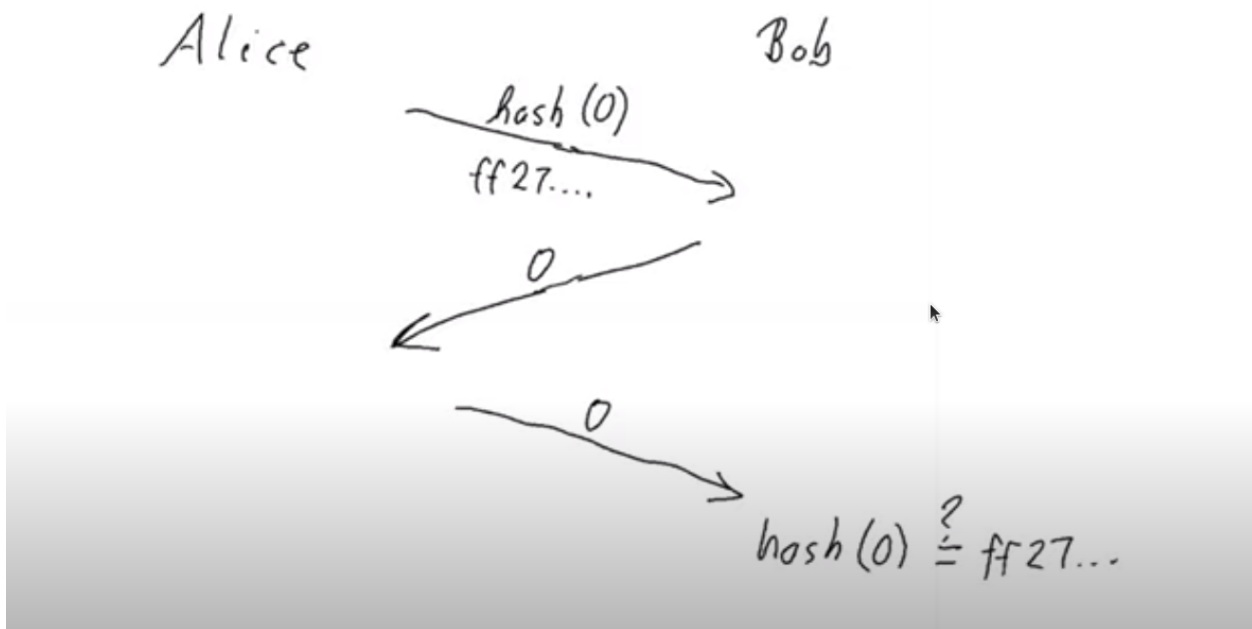
Bob then replies with his move, picking, for example 0. There is no need for him to use a hash, he can just send his response in clear text.

Now, Alice would have won. But perhaps Bob doesn't believe her. So there is one additional step that Alice has to take.





Alice has to send her actual choice to Bob in clear text. Bob then has to check that the hash of her choice is indeed the same as the hash Alice sent earlier.



If it is, then he knows that Alice is not lying and that indeed he lost. If it does not match, then he knows that Alice is cheating and he would win.

This all sounds promising, but there is one big problem with it.

In this game there are only two choices, 0 and 1. Which means that there are only two possible hashes. They may look very cryptic to Bob the first time they play, but before long he will notice that he always sees one of only two possible hashes, and then he can know which choice Alice made.

This is almost as bad as if Alice had just sent her choice in clear text.

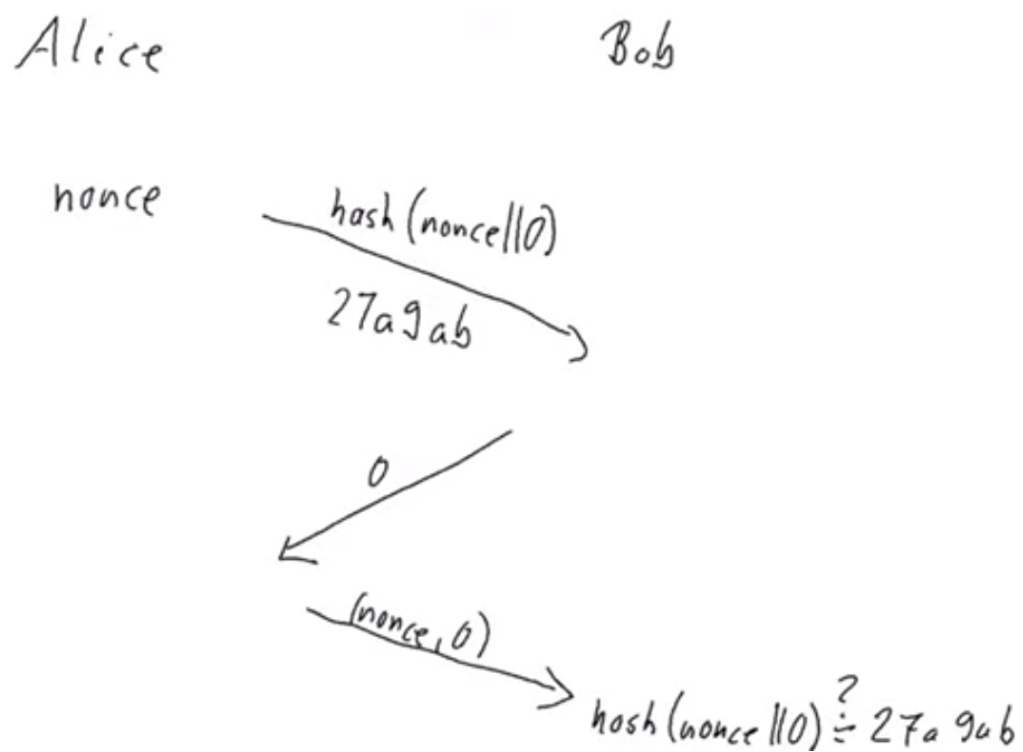
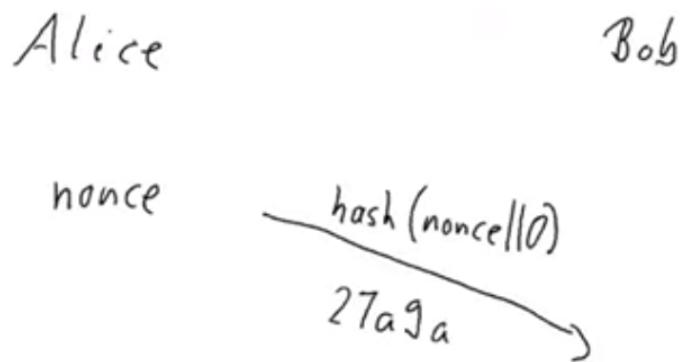
What we can do about this is that, instead of sending the hash of her choice, she instead first selects an arbitrary byte string and then hashes the concatenated of this byte string and her choice. The arbitrary byte string that Alice chooses is called a *nonce* - a number to be used just once.

So now, it is not always the same byte string if she picks 0, provided she chooses some random, unpredictable nonce.

Now, Bob receives this and we proceed as before - Bob sends his choice, and then, in the third message Alice needs to send not only her original choice, but she also has to send the nonce as well.

And then Bob checks that the hash of Alice's claimed nonce concatenated with her choice is indeed the hash that he originally received. If it is, he knows he lost, and if it is not, he knows that she tried to cheat him.

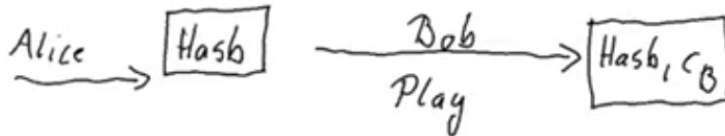
This works very nicely and this is what we will try to implement in Cardano. First we will do it using techniques we have already seen, and then we will see how, by using state machines, the code can be much clearer and much shorter.



## 7.2 Code Example 1

We can imagine that, at the start of the game, Alice and Bob have put down the same amount of money each and that the winner will take it all.

The game starts with Alice posting her hash, as described above. Bob, if he plays along, will post his own choice. At this point, we have Alice's hash and Bob's choice.



If, at this point, Alice realizes that she has won, based on Bob's choice, she can reveal her secret, the game ends, and she has won.

If, however, after Bob makes his move, Alice sees that she has lost, there is no need for her to do anything. After a certain deadline has been reached, if Alice has not responded, Bob will be able to claim the funds.

There is another scenario. Perhaps, after Alice starts playing, Bob simply isn't interested. In this case, there must be a way for Alice to get her own money back.

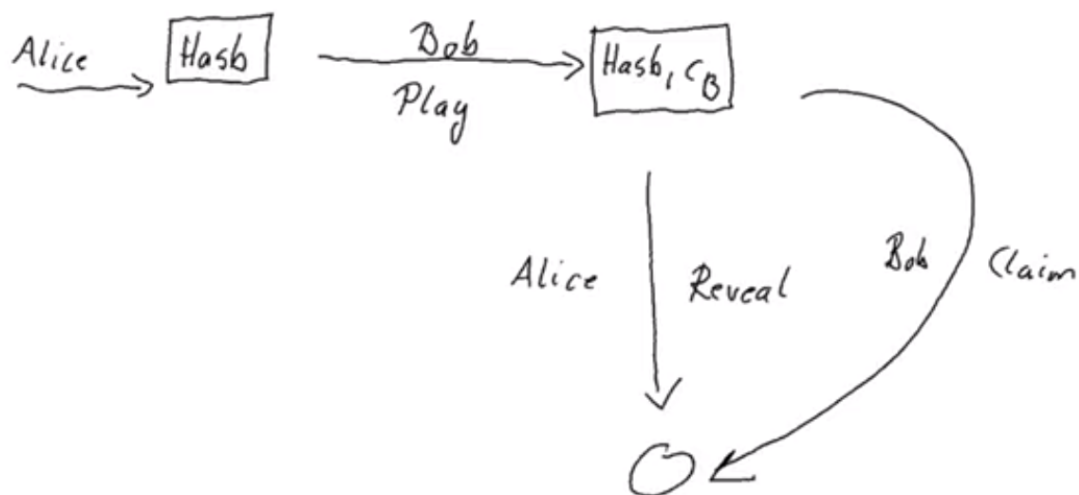
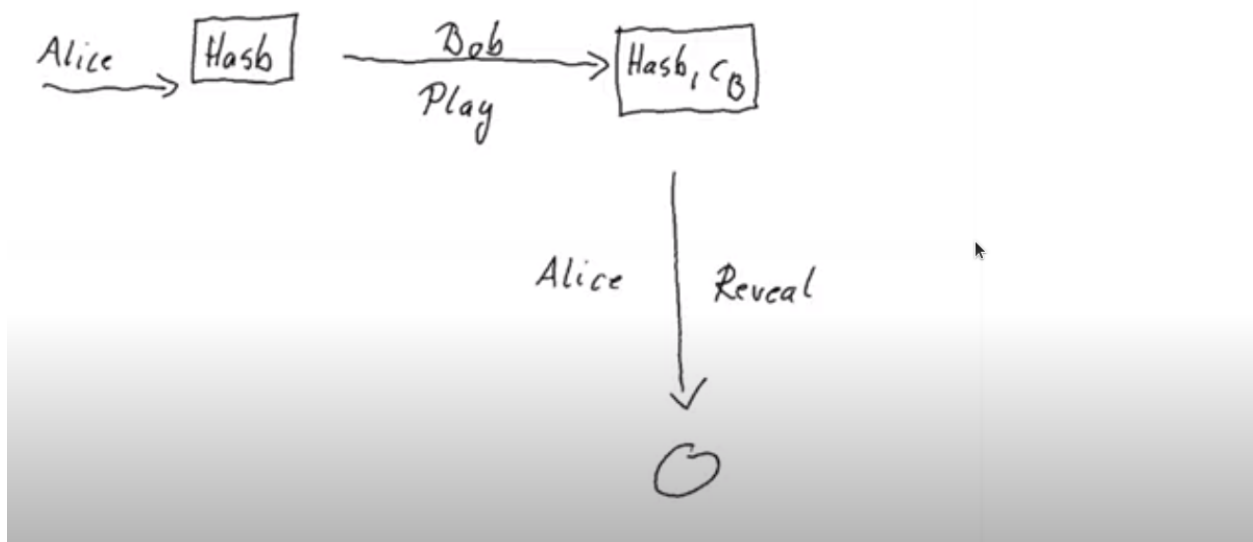
As mentioned, our first attempt at coding this in Plutus will be using the techniques we have learned in previous lectures.

The code we are working with is in the following module

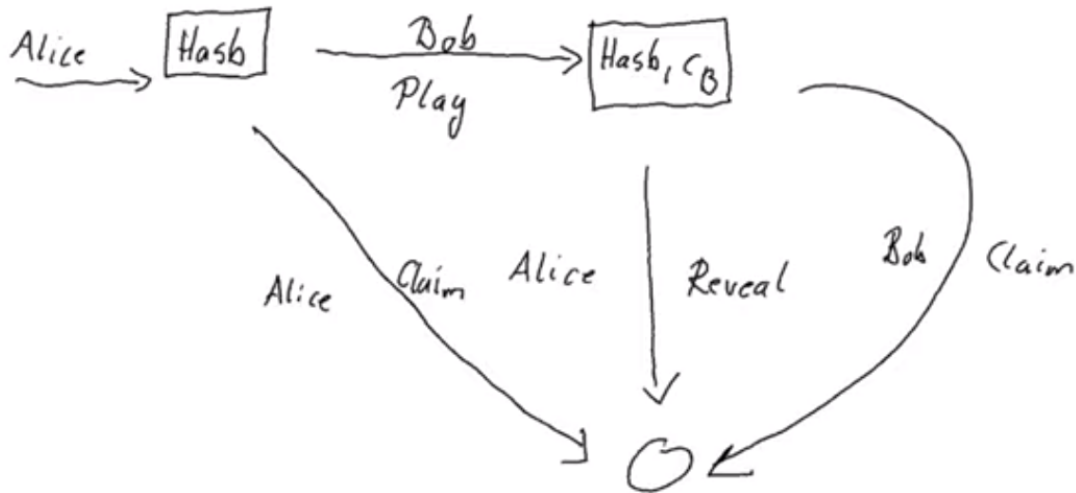
```
module Week07.EvenOdd
```

We call the game *EvenOdd* due to the fact that if the sum of the numbers is even, then the first player wins, and if the sum is odd, the second player wins.

In our code we will call the players *first* and *second* rather than Alice and Bob.







### 7.2.1 On chain

We define a data type *Game* which will be used as a parameter for the contract.

```

data Game = Game
  { gFirst      :: !PubKeyHash
  , gSecond     :: !PubKeyHash
  , gStake      :: !Integer
  , gPlayDeadline :: !Slot
  , gRevealDeadline :: !Slot
  , gToken      :: !AssetClass
  } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq, Prelude.Ord)

```

The players are identified by their public key hashes as *gFirst* and *gSecond*.

The number of lovelace to be used as stake in the game is represented by *gStake* - This stake must be provided by each player.

There are two deadlines. The *gPlayDeadline* is the slot by which the second player must make their move. In the case where the second player has made a move, the *gRevealDeadline* is the slot by which player 1 must claim victory by revealing his nonce.

Finally we have a token represented by *gToken*. This will be the same trick that we used for the oracle. It will be an arbitrary NFT, used to identify the right instance of the UTxO that we are using. The idea is to use the datum sitting at a UTxO in this contract's script address to keep track of where we are in the game.

Next, we define the two moves that the players can make.

```

data GameChoice = Zero | One
  deriving (Show, Generic, FromJSON, ToJSON, ToSchema, Prelude.Eq, Prelude.Ord)

instance Eq GameChoice where
  {-# INLINABLE (==) #-}

```

(continues on next page)

(continued from previous page)

```
Zero == Zero = True
One  == One  = True
_    == _    = False
```

We need Plutus *Eq* for the instance, but it is not possible to declare that in the deriving clause, which is why the *Eq* in the deriving clause is qualified as being from the standard Haskell Prelude.

Note that we have used the *INLINABLE* pragma on the *Eq* instance for *GameChoice*. This is again to make it compatible with the Template Haskell we will need to use.

For state, we will use a type called *GameDatum*.

```
data GameDatum = GameDatum ByteString (Maybe GameChoice)
deriving Show

instance Eq GameDatum where
  {-# INLINABLE (==) #-}
  GameDatum bs mc == GameDatum bs' mc' = (bs == bs') && (mc == mc')
```

Here, the *ByteString* is the hash that the first player submits, and *Maybe GameChoice* is either *Just* the move of the second player, or *Nothing*, if they have not yet moved.

Now we come to the redeemer, and we will use a custom type for this as well.

```
data GameRedeemer = Play GameChoice | Reveal ByteString | ClaimFirst | ClaimSecond
deriving Show
```

Here *Play* is where the second player moves and, as an argument, it has a *GameChoice*. *Reveal* is for the case where the first player has won and must prove that by revealing their nonce, and the nonce is represented by the *ByteString* argument. We don't need to include the move for the *Reveal*, as they will only reveal if they have won, and we know what move makes them win.

*ClaimFirst* is when the first player claims back the stake in the even that the second player does not make a move by the play deadline. *ClaimSecond* is for the case when the first player does not reveal by the reveal deadline.

We then have our *lovelaces* helper function which we have used in other scripts, which gets the number of lovelaces held in a *Value*.

```
lovelaces :: Value -> Integer
lovelaces = Ada.getLovelace . Ada.fromValue
```

And we have a helper function *gameDatum* which behaves exactly the same way as the function *oracleValue*, which you can find in the notes from lecture 6.

```
gameDatum :: TxOut -> (DatumHash -> Maybe Datum) -> Maybe GameDatum
gameDatum o f = do
  dh    <- txOutDatum o
  Datum d <- f dh
  PlutusTx.fromData d
```

Now we come to the core business logic in the *mkGameValidator* function.

```
mkGameValidator :: Game -> ByteString -> ByteString -> GameDatum -> GameRedeemer ->
  ↳ ScriptContext -> Bool
mkGameValidator game bsZero' bsOne' dat red ctx =
  ...
```

The first argument is the *Game* parameter discussed above.

The second and the third arguments are somewhat of a nuisance. We just need them due to the fact that it is not possible to use string literals to get *ByteStrings* in Haskell that is compiled to Plutus core. And, we want string literals representing the 0 and 1 choices. So *bsZero* will hold “0” and *bsOne* will hold “1”. You will see how we pass these in as auxiliary arguments later.

Then we pass in the usual arguments for datum, redeemer and context.

Let’s look at some helper functions first. There are three functions we have used before and discussed in lecture 6.

```
info :: TxInfo
info = scriptContextTxInfo ctx

ownInput :: TxOut
ownInput = case findOwnInput ctx of
    Nothing -> traceError "game input missing"
    Just i   -> txInInfoResolved i

ownOutput :: TxOut
ownOutput = case getContinuingOutputs ctx of
    [o] -> o
    _    -> traceError "expected exactly one game output"
```

Note the *ownInput* should never fail as we are in the process of validating a UTxO.

The *outputDatum* helper makes use of the *GameDatum* type which we defined earlier. Given the case we have exactly one output (the return from *ownOutput*), it will give us the datum.

```
outputDatum :: GameDatum
outputDatum = case gameDatum ownOutput (`findDatum` info) of
    Nothing -> traceError "game output datum not found"
    Just d   -> d
```

The *checkNonce* function is for the case where the first player has won and wants to prove it by revealing their nonce. The first argument is hash that was originally sent, the second argument is the nonce that is being revealed.

For the *GameChoice*-typed parameter, we will be passing in the move made by player 2. This should be the same as the move made by player 1, and this is what this function will determine using the hash and the nonce.

In order to check the hash of the nonce concatenated with the *GameChoice*, we use a helper function to convert the *GameChoice* to a *ByteString*. Note that the use of the *cFirst* and *cSecond* in the *checkNonce* function could be swapped around, and the function would work just the same - the difference between the two is that one is a *GameChoice* and one is a *ByteString*.

```
checkNonce :: ByteString -> ByteString -> GameChoice -> Bool
checkNonce bs nonce cSecond = sha2_256 (nonce `concatenate` cFirst) == bs
  where
    cFirst :: ByteString
    cFirst = case cSecond of
        Zero -> bsZero'
        One  -> bsOne'
```

Finally, there is the question of what happens to the NFT once the game is over and there is no game address anymore. The way we have implemented it here, is that the NFT goes back to the first player. The first player needs it in the beginning to kick off the game and put the NFT into the correct UTxO, so it is reasonable to give it back the player 1 in the end.

To verify that this condition is met, we have created a helper function called *nftToFirst*.

```
nftToFirst :: Bool
nftToFirst = assetClassValueOf (valuePaidTo info $ gFirst game) (gToken game) == 1
```

Now that we have covered the helper functions, let's look at the conditions.

There is one condition that covers all the cases, and that is that the input we are validating must contain the NFT.

```
traceIfFalse "token missing from input" (assetClassValueOf (txOutValue ownInput) (gToken
↳ game) == 1) &&
```

After that, the rules depend on the situation.

```
case (dat, red) of
```

The first situation is the one where the second player has not yet moved, and they are just now making their move.

```
(GameDatum bs Nothing, Play c) ->
  traceIfFalse "not signed by second player" (txSignedBy info (gSecond game))
↳ &&
  traceIfFalse "first player's stake missing" (lovelaces (txOutValue ownInput) ==
↳ gStake game) &&
  traceIfFalse "second player's stake missing" (lovelaces (txOutValue ownOutput) == (2
↳ * gStake game)) &&
  traceIfFalse "wrong output datum" (outputDatum == GameDatum bs (Just c))
↳ &&
  traceIfFalse "missed deadline" (to (gPlayDeadline game) `contains`
↳ txInfoValidRange info) &&
  traceIfFalse "token missing from output" (assetClassValueOf (txOutValue
↳ ownOutput) (gToken game) == 1)
```

Here, the first part is the *GameDatum* and it contains the first player's hash and a *Nothing* which shows that the second player has not yet moved. The second part is the *GameRedeemer* and has been determined to be of type *Play GameChoice*. We assign the *GameChoice* part to *c* using pattern matching.

We check that the second player has signed the transaction.

```
traceIfFalse "not signed by second player" (txSignedBy info (gSecond game))
```

Then, we check that the first player's stake is present in the input.

```
traceIfFalse "first player's stake missing" (lovelaces (txOutValue ownInput) == gStake
↳ game)
```

The output should have the second player's stake added to the total stake.

```
traceIfFalse "second player's stake missing" (lovelaces (txOutValue ownOutput) == (2 *
↳ gStake game))
```

We now exactly what the datum of the output must be. It must be the same hash, plus the move made by the second player.

```
traceIfFalse "wrong output datum" (outputDatum == GameDatum bs (Just c))
```

The, the move must happen before the play deadline.

```
traceIfFalse "missed deadline" (to (gPlayDeadline game) `contains` txInfoValidRange info)
```

And finally, the NFT must be passed on in the output UTxO.

```
traceIfFalse "token missing from output" (assetClassValueOf (txOutValue ownOutput)
↳ (gToken game) == 1)
```

The second situation is where both players have moved, and the second player discovers that they have won. In order to prove that and get the winnings, they have to reveal their nonce.

So, the transaction must be signed by the first player, the nonce must indeed agree with the hash submitted earlier, it must be done before the reveal deadline, the input must contain both players' stakes and, finally, the NFT must go back to the first player.

```
(GameDatum bs (Just c), Reveal nonce) ->
  traceIfFalse "not signed by first player"    (txSignedBy info (gFirst game))
  ↳
  ↳ &&
  traceIfFalse "commit mismatch"                (checkNonce bs nonce c)
  ↳
  ↳ &&
  traceIfFalse "missed deadline"                (to (gRevealDeadline game) `contains`
  ↳ txInfoValidRange info) &&
  traceIfFalse "wrong stake"                    (lovelaces (txOutValue ownInput) == (2
  ↳ * gStake game)) &&
  traceIfFalse "NFT must go to first player"    nftToFirst
```

Next we have the case where the second player doesn't move within the deadline, and the first player is reclaiming their stake. Here, the first player must have signed the transaction, the play deadline must have passed, their stake must be present, and the NFT must go back to the first player.

```
(GameDatum _ Nothing, ClaimFirst) ->
  traceIfFalse "not signed by first player"    (txSignedBy info (gFirst game))
  ↳
  ↳ &&
  traceIfFalse "too early"                      (from (1 + gPlayDeadline game)
  ↳ `contains` txInfoValidRange info) &&
  traceIfFalse "first player's stake missing"    (lovelaces (txOutValue ownInput) ==
  ↳ gStake game) &&
  traceIfFalse "NFT must go to first player"    nftToFirst
```

Finally, the case where both players have moved and the first player has either lost or not revealed in time, so the second player is claiming the winnings. This time, the transaction must be signed by the second player, the reveal deadline must have passed, both players' stakes must be present, and the NFT must, as usual, go back to the first player.

```
(GameDatum _ (Just _), ClaimSecond) ->
  traceIfFalse "not signed by second player"    (txSignedBy info (gSecond game))
  ↳
  ↳ &&
  traceIfFalse "too early"                      (from (1 + gRevealDeadline game)
  ↳ `contains` txInfoValidRange info) &&
  traceIfFalse "wrong stake"                    (lovelaces (txOutValue ownInput) == (2
  ↳ * gStake game)) &&
  traceIfFalse "NFT must go to first player"    nftToFirst
```

These four cases are all the legitimate cases that we can have, so in all other cases we fail validation.

```
_ -> False
```

So now let's look at the rest of the on-chain code.

As usual, we define a data type that holds the information about the types the datum and redeemer.

```
data Gaming
instance Scripts.ScriptType Gaming where
  type instance DatumType Gaming = GameDatum
  type instance RedeemerType Gaming = GameRedeemer
```

And we define the *ByteStrings* that will be used to represent the two choices. These values are completely arbitrary - they just can't be the same as each other.

```
bsZero, bsOne :: ByteString
bsZero = "0"
bsOne  = "1"
```

Boilerplate to compile our parameterized *mkGameValidator* to Plutus code. We apply the three parameters, *Game* and the two *ByteStrings*. Remember that, we need to pass in these *ByteString* parameters because we can't refer to *ByteStrings* as string literals within Plutus.

```
gameInst :: Game -> Scripts.ScriptInstance Gaming
gameInst game = Scripts.validator @Gaming
  ($$(PlutusTx.compile [| mkGameValidator |]))
  `PlutusTx.applyCode` PlutusTx.liftCode game
  `PlutusTx.applyCode` PlutusTx.liftCode bsZero
  `PlutusTx.applyCode` PlutusTx.liftCode bsOne)
  $$ (PlutusTx.compile [| wrap |])
where
  wrap = Scripts.wrapValidator @GameDatum @GameRedeemer
```

The usual boilerplate for validator and address.

```
gameValidator :: Game -> Validator
gameValidator = Scripts.validatorScript . gameInst

gameAddress :: Game -> Ledger.Address
gameAddress = scriptAddress . gameValidator
```

Now, as preparation for the off-chain code, we will need to be able to find the right UTxO - the one that carries the NFT. To do this we will write a helper function called *findGameOutput*.

```
findGameOutput :: HasBlockchainActions s => Game -> Contract w s Text (Maybe (TxOutRef,
  ↳ TxOutTx, GameDatum))
findGameOutput game = do
  utxos <- utxoAt $ gameAddress game
  return $ do
    (oref, o) <- find f $ Map.toList utxos
    dat      <- gameDatum (txOutTxOut o) (`Map.lookup` txData (txOutTxTx o))
    return (oref, o, dat)
where
  f :: (TxOutRef, TxOutTx) -> Bool
  f (_, o) = assetClassValueOf (txOutValue $ txOutTxOut o) (gToken game) == 1
```

The *findGameOutput* function takes the *Game*, then uses the *Contract* monad to try to find the UTxO with the NFT. It returns a *Maybe*, because it may not find one. If we find it, we return a *Just* of a triple containing the transaction reference, the transaction itself, and the *GameDatum*.

First we get a list of all the UTxOs at the game address, then we use the *find* function, passing in a helper function *f*, which checks whether the output contains the NFT.

The *find* function is found in module *Data.List* and is defined as

```
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

This works for more general containers than just lists, but you can think of lists in this example. It gets a predicate for an element of the *Foldable* type - the list in this case, and also takes a container of *as* - again a list in this example, and returns a *Maybe a*.

The logic is that if it finds an element that satisfies the predicate, it will return it as a *Just*, otherwise it will return *Nothing*. For example

```
Prelude Data.List Week07.EvenOdd> find even [1 :: Int, 3, 5, 8, 11, 12]
Just 8

Prelude Data.List Week07.EvenOdd> find even [1 :: Int, 3, 5, 11]
Nothing
```

### The *firstGame* contract

We have two contracts, one for each of the players.

Each contract has its own params type. For the *firstGame* contract, we call this type *FirstParams*.

```
data FirstParams = FirstParams
  { fpSecond      :: !PubKeyHash
  , fpStake       :: !Integer
  , fpPlayDeadline :: !Slot
  , fpRevealDeadline :: !Slot
  , fpNonce       :: !ByteString
  , fpCurrency    :: !CurrencySymbol
  , fpTokenName   :: !TokenName
  , fpChoice      :: !GameChoice
  } deriving (Show, Generic, FromJSON, ToJSON, ToSchema)
```

We don't need a *fpFirst* field here, as the first player is the owner of the wallet, so we know their public key hash. But we need *fpSecond* and also the familiar fields for stake, play deadline and reveal deadline.

Then we need the nonce, the NFT (split into *fpCurrency* and *fpTokenName*), and finally the move that the player wants to make.

Now, for the contract

```
firstGame :: forall w s. HasBlockchainActions s => FirstParams -> Contract w s Text ()
firstGame fp = do
  ...
```

The first thing we do is to get our own public key hash.

```
pkh <- pubKeyHash <$> Contract.ownPubKey
```

Then we populate the fields of the game.

```
let game = Game
  { gFirst      = pkh
  , gSecond     = fpSecond fp
  , gStake      = fpStake fp
  , gPlayDeadline = fpPlayDeadline fp
  , gRevealDeadline = fpRevealDeadline fp
  , gToken      = AssetClass (fpCurrency fp, fpTokenName fp)
  }
```

The  $v$  value is our stake plus the NFT, which must both go into the UTxO.

```
let ...
  v = lovelaceValueOf (fpStake fp) <> assetClassValue (gToken game) 1
```

We then calculate the hash that we need to send as our disguised move.

```
let ...
  c = fpChoice fp
  bs = sha2_256 $ fpNonce fp `concatenate` if c == Zero then bsZero else bsOne
```

We then submit the transaction and wait as usual. The constraints are very simple. We just need to create a UTxI with the datum of our move (nothing yet for the second player), and the value  $v$  we defined above.

```
let ...
  tx = Constraints.mustPayToTheScript (GameDatum bs Nothing) v
ledgerTx <- submitTxConstraints (gameInst game) tx
void $ awaitTxConfirmed $ txId ledgerTx
logInfo @String $ "made first move: " ++ show (fpChoice fp)
```

And we wait for the play deadline slot, at which point the winner can be determined.

```
void $ awaitSlot $ 1 + fpPlayDeadline fp
```

Once the deadline passed, we get hold of the UTxO. If, at this point, the UTxO is not found, something has gone very wrong. We know that we have produced the UTxO, and the only thing that the second player should be able to do is create a new one.

```
m <- findGameOutput game
case m of
  Nothing      -> throwError "game output not found"
```

So, assuming we find it, the first case we define is the one where the second player hasn't moved. So we can use the *ClaimFirst* redeemer to get the stake back.

As lookups we need to provide the UTxO and the validator of the game.

```
Just (oref, o, dat) -> case dat of
  GameDatum _ Nothing -> do
    logInfo @String "second player did not play"
    let lookups = Constraints.unspentOutputs (Map.singleton oref o) <>
                  Constraints.otherScript (gameValidator game)
    tx' = Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toData_
    ClaimFirst)
    ledgerTx' <- submitTxConstraintsWith @Gaming lookups tx'
```

(continues on next page)



(continued from previous page)

```
void $ awaitTxConfirmed $ txId ledgerTx'
  logInfo @String "reclaimed stake"
```

The second case is that the second player did move, and they lost. In which case we must now reveal our nonce, which we do using the *Reveal* redeemer.

We must put an additional constraint that the transaction must be submitted before the reveal deadline has passed.

```
GameDatum _ (Just c') | c' == c -> do
  logInfo @String "second player played and lost"
  let lookups = Constraints.unspentOutputs (Map.singleton oref o)
  <- Constraints.otherScript (gameValidator game)
  tx' = Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toData $ _
  <- Reveal $ fpNonce fp) <- Constraints.mustValidateIn (to $ fpRevealDeadline fp)
  ledgerTx' <- submitTxConstraintsWith @Gaming lookups tx'
  void $ awaitTxConfirmed $ txId ledgerTx'
  logInfo @String "victory"
```

If the second player moved and won, there is nothing for use to do.

```
_ -> logInfo @String "second player played and won"
```

### The *secondGame* contract

The params for the second player are similar to those of the first player. This time we don't need the second player's public key hash, because that is ours, and we already know what it is. Instead we need the first player's public key hash. Also, we don't need the nonce.

```
data SecondParams = SecondParams
  { spFirst      :: !PubKeyHash
  , spStake      :: !Integer
  , spPlayDeadline :: !Slot
  , spRevealDeadline :: !Slot
  , spCurrency    :: !CurrencySymbol
  , spTokenName   :: !TokenName
  , spChoice      :: !GameChoice
  } deriving (Show, Generic, FromJSON, ToJSON, ToSchema)
```

First we get our own public key hash then we set up the game values, in a similar way as we did for the first player.

```
secondGame :: forall w s. HasBlockchainActions s => SecondParams -> Contract w s Text ()
secondGame sp = do
  pkh <- pubKeyHash <$> Contract.ownPubKey
  let game = Game
    { gFirst      = spFirst sp
    , gSecond     = pkh
    , gStake      = spStake sp
    , gPlayDeadline = spPlayDeadline sp
    , gRevealDeadline = spRevealDeadline sp
```

(continues on next page)

(continued from previous page)

```

    , gToken      = AssetClass (spCurrency sp, spTokenName sp)
  }

```

Now, we try to find the UTxO that contains the NFT

```
m <- findGameOutput game
```

If we don't find it, then there is nothing to do, but if we do find it...

```

case m of
  Just (oref, o, GameDatum bs Nothing) -> do
    logInfo @String "running game found"

```

Then we want to call the script with the *Play* redeemer.

We assign the NFT to *token*.

```
let token = assetClassValue (gToken game) 1
```

We now calculate the value that we must put in the new output. Remember, if we decide to play, we must consume the existing UTxO and create a new one at the same address. The first will contain the stake that the first player added, and now we must add our own stake, and we must keep the NFT in there.

```
let v = let x = lovelaceValueOf (spStake sp) in x <> x <> token
```

Next, our choice.

```
let c = spChoice sp
```

Then the constraints and their required lookups.

We must consume the existing UTxO using the *Play* redeemer with our choice

```
let tx = Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toData $ Play c) <>
```

And create a new UTxO with the updated datum (the same *bs*, but with our choice), and with the *v* that we computed.

```
Constraints.mustPayToTheScript (GameDatum bs $ Just c) v <>
```

And it must be done before the deadline passes.

```
Constraints.mustValidateIn (to $ spPlayDeadline sp)
```

For lookups, we need the UTxO, the validator, and, because we are producing a UTxO for the script, we need the script instance.

```

let lookups = Constraints.unspentOutputs (Map.singleton oref o)
  ↳ <>
    Constraints.otherScript (gameValidator game)
  ↳ <>
    Constraints.scriptInstanceLookups (gameInst game)

```

Then we do the usual thing, we submit, we wait for confirmation and we log.

```
ledgerTx <- submitTxConstraintsWith @Gaming lookups tx
let tid = txId ledgerTx
void $ awaitTxConfirmed tid
logInfo @String $ "made second move: " ++ show (spChoice sp)
```

Then we wait until the reveal deadline has passed.

```
void $ awaitSlot $ 1 + spRevealDeadline sp
```

And we again try to find the UTxO, which could now be a different one.

```
m' <- findGameOutput game
```

If *m'* is *Nothing* - in other words, if we did not find a UTxO, then that means that while we were waiting, the first player revealed and won. So there is nothing for us to do.

```
case m' of
  Nothing          -> logInfo @String "first player won"
```

However, if we do find the UTxO, it means the first player didn't reveal, which means that either they decided not to play, probably because they lost. In any case, we can now claim the winnings.

Our constraints are that we must spend the UTxO that we found after the deadline has passed, and we must hand back the NFT to the first player.

```
Just (oref', o', _) -> do
  logInfo @String "first player didn't reveal"
  let lookups' = Constraints.unspentOutputs (Map.singleton oref' o')
  <- <
    Constraints.otherScript (gameValidator game)
    tx' = Constraints.mustSpendScriptOutput oref' (Redeemer $ PlutusTx.toData_
->ClaimSecond) <-
    Constraints.mustValidateIn (from $ 1 + spRevealDeadline sp)
    <- <
    Constraints.mustPayToPubKey (spFirst sp) token
  ledgerTx' <- submitTxConstraintsWith @Gaming lookups' tx'
  void $ awaitTxConfirmed $ txId ledgerTx'
  logInfo @String "second player won"
```

If we didn't find the NFT, then there is nothing for use to do.

```
_ -> logInfo @String "no running game found"
```

That is all the code we need for the two on-chain contracts.

To make them more accessible, we define two *Endpoints*, one for the first player, and one for the second. And then we define a contract called *endpoints* which offers a choice between these two *Endpoints*, and recursively calls itself.

```
type GameSchema = BlockchainActions .\ Endpoint "first" FirstParams .\ Endpoint "second"
-> " SecondParams

endpoints :: Contract () GameSchema Text ()
endpoints = (first `select` second) >> endpoints
  where
```

(continues on next page)

(continued from previous page)

```
first = endpoint @"first" >>= firstGame
second = endpoint @"second" >>= secondGame
```

So this concludes the first version of the game - the version that does not use state machines.

Now, let's test it using the *EmulatorTrace* monad.

## Testing

The *test* function tests each of the four combinations by calling the *test'* function which takes the first and second players' choices respectively.

The *test'* function uses the *runEmulatorTraceIO'* variant which allows us to set up the initial wallet distributions using an *EmulatorConfig*.

```
test :: IO ()
test = do
  test' Zero Zero
  test' Zero One
  test' One Zero
  test' One One

test' :: GameChoice -> GameChoice -> IO ()
test' c1 c2 = runEmulatorTraceIO' def emCfg $ myTrace c1 c2
  where
    emCfg :: EmulatorConfig
    emCfg = EmulatorConfig $ Left $ Map.fromList
      [ (Wallet 1, v <> assetClassValue (AssetClass (gameTokenCurrency,
        ↪gameTokenName)) 1)
        , (Wallet 2, v)
      ]

    v :: Value
    v = Ada.lovelaceValueOf 1000_000_000
```

As NFTs are not the focus of this lecture, we have conjured a test NFT out of thin air. In a real world scenario, we would need to mint a real NFT, using one of the methods we have seen before.

Now the trace. We pass the two game choices into the *myTrace* function.

```
myTrace :: GameChoice -> GameChoice -> EmulatorTrace ()
myTrace c1 c2 = do
  Extras.logInfo $ "first move: " ++ show c1 ++ ", second move: " ++ show c2
```

Then we start two instances of the contract, one for wallet 1 and one for wallet 2.

```
h1 <- activateContractWallet (Wallet 1) endpoints
h2 <- activateContractWallet (Wallet 2) endpoints
```

We look up the two public key hashes.

```
let pkh1 = pubKeyHash $ walletPubKey $ Wallet 1
    pkh2 = pubKeyHash $ walletPubKey $ Wallet 2
```

Then we define the parameters that we are going to use for the contracts. In reality *fpNonce* would be some random string, but here we just hardcode as "SECRETNONCE".

```
fp = FirstParams
  { fpSecond      = pkh2
  , fpStake       = 5000000
  , fpPlayDeadline = 5
  , fpRevealDeadline = 10
  , fpNonce       = "SECRETNONCE"
  , fpCurrency    = gameTokenCurrency
  , fpTokenName   = gameTokenName
  , fpChoice      = c1
  }
sp = SecondParams
  { spFirst       = pkh1
  , spStake       = 5000000
  , spPlayDeadline = 5
  , spRevealDeadline = 10
  , spCurrency    = gameTokenCurrency
  , spTokenName   = gameTokenName
  , spChoice      = c2
  }
```

And then we call the endpoints.

```
callEndpoint @"first" h1 fp
void $ Emulator.waitNSlots 3
callEndpoint @"second" h2 sp
void $ Emulator.waitNSlots 10
```

Now, we can run this test from the REPL.

```
cabal repl
Prelude Week07.StateMachine> :l Week07.Test
Prelude Week07.Test> test
```

## Test 1

The first scenario is that both play zero, so the first wallet should win.

```
Slot 00000: TxnValidate 9f7e753823edc9d69538ae9a03702708ccac2b9ae58b8426bcfcf99e274dd552
Slot 00000: SlotAdd Slot 1
Slot 00001: *** USER LOG: first move: Zero, second move: Zero
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Contract instance started
```

The first wallet creates the initial UTxO with its stake, and logs a message that it made the move.

```

Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "first"),("value",Object_
  ↳(fromList [("unEndpointValue",Object (fromList [("fpChoice",String "Zero"),("fpCurrency
  ↳",Object (fromList [("unCurrencySymbol",String "ff"))]),("fpNonce",String
  ↳"5345435245544e4f4e4345"),("fpPlayDeadline",Object (fromList [("getSlot",Number 5.
  ↳0))]),("fpRevealDeadline",Object (fromList [("getSlot",Number 10.0))]),("fpSecond",
  ↳Object (fromList [("getPubKeyHash",String
  ↳"39f713d0a644253f04529421b9f51b9b08979d08295959c4f3990ee617f5139f"))]),("fpStake",
  ↳Number 5000000.0),("fpTokenName",Object (fromList [("unTokenName",String "STATE TOKEN
  ↳")]))])))))))
Slot 00001: W1: TxSubmit:␣
  ↳6f41600a05f16728a64f9f227bd2e828a0ccbbf9b56f46503f06873d3e8906a6
Slot 00001: TxnValidate 6f41600a05f16728a64f9f227bd2e828a0ccbbf9b56f46503f06873d3e8906a6
Slot 00001: SlotAdd Slot 2
Slot 00002: *** CONTRACT LOG: "made first move: Zero"
Slot 00002: SlotAdd Slot 3
Slot 00003: SlotAdd Slot 4

```

While the first wallet is waiting, the second wallet kicks in and finds the UTxO, sees that it can make a move, and does so.

```

Slot 00004: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Receive endpoint call: Object (fromList [("tag",String "second"),("value",Object_
  ↳(fromList [("unEndpointValue",Object (fromList [("spChoice",String "Zero"),("spCurrency
  ↳",Object (fromList [("unCurrencySymbol",String "ff"))]),("spFirst",Object (fromList [(
  ↳"getPubKeyHash",String
  ↳"21fe31dfa154a261626bf854046fd2271b7bed4b6abe45aa58877ef47f9721b9"))]),("spPlayDeadline
  ↳",Object (fromList [("getSlot",Number 5.0))]),("spRevealDeadline",Object (fromList [(
  ↳"getSlot",Number 10.0))]),("spStake",Number 5000000.0),("spTokenName",Object (fromList_
  ↳[("unTokenName",String "STATE TOKEN"))]))])))))))
Slot 00004: *** CONTRACT LOG: "running game found"
Slot 00004: W2: TxSubmit:␣
  ↳9ff5cf1ce61c0395b653a57449c39ed14f06bb75600057ea0e32a8d1588d048e
Slot 00004: TxnValidate 9ff5cf1ce61c0395b653a57449c39ed14f06bb75600057ea0e32a8d1588d048e
Slot 00004: SlotAdd Slot 5
Slot 00005: *** CONTRACT LOG: "made second move: Zero"

```

The first player realizes that they have won, and so must reveal. And we see in the final balances that Wallet 1 does indeed have the NFT back and it also has almost 5 ada more than it started with. The difference is, of course, due to transaction fees. And the second wallet has a little more than 5 ada less.

```

Slot 00005: SlotAdd Slot 6
Slot 00006: *** CONTRACT LOG: "second player played and lost"
Slot 00006: W1: TxSubmit:␣
  ↳ea946a524a7a3959743fc4c5dbc3982bf1510a84d973fecbb660a328bb58c0b5
Slot 00006: TxnValidate ea946a524a7a3959743fc4c5dbc3982bf1510a84d973fecbb660a328bb58c0b5
Slot 00006: SlotAdd Slot 7
Slot 00007: *** CONTRACT LOG: "victory"
Slot 00007: SlotAdd Slot 8
Slot 00008: SlotAdd Slot 9
Slot 00009: SlotAdd Slot 10
Slot 00010: SlotAdd Slot 11
Slot 00011: *** CONTRACT LOG: "first player won"

```

(continues on next page)

(continued from previous page)

```

Slot 00011: SlotAdd Slot 12
Slot 00012: SlotAdd Slot 13
Slot 00013: SlotAdd Slot 14
Slot 00014: SlotAdd Slot 15
Final balances
Wallet 1:
  {, ""}: 1004999980
  {ff, "STATE TOKEN"}: 1
Wallet 2:
  {, ""}: 994999990

```

## Test 2

In the second case, Wallet 1 again plays Zero, but this time Wallet 2 plays One.

```

Slot 00000: TxnValidate 9fbe753823edc9d69538ae9a03702708ccac2b9ae58b8426bcfcf99e274dd552
Slot 00000: SlotAdd Slot 1
Slot 00001: *** USER LOG: first move: Zero, second move: One

```

The beginning is the same.

```

Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Contract instance started
Slot 00001: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "first"),("value",Object_
  ↳ (fromList [("unEndpointValue",Object (fromList [("fpChoice",String "Zero"),("fpCurrency
  ↳ ",Object (fromList [("unCurrencySymbol",String "ff")])),("fpNonce",String
  ↳ "5345435245544e4f4e4345"),("fpPlayDeadline",Object (fromList [("getSlot",Number 5.
  ↳ 0)])),("fpRevealDeadline",Object (fromList [("getSlot",Number 10.0)])),("fpSecond",
  ↳ Object (fromList [("getPubKeyHash",String
  ↳ "39f713d0a644253f04529421b9f51b9b08979d08295959c4f3990ee617f5139f"))]),("fpStake",
  ↳ Number 5000000.0),("fpTokenName",Object (fromList [("unTokenName",String "STATE TOKEN
  ↳ ")])])])])])])])
Slot 00001: W1: TxSubmit:_
  ↳ 6f41600a05f16728a64f9f227bd2e828a0ccbbf9b56f46503f06873d3e8906a6
Slot 00001: TxnValidate 6f41600a05f16728a64f9f227bd2e828a0ccbbf9b56f46503f06873d3e8906a6
Slot 00001: SlotAdd Slot 2
Slot 00002: *** CONTRACT LOG: "made first move: Zero"
Slot 00002: SlotAdd Slot 3
Slot 00003: SlotAdd Slot 4

```

Now the second wallet finds the game, and makes its move, but now the move is One.

```

Slot 00004: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 2}:
  Receive endpoint call: Object (fromList [("tag",String "second"),("value",Object_
  ↳ (fromList [("unEndpointValue",Object (fromList [("spChoice",String "One"),("spCurrency
  ↳ ",Object (fromList [("unCurrencySymbol",String "ff")])),("spFirst",Object (fromList [(
  ↳ "getPubKeyHash",String
  ↳ "21fe31dfa154a261626bf854046fd2271b7bed4b6abe45aa58877ef47f9721b9"))]),("spPlayDeadline
  ↳ ",Object (fromList [("getSlot",Number 5.0)])),("spRevealDeadline",Object (fromList [(
  ↳ "getSlot",Number 10.0)])),("spStake",Number 5000000.0),("spTokenName",Object (fromList_
  ↳ (continues on next page)
  ↳ [("unTokenName",String "STATE TOKEN")]))]))]))]

```

(continued from previous page)

```
Slot 00004: *** CONTRACT LOG: "running game found"
Slot 00004: W2: TxSubmit:␣
↳3200aab18d986869a7e9aa65ff45a635e0bc2dff9b04df26a0864355990f9c10
Slot 00004: TxnValidate 3200aab18d986869a7e9aa65ff45a635e0bc2dff9b04df26a0864355990f9c10
Slot 00004: SlotAdd Slot 5
Slot 00005: *** CONTRACT LOG: "made second move: One"
```

Now the first wallet realizes it has lost and does nothing.

```
Slot 00005: SlotAdd Slot 6
Slot 00006: *** CONTRACT LOG: "second player played and won"
Slot 00006: SlotAdd Slot 7
Slot 00007: SlotAdd Slot 8
Slot 00008: SlotAdd Slot 9
Slot 00009: SlotAdd Slot 10
Slot 00010: SlotAdd Slot 11
```

The second wallet detects that the deadline has passed without a reveal, and invokes the *ClaimSecond* endpoint to get the money. When we look at the final balances, Wallet 1 again has the NFT back, but the Ada balance situation is reversed.

```
Slot 00011: *** CONTRACT LOG: "first player didn't reveal"
Slot 00011: W2: TxSubmit:␣
↳a66744d7b4692db9457d9c3a5d832db7d1471299bd36ffe27827f41ec3e999f1
Slot 00011: TxnValidate a66744d7b4692db9457d9c3a5d832db7d1471299bd36ffe27827f41ec3e999f1
Slot 00011: SlotAdd Slot 12
Slot 00012: *** CONTRACT LOG: "second player won"
Slot 00012: SlotAdd Slot 13
Slot 00013: SlotAdd Slot 14
Slot 00014: SlotAdd Slot 15
Final balances
Wallet 1:
  {ff, "STATE TOKEN"}: 1
  {}, ""}: 994999990
Wallet 2:
  {}, ""}: 1004999980
```

The remaining two cases are very similar, so we won't post the logs here.

So, this all seems to work as expected.



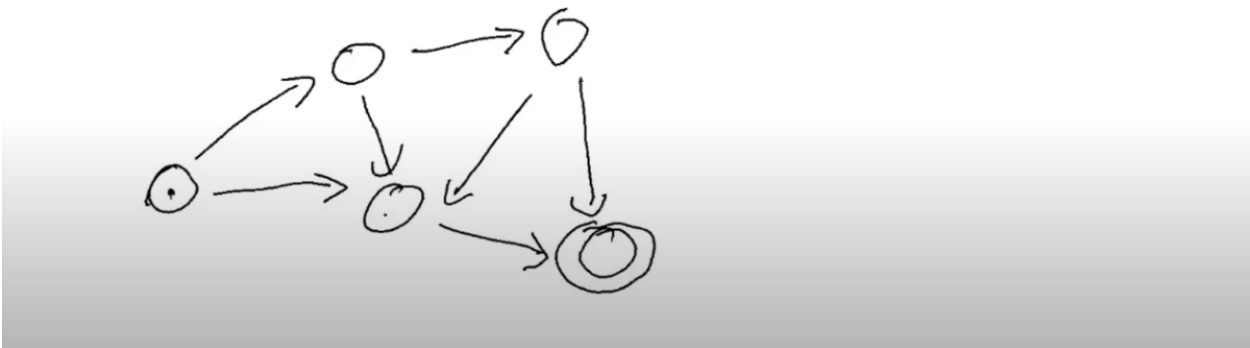
## 7.3 Code Example 2

Now we will rewrite this code using state machines.

### 7.3.1 What is a state machine?

A state machine has nothing in particular to do with blockchain. It is a system that starts with some sort of state, there are one or more transitions to other states, and from those states there are further transitions, and so on, like a directed graph. Some states can be *final* states, from which there can be no further transitions.

# State Machine



If we look again at how our games works, then we can consider it to be a state machine.

The initial state would be [Hash], where the first player has made the move.

From the initial state, there are two possible transitions. One where Bob plays, and the other where Bob does not play and Alice can reclaim.

In the diagram, all the nodes correspond to states, and all the arrows correspond to transitions.

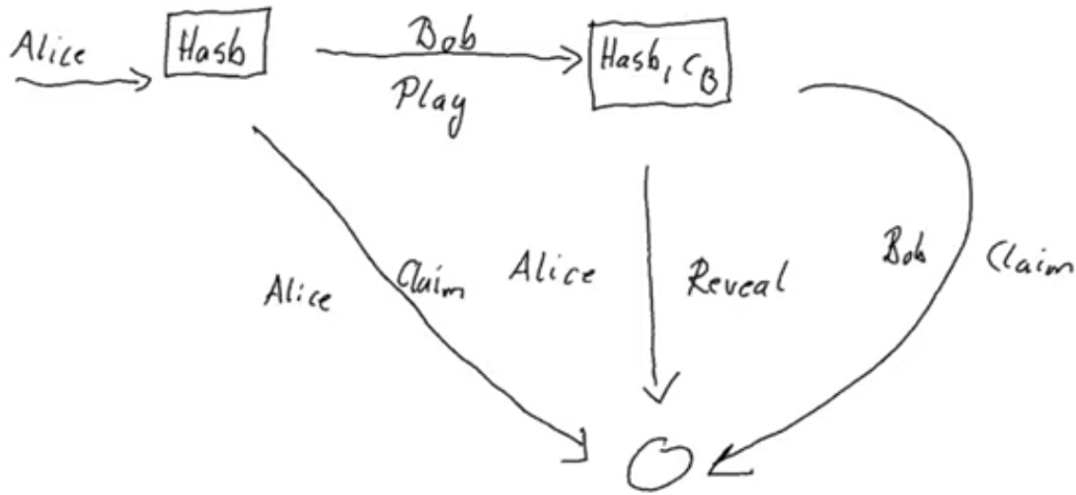
In the blockchain, the state machine will be represented by a UTxO sitting at the state machine address. The state of the machine will be the datum of that UTxO. A transition will be a transaction that consumes the current state, using a redeemer that characterizes the transition, and then produces a new UTxO at the same address, where the datum now reflects the new state.

This pattern fits a lot of situations very nicely, and there is special support in the Plutus libraries to implement such state machines. We will see that when we use this approach, our code will be much shorter.

The support for state machines is in the package *plutus-contract*, in module `Language.Plutus.Contract.StateMachine`

A `StateMachine` has two type parameters, *s* and *i*, which stand for state and input. These correspond to datum and redeemer, respectively.

It is a record type with four fields. Probably the most important one is *smTransition*, which defines which transitions can move which states which other states.



```
data StateMachine s i
```

# Source

Specification of a state machine, consisting of a transition function that determines the next state from the current state and an input, and a checking function that checks the validity of the transition in the context of the current transaction.

## Constructors

## StateMachine

<code>smTransition :: State s -&gt; i -&gt; Maybe (TxConstraints Void Void, State s)</code>	The transition function of the state machine. <code>Nothing</code> indicates an invalid transition from the current state.
<code>smFinal :: s -&gt; Bool</code>	Check whether a state is the final state
<code>smCheck :: s -&gt; i -&gt; ValidatorCtx -&gt; Bool</code>	The condition checking function. Can be used to perform checks on the pending transaction that aren't covered by the constraints. <code>smCheck</code> is always run in addition to checking the constraints, so the default implementation always returns true.

## Instances

<code>&gt; ScriptType (StateMachine s i)</code>	# Source
<code>&gt; type DatumType (StateMachine s i)</code>	# Source
<code>&gt; type RedeemerType (StateMachine s i)</code>	# Source

The *State s* type is basically the datum. It consists of the state itself and a value. Remember that the state of the state machine is represented by a UTxO, which has a datum and a value.

```
data State s # Source
```

Constructors

```
State
  stateData :: s
  stateValue :: Value
```

Instances

```
> Eq s => Eq (State s) # Source
> Show s => Show (State s) # Source
> Generic (State s) # Source
> ToJSON s => ToJSON (State s) # Source
> FromJSON s => FromJSON (State s) # Source
> type Rep (State s) # Source
```

Given the state type *s*, and a transaction that tries to consume this UTxO with a redeemer *i*, we can indicate that this transition is not allowed by returning *Nothing*. If it is allowed, we return a tuple.

The second component of the tuple is the new state (the new datum and value), which is the new UTxO sitting at the same address, with the first UTxO having been consumed.

The first component of the tuple specifies additional constraints that the transaction that does this must satisfy. Until now, we have only seen constraints in off-chain code.

We then have a function *setFinal* which is predicated on the state which tells us whether it is a final state or not. Final states are special in that the resulting *State* from the *setTransition* function must have no value attached to it, and the output does not get produced. The machine ends there.

The function *smCheck* is very similar to the *setTransition* function. It gets the datum, the redeemer and the context and returns a bool. It provides additional checks that can't be expressed by the *TxConstraints* in *setTransition*.

Finally, *smThreadToken* allows us to identify the UTxO which represents the current state. This is in the even that there us more than one UTxO sitting at the address of the state machine. It uses the same trick that we have seen before of using an NFT sitting in the value of the correct UTxO. You could, however, always return *Nothing* from *smThreadToken* and use some other mechanism to identify the correct UTxO.

The same game from example 1 has been implemented using a state machine, in the following module.

```
module Week07.StateMachine
```

The first parts of the code are the same - we have the same *Game* type and the same *GameChoice*. The first change we notice is with *GameDatum*.

We have added a second constructor to *GameDatum* called *Finished*. This will represent the final state of the state machine. It won't correspond to a UTxO, but we need it for the state machine mechanism to work.

```
data GameDatum = GameDatum ByteString (Maybe GameChoice) | Finished
deriving Show
```

And this adds a little more complexity to the definition of equality.

```
instance Eq GameDatum where
{-# INLINABLE (==) #-}
GameDatum bs mc == GameDatum bs' mc' = (bs == bs') && (mc == mc')
Finished          == Finished          = True
-                  == -                  = False
```

The redeemer is exactly the same as before. The *lovelaces* and *gameDatum* helper functions are also exactly the same as before.

Now we get to the *transition* function, which sort of corresponds to the *mkGameValidator* function that we used in the previous example. It is basically the core business logic.

```
transition :: Game -> State GameDatum -> GameRedeemer -> Maybe (TxConstraints Void Void,
↳State GameDatum)
```

The *transition* function takes the *Game*, then a *State GameDatum*, which, as we saw in the definition of *StateMachine*, is a pair consisting of the datum and the value. Thirdly, it takes the redeemer, and then returns a *Maybe* of the new state and constraints on the transaction.

Let's compare the *transition* function of the state machine with the *mkGameValidator* function from our first game version.

```
transition :: Game -> State GameDatum -> GameRedeemer -> Maybe (TxConstraints Void Void,
↳State GameDatum)
transition game s r = case (stateValue s, stateData s, r) of
  (v, GameDatum bs Nothing, Play c)
    | lovelaces v == gStake game          -> Just ( Constraints.mustBeSignedBy_
↳(gSecond game)                               <>
                                                    Constraints.mustValidateIn (to $_
↳gPlayDeadline game)                               , State (GameDatum bs $ Just c)_
                                                    , State (GameDatum bs $ Just c)_
↳(lovelaceValueOf $ 2 * gStake game)                )
    (v, GameDatum _ (Just _), Reveal _)
    | lovelaces v == (2 * gStake game)    -> Just ( Constraints.mustBeSignedBy_
↳(gFirst game)                               <>
                                                    Constraints.mustValidateIn (to $_
↳gRevealDeadline game)                               Constraints.mustPayToPubKey_
↳(gFirst game) token                               , State Finished mempty
                                                    )
    (v, GameDatum _ Nothing, ClaimFirst)
    | lovelaces v == gStake game          -> Just ( Constraints.mustBeSignedBy_
↳(gFirst game)                               <>
                                                    Constraints.mustValidateIn (from
↳$ 1 + gPlayDeadline game)                               Constraints.mustPayToPubKey_
↳(gFirst game) token                               , State Finished mempty
                                                    )
    (v, GameDatum _ (Just _), ClaimSecond)
    | lovelaces v == (2 * gStake game)    -> Just ( Constraints.mustBeSignedBy_
↳(gSecond game)                               <>
```

(continues on next page)

(continued from previous page)

```

-> $ 1 + gRevealDeadline game) <>
-> (gFirst game) token
                                , State Finished mempty
                                )
-> Nothing
-
where
  token :: Value
  token = assetClassValue (gToken game) 1

```

The first thing to notice is that, in the *transition* function we do not need to do our initial check for the presence of the NFT. This is because the state machine takes care of that, so long as we set the last field of the *StateMachine* to some NFT asset class.

```

-- we no longer need something like this for our state machine version
traceIfFalse "token missing from input" (assetClassValueOf (txOutValue ownInput) (gToken
-> game) == 1) &&

```

Let's remind ourselves how we defined the first case where the first player had moved, the second player had not yet moved, and now the second player wants to make a move. We had six conditions.

```

(GameDatum bs Nothing, Play c) ->
  traceIfFalse "not signed by second player" (txSignedBy info (gSecond game))
  &&
  traceIfFalse "first player's stake missing" (lovelaces (txOutValue ownInput) ==
-> gStake game)
  &&
  traceIfFalse "second player's stake missing" (lovelaces (txOutValue ownOutput) == (2
-> * gStake game))
  &&
  traceIfFalse "wrong output datum" (outputDatum == GameDatum bs (Just c))
  &&
  traceIfFalse "missed deadline" (to (gPlayDeadline game) `contains`
-> txInfoValidRange info)
  &&
  traceIfFalse "token missing from output" (assetClassValueOf (txOutValue
-> ownOutput) (gToken game) == 1)

```

Let's see how these conditions are reflected in the state machine version.

```

transition game s r = case (stateValue s, stateData s, r) of
  (v, GameDatum bs Nothing, Play c)
    | lovelaces v == gStake game -> Just ( Constraints.mustBeSignedBy (gSecond game)
->
  <>
                                Constraints.mustValidateIn (to $
-> gPlayDeadline game)
                                , State (GameDatum bs $ Just c)
-> (lovelaceValueOf $ 2 * gStake game)
                                )

```

We can access the value and datum components of our *State* parameter using *stateValue* and *stateData*, which gives us the triple for our *case* statement of value, datum and redeemer.

Our matching case is now

```
(v, GameDatum bs Nothing, Play c)
```

First we check that the number of lovelaces in the value matches the stake of the game, which was our second condition in our code for this case from example 1. If this condition is satisfied, we return a *Just* pair. The first component of the pair is the constraints on the transaction (formulated from the *Constraints* module that we know from off-chain code). The two constraints that comprise this part of the pair correspond to the first and fifth conditions in our old code.

The second component of the pair is the new state - the resulting UTxO - which again is given by datum and value. So here we are specifying that the datum of the new UTxO will contain both players' choices, and the value of the UTxO will contain both players' stakes. We leave the NFT out of this condition, even though it will be present in the UTxO, and that is again because the state machine implicitly takes care of this for us.

This second component corresponds with the third and fourth conditions from our old code.

The sixth condition from our old code related to the NFT which, as we have seen, we do not need to worry about.

Now let's compare the code from the second interesting case, where the second player has played and the first player sees that they have won.

```
-- old version
(GameDatum bs (Just c), Reveal nonce) ->
  traceIfFalse "not signed by first player"    (txSignedBy info (gFirst game))
  ↳                                             &&
  traceIfFalse "commit mismatch"                (checkNonce bs nonce c)
  ↳                                             &&
  traceIfFalse "missed deadline"                (to (gRevealDeadline game) `contains`
  ↳ txInfoValidRange info) &&
  traceIfFalse "wrong stake"                    (lovelaces (txOutValue ownInput) == (2 *
  ↳ gStake game)) &&
  traceIfFalse "NFT must go to first player"    nftToFirst
```

```
-- new version
(v, GameDatum _ (Just _), Reveal _)
| lovelaces v == (2 * gStake game) -> Just ( Constraints.mustBeSignedBy (gFirst game)
  ↳                                     <>
  ↳                                     Constraints.mustValidateIn (to $
  ↳ gRevealDeadline game) <>
  ↳                                     Constraints.mustPayToPubKey (gFirst game)
  ↳ token
  ↳                                     , State Finished mempty
  ↳                                     )
```

Again, we see that the first thing we do is to check that the correct stake exists, and if it does, we again return a *Just*. So, this takes care of condition four from the old code. Condition one from the old code is taken care of by the *Constraints.mustBeSignedBy* constraint in the new code.

Note that we do not check the nonce in the new code. The reason for this is that this check cannot be expressed in terms of a constraint. And this is exactly what the *smCheck* function is for, and we will see how this is used for this in a moment.

We can also match up the deadline check from each code sample, with it being defined using *Constraints.mustValidateIn* in the new code.

In the old code, when the game was over, we returned the NFT to the first player. In the new code, we also make sure the NFT goes back to the first player, but we also specify the *Finished* state and say that there is no money left in the contract using *mempty*.

Now we compare the old and the new code for the third case where the first player reclaims their stake when the second player does not play by the deadline.

```
-- old version
(GameDatum _ Nothing, ClaimFirst) ->
  traceIfFalse "not signed by first player"    (txSignedBy info (gFirst game))
  ↳
  ↳      &&
  traceIfFalse "too early"                      (from (1 + gPlayDeadline game))
  ↳
  ↳ `contains` txInfoValidRange info) &&
  traceIfFalse "first player's stake missing"    (lovelaces (txOutValue ownInput) ==
  ↳ gStake game) &&
  traceIfFalse "NFT must go to first player"    nftToFirst
```

```
-- new version
(v, GameDatum _ Nothing, ClaimFirst)
| lovelaces v == gStake game      -> Just ( Constraints.mustBeSignedBy (gFirst game)
  ↳
  ↳      <>
  ↳
  ↳      Constraints.mustValidateIn (from $ 1 +
  ↳ gPlayDeadline game) <>
  ↳
  ↳      Constraints.mustPayToPubKey (gFirst game)
  ↳ token
  ↳
  ↳      , State Finished mempty
  ↳
  ↳      )
```

These two match up fairly easily, with the lovelaces being the condition on the left in the new code, and the remaining conditions on the right in the new code matching up with corresponding conditions in the old code. Again we add the *Finished* state in the new code.

The last case, where the second player has played and the first player does not reveal by the deadline, probably because they lost.

```
(GameDatum _ (Just _), ClaimSecond) ->
  traceIfFalse "not signed by second player"    (txSignedBy info (gSecond game))
  ↳
  ↳      &&
  traceIfFalse "too early"                      (from (1 + gRevealDeadline game))
  ↳
  ↳ `contains` txInfoValidRange info) &&
  traceIfFalse "wrong stake"                    (lovelaces (txOutValue ownInput) == (2 *
  ↳ * gStake game)) &&
  traceIfFalse "NFT must go to first player"    nftToFirst
```

```
(v, GameDatum _ (Just _), ClaimSecond)
| lovelaces v == (2 * gStake game) -> Just ( Constraints.mustBeSignedBy (gSecond
  ↳ game)
  ↳      <>
  ↳
  ↳      Constraints.mustValidateIn (from $ 1 +
  ↳ + gRevealDeadline game) <>
  ↳
  ↳      Constraints.mustPayToPubKey (gFirst
  ↳ game) token
  ↳
  ↳      , State Finished mempty
  ↳
  ↳      )
```

The conditions in the old and the new code for this last case can be matched up in a very similar way to those for the third case.

All other states with arbitrary transitions are invalid, and we indicate that by returning *Nothing*.

```
_ -> Nothing
```

In the end we see that while the conditions themselves may not be much shorter in the new version than those in the old version, we also see that we only need one helper function.

```
token :: Value
token = assetClassValue (gToken game) 1
```

But we are not yet finished defining the state machine. There are some other fields in the *StateMachine* record.

One is *smFinal*, which lets us define what the final states are. For us, it is just the *Finished* state. We define a helper function that we can use for this field.

```
final :: GameDatum -> Bool
final Finished = True
final _        = False
```

Another field to define is *smCheck*. Recall that this is where we can put conditions that cannot be expressed as *Constraints*. So this is where we can put our nonce check.

We define another helper function *check*, with two auxiliary *ByteString* parameters to represent the zero and one choices, for reasons that we have seen before. We also pass it the datum, redeemer and context, and it will return us a boolean.

We don't need the script context, but we need the datum to get the second player's choice (which the first player is claiming is the same as theirs), and the redeemer to get the nonce that the first player is claiming to have used. We can then check that the hash of the choice and the nonce match the original hash from the datum.

```
check :: ByteString -> ByteString -> GameDatum -> GameRedeemer -> ScriptContext -> Bool
check bsZero' bsOne' (GameDatum bs (Just c)) (Reveal nonce) _ =
    sha2_256 (nonce `concatenate` if c == Zero then bsZero' else bsOne') == bs
```

In all other situations, those that are not checking the revealed nonce, we don't need to perform any checks.

```
check _ _ _ _ _ = True
```

Now we can define our state machine.

```
gameStateMachine :: Game -> ByteString -> ByteString -> StateMachine GameDatum_
    ↳GameRedeemer
gameStateMachine game bsZero' bsOne' = StateMachine
    { smTransition = transition game
    , smFinal      = final
    , smCheck      = check bsZero' bsOne'
    , smThreadToken = Just $ gToken game
    }
```

Our old *mkGameValidator* can now be replaced by using machinery provided by the state machine. There is a *mkValidator* function which will take our state machine, generated by the *gameStateMachine* function and turn it into a validator with exactly the same type as we had in the old code.

```
mkGameValidator :: Game -> ByteString -> ByteString -> GameDatum -> GameRedeemer ->_
    ↳ScriptContext -> Bool
mkGameValidator game bsZero' bsOne' = mkValidator $ gameStateMachine game bsZero' bsOne'
```

In the old code we had this mechanism to bundle datum and redeemer into a *Gaming* type.



```
-- old code
data Gaming
instance Scripts.ScriptType Gaming where
  type instance DatumType Gaming = GameDatum
  type instance RedeemerType Gaming = GameRedeemer
```

But now we can define this as.

```
type Gaming = StateMachine GameDatum GameRedeemer
```

We also provide an alternate version of *gameStateMachine*, which doesn't take the two auxiliary *ByteStrings*. This won't work for on-chain code, but for off-chain code it works just fine.

```
gameStateMachine' :: Game -> StateMachine GameDatum GameRedeemer
gameStateMachine' game = gameStateMachine game bsZero bsOne
```

We have the same boilerplate as before for *gameInst*, *gameValidator* and *gameAddress*, which we won't copy again here.

The function *gameClient* is new. It is a *StateMachineClient*, and this is what we need to interact with our state machine from our wallet in the *Contract* monad.

```
data StateMachineClient s i
```

# Source

Client-side definition of a state machine.

#### Constructors

##### StateMachineClient

```
scInstance :: StateMachineInstance s i
```

The instance of the state machine, defining the machine's transitions, its final states and its check function.

```
scChooser :: [OnChainState s i] -> Either SMContractError (OnChainState s i)
```

A function that chooses the relevant on-chain state, given a list of all potential on-chain states found at the contract address.

As you can see from the definition, it contains a *StateMachineInstance*. And the *StateMachineInstance* in turn is just a *StateMachine* and the corresponding script instance.

```
data StateMachineInstance s i
```

# Source

#### Constructors

##### StateMachineInstance

```
stateMachine :: StateMachine s i
```

The state machine specification.

```
validatorInstance :: ScriptInstance (StateMachine s i)
```

The validator code for this state machine.

Once we have that, the *StateMachineClient* still needs to be bundled with a so called *chooser*, which is the mechanism, from the off-chain code, for finding the UTxO that represents our state machine. In general there will be a list of UTxOs at the address of the state machine, and *scChooser* is a function that specifies which to pick.

We don't have to worry about that, because we are using the NFT approach, which means that the choosing is taken care of for us automatically.

There is a function *mkStateMachineClient* that takes a *StateMachineInstance* and returns a *StateMachineClient*, and this uses the default implementation of the chooser. And this will do the right thing and pick the UTxO that contains

our NFT.

```
gameClient :: Game -> StateMachineClient GameDatum GameRedeemer
gameClient game = mkStateMachineClient $ StateMachineInstance (gameStateMachine' game)
↳ (gameInst game)
```

Now, *gameClient* can be used to interact with the state machine from off-chain code.

*FirstParams* is exactly the same, so we won't repeat it here.

There is one small nuisance. The state machine contracts provided by the state machine module have a specific constraint on the error type. One error type that works is *SMContractError*.

But we want to do what we did in the last lectures and always use *Text* as the error type. To achieve this we will use a helper function to convert the *SMContractError* type into a *Text* type. Recall that *show* will return a *String* and *pack* will convert a *String* into a *Text*.

```
mapError' :: Contract w s SMContractError a -> Contract w s Text a
mapError' = mapError $ pack . show
```

So now the first player contract becomes much shorter and more compact.

The beginning is the same.

```
firstGame :: forall w s. HasBlockchainActions s => FirstParams -> Contract w s Text ()
firstGame fp = do
  pkh <- pubKeyHash <$> Contract.ownPubKey
  let game = Game
      { gFirst      = pkh
      , gSecond     = fpSecond fp
      , gStake       = fpStake fp
      , gPlayDeadline = fpPlayDeadline fp
      , gRevealDeadline = fpRevealDeadline fp
      , gToken       = AssetClass (fpCurrency fp, fpTokenName fp)
      }

```

Now, we take the client along with some values that we get as before.

```
client = gameClient game
v      = lovelaceValueOf (fpStake fp)
c      = fpChoice fp
bs     = sha2_256 $ fpNonce fp `concatenate` if c == Zero then bsZero else bsOne
```

There is a function *runInitialise* that starts a state machine and creates a *UTxO* at the state machine address. It takes the client as its first argument and then it needs the initial datum and the initial value for the *UTxO* sitting at that address. And it will automatically put the NFT there as well.

```
void $ mapError' $ runInitialise client (GameDatum bs Nothing) v
logInfo @String $ "made first move: " ++ show (fpChoice fp)
```

Now, the state machine is setup and the first player has made their move.

We wait until the play deadline.

```
void $ awaitSlot $ 1 + fpPlayDeadline fp
```

In our first example, we defined a helper function *findGameOutput* to get the current *UTxO*, but this can now be done in a simpler way using *getOnChainState*.

```
getOnChainState :: (AsSMContractError e, IsData state, HasUtxoAt schema) => StateMachineClient state i -> Contract
schema e (Maybe (OnChainState state i, UtxoMap))
```

# Source

Get the current on-chain state of the state machine instance. Return *Nothing* if there is no state on chain. Throws an *SMContractError* if the number of outputs at the machine address is greater than one.

The function *getOnChainState* will return a *Just OnChainState* if it finds the state machine, or a *Nothing* if it does not find it.

```
type OnChainState s i = (TypedScriptTxOut (StateMachine s i), TypedScriptTxOutRef (StateMachine s i))
```

# Source

So what is *OnChainState*? It is a tuple consisting of *TypedScriptTxOut* and *TypedScriptTxOutRef*. This is similar to what *utxoAt* gives us, which was a map of *TxOutRef*s to *TxOut*s. This is similar in that it is an output and its reference, but it is this *Typed* version that we haven't seen before.

All that does is bundle what we know from before, *TxOut*, but additionally it provides the datum. You'll recall that in our off-chain code we always have to scramble and write helper functions to access the datum once we had found the UTXO. We had to look up the datum hash, which could fail, and so on. *TypedScriptTxOut* hides all this from us.

```
data TypedScriptTxOut a
```

# Source

A *TxOut* tagged by a phantom type: and the connection type of the output.

#### Constructors

```
IsData (DatumType a) => TypedScriptTxOut
```

```
tyTxOutTxOut :: TxOut
```

```
tyTxOutData :: DatumType a
```

#### Instances

```
Eq (DatumType a) => Eq (TypedScriptTxOut a)
```

# Source

```
ToJSON (DatumType a) => ToJSON (TypedScriptTxOut a)
```

# Source

```
(FromJSON (DatumType a), IsData (DatumType a)) => FromJSON (TypedScriptTxOut a)
```

# Source

```
m <- mapError' $ getOnChainState client
```

As before, we should never get *Nothing* for *m*.

```
case m of Nothing -> throwError "game output not found"
```

Now, we are only interested in the *TypedScriptTxOut* parameter, which we assign to *o*, and use it to lookup the datum using *tyTxOutData*.

```
Just ((o, _), _) -> case tyTxOutData o of
```

As before we have the two cases. Either the second player has moved, or they haven't moved.

If they haven't moved, we must reclaim. Earlier we had lots of code to setup the lookups and constraints that we needed. Now we only need one line, and the important function here is *runStep*, which creates and submits a transaction that will transition the state machine.

It takes as input the client and the redeemer. It then returns a *TransitionResult*, which we are not using in this example, but basically encodes whether it succeeded or failed.



Which means, that we can use *runStep* with just the client and redeemer to replace all the lookups, the constraints, the transaction submissions and the waiting.

The way it works it that the *transition* function is that all the necessary constraints have been defined as part of the state machine.

```

GameDatum _ Nothing -> do
  logInfo @String "second player did not play"
  void $ mapError' $ runStep client ClaimFirst
  logInfo @String "first player reclaimed stake"

```

The second case is that the first player did reveal, and we again use the *runStep* function to transition the state machine.

```

GameDatum _ (Just c') | c' == c -> do
  logInfo @String "second player played and lost"
  void $ mapError' $ runStep client $ Reveal $ fpNonce fp
  logInfo @String "first player revealed and won"

```

And in all other situations, the second player wins.

```

_ -> logInfo @String "second player played and won"

```

The second player's contract is very similar, and just as simple.

```

secondGame :: forall w s. HasBlockchainActions s => SecondParams -> Contract w s Text ()
secondGame sp = do
  pkh <- pubKeyHash <$> Contract.ownPubKey
  let game = Game
    { gFirst      = spFirst sp
    , gSecond     = pkh
    , gStake      = spStake sp
    , gPlayDeadline = spPlayDeadline sp
    , gRevealDeadline = spRevealDeadline sp
    , gToken      = AssetClass (spCurrency sp, spTokenName sp)
    }
  client = gameClient game
  m <- mapError' $ getOnChainState client
  case m of
    Nothing      -> logInfo @String "no running game found"
    Just ((o, _), _) -> case tyTxOutData o of

```

The only case we need to address is where we haven't played yet, and so should play. And, in order to play, we again use the *runStep* function.

```

GameDatum _ Nothing -> do
  logInfo @String "running game found"
  void $ mapError' $ runStep client $ Play $ spChoice sp
  logInfo @String $ "made second move: " ++ show (spChoice sp)

```

We then wait until the reveal deadline has passed, then get the new state.

```

void $ awaitSlot $ 1 + spRevealDeadline sp
m' <- mapError' $ getOnChainState client
case m' of

```

If there is no state, the first player has won and claimed their winnings.

```

Nothing -> logInfo @String "first player won"

```

Otherwise, we have won, and we claim our winnings using the *runStep* function, giving the *ClaimSecond* redeemer.

```

Just _ -> do
  logInfo @String "first player didn't reveal"
  void $ mapError' $ runStep client ClaimSecond
  logInfo @String "second player won"

```

And a final catch all.

```

_ -> throwError "unexpected datum"

```

That concludes the state machine version of the code.

What is particularly nice about this approach is that we don't need to replicate logic anymore. We have discussed how off-chain code is used for construction and on-chain code is used for checking. When we use the state machine approach, we define logic that can be used for both, so we do not need to write it explicitly for the off-chain part and the on-chain part of the code.

In order to test this, there is a module called *Week07.TestStateMachine* in the example code. It is exactly the same as the test for the old code, with one exception, and that is just that instead of importing *Week07.EvenOdd*, it imports *Week07.StateMachine*. This is a quick and dirty way of doing things - we could, of course, have written a script that was parameterized over the contract we want to use.

If you load *Week07.TestStateMachine* in the REPL and run *test*, you should get exactly the same results as before.

## 7.4 Conclusion

State machines are not always appropriate, but when they are, you should definitely use them. They dramatically reduce the amount of code you have to write, and also reduce sources of errors.

The state machine mechanism automatically ensures that you have on-chain and off-chain code that are working correctly together. Until now, we have always had to take care of that ourselves.



## WEEK 08 - PROPERTY BASED TESTING

---

**Note:** This is a written version of [Lecture #8](#).

In this lecture we cover another state machine example, automatic testing using emulator traces, optics, and property-based testing.

This week we were using Plutus commit `ae35c4b8fe66dd626679bd2951bd72190e09a123`, the same commit as we used in the last lecture.

---

### 8.1 Token Sale

In the last lecture we looked at state machines, and saw how they often allow us to write much less code to express the logic of a smart contract, partly because there is a lot of sharing between on-chain and off-chain code and partly because a lot of boilerplate is encapsulated in the state machine machinery.

In this lecture we will see another example using a state machine, because the concept is very important. We will also take a look at testing. First we will look at the code, then we will explore various ways to go about testing.

The example we will use is a contract that allows somebody to sell tokens. The idea is that someone call lock some tokens in a contract, set a price, and then other people can buy them.

To begin, the seller starts with an NFT. It can be an arbitrary NFT and it will just be used, as before, to identify the correct UTxO that contains the contract state.

The first step is to lock the NFT at the script address of the smart contract that we are about to write. We'll call that contract *TS* for Token Sale. As a datum, we will use a simple integer, which will represent the price of the token we are selling, and this will start off as zero.

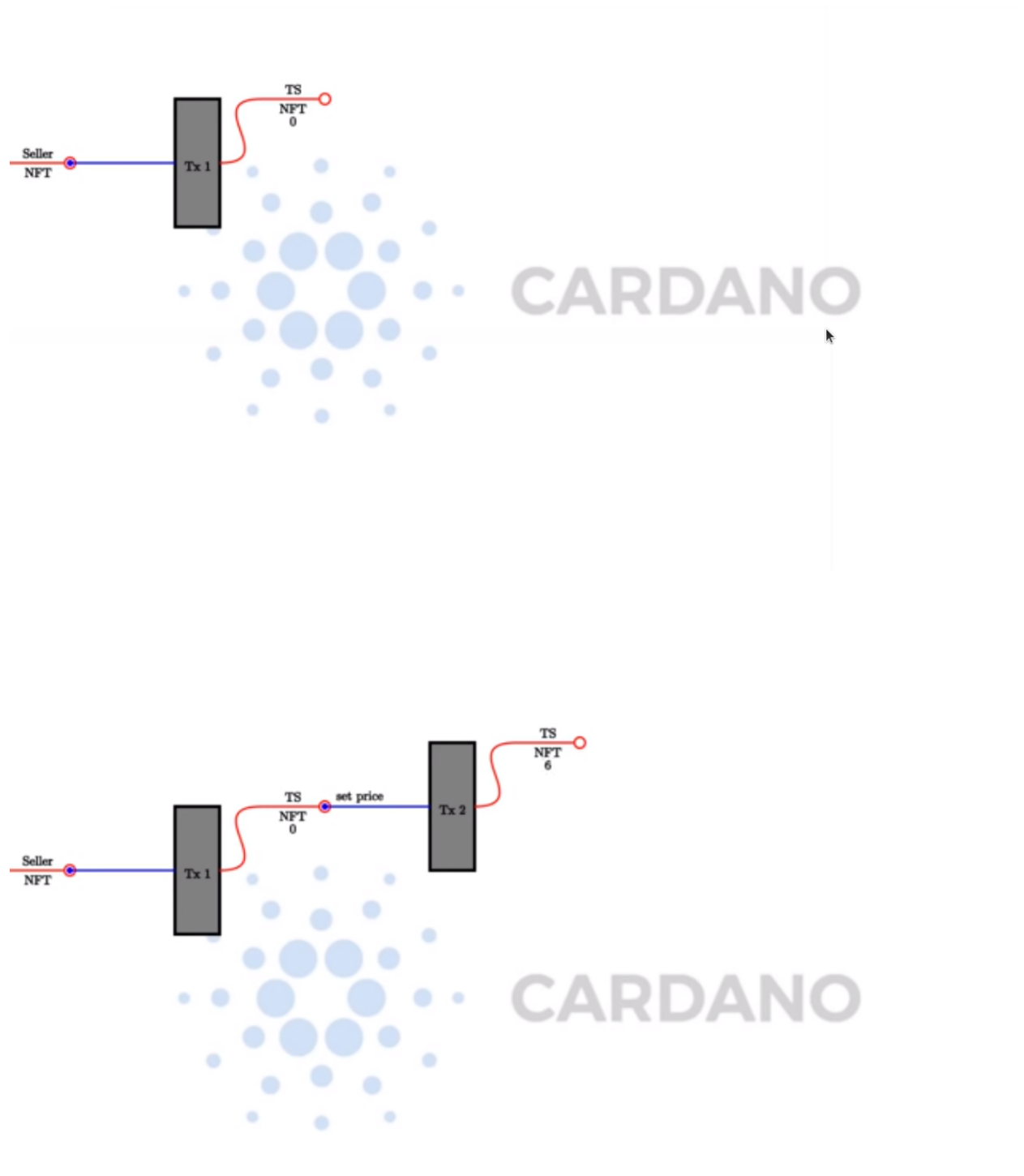
There will be several operations that the seller can do. One of those will be setting the price to a different value. In order to do that the seller will submit a transaction which has the current UTxO as input and the updated UTxO as output, where the datum has been changed to a different price per token.

Another thing that the seller can do is to lock some tokens in the contract. In order to do that they have to create another transaction which has as input the UTxO of the contract and a UTxO containing some tokens and, as output, the updated UTxO at the contract address which now contains the provided tokens.

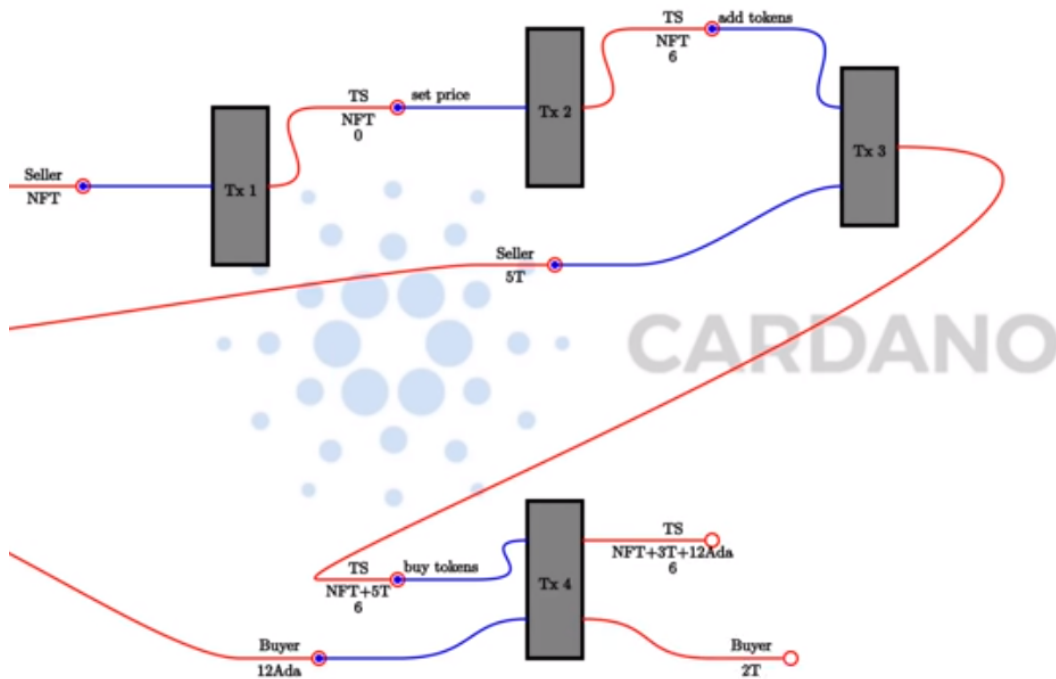
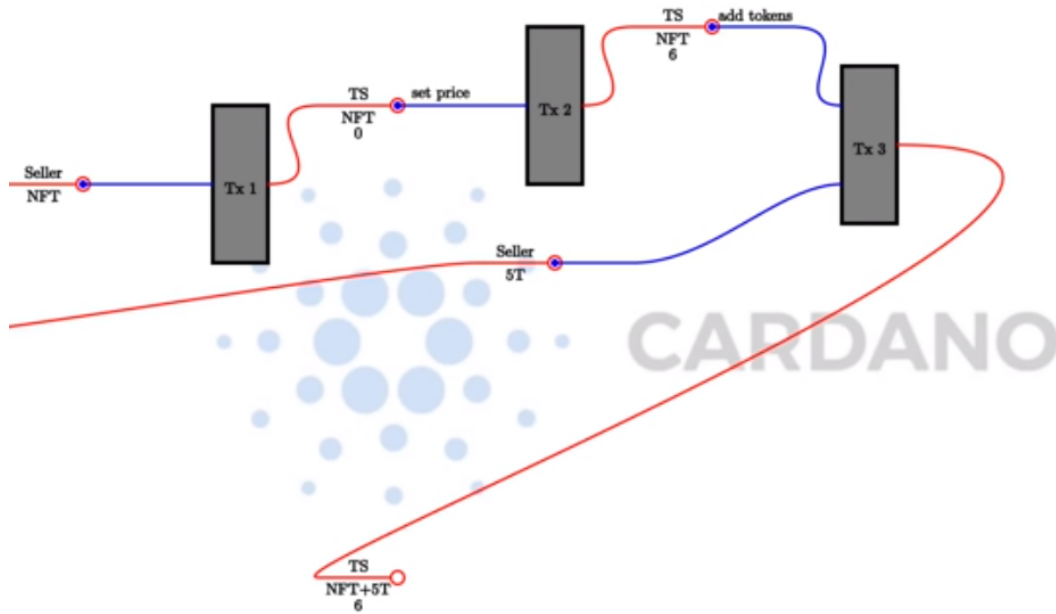
In this example, the seller provides five tokens to the contract.

In order to buy tokens, there needs to be a transaction created by the buyer. This transaction has as input the UTxO sitting at the TS script address, and the buying price in Ada.

So, if a buyer wants to buy two tokens, they will create a transaction that has, as input, 12 Ada, and the UTxO at the script address. Then, two outputs. One the updated contract state where now the tokens are taken out and the Ada has been added, and one output going to the buyer with the tokens that they have just bought.







Finally, there must be a way for the seller to retrieve tokens and Ada. In this example if, after the sale, the seller wants to retrieve all the Ada and one token, they would create a transaction that, again, has the script UTxO as input, and, as output, the updated script UTxO with the reduced balances, and one to themselves with the retrieved funds.

The diagram just shows one scenario, but these operations can be performed in any order - tokens can be added, the price can be changed, tokens can be bought, and so on, in an arbitrary order.

### 8.1.1 On-chain code

This week's first example is implemented in

```
module Week08.TokenSale
```

Let's first look at the type that we will use as the parameter that we will use for the contract.

```
data TokenSale = TokenSale
  { tsSeller :: !PubKeyHash
  , tsToken  :: !AssetClass
  , tsNFT    :: !AssetClass
  } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq, Prelude.Ord)
```

This has three fields - the seller's public key has, the token being sold, and the NFT used to identify the UTxO.

For the redeemer, we provide exactly the operations we saw in the diagram

```
data TSRedeemer =
  SetPrice Integer           -- the price
| AddTokens Integer         -- the number of tokens to add
| BuyTokens Integer         -- the number of tokens to buy
| Withdraw Integer Integer -- first argument is the number of tokens, the second is
-- the number of lovelace
  deriving (Show, Prelude.Eq)
```

Again we have the helper function that we have used in previous examples

```
lovelaces :: Value -> Integer
lovelaces = Ada.getLovelace . Ada.fromValue
```

Now, we get to the *transition* function of the state machine. We see the *TokenSale* parameter which holds the state machines configuration values, the *State* object with an *Integer* value to represent the price of the token, then the redeemer *TsRedeemer*. Again, we return a *Maybe*, which will be *Nothing* if the corresponding transition is illegal, or, if it is legal, a *Just* containing constraints and the new state.

```
transition :: TokenSale -> State Integer -> TSRedeemer -> Maybe (TxConstraints Void Void,
-- State Integer)
transition ts s r = case (stateValue s, stateData s, r) of
```

If the *SetPrice* redeemer is provided, then we only consider it to be legal if the price is not negative. We then return a *Just* with the constraint that the transaction must be signed by the token seller, and with the new state. The new state will be the new price *p*, and the *Value* in the contract remains the same, except for one thing.

It is a little unfortunate, but there is a discrepancy between the *v* on the left and the *v* on the right. On the left it does not contain the NFT, but on the right it does not. So, even though we want to say that we don't want the value changed, in fact we have to remove the NFT, because the Plutus libraries will add it again. This is perhaps not an ideal design, but that is how it currently is.

```
(v, _, SetPrice p) | p >= 0 -> Just ( Constraints.mustBeSignedBy (tsSeller ts)
    , State p $
    v <>
    nft (negate 1)
    )
```

We use a helper function to reference the NFT.

```
nft :: Integer -> Value
nft = assetClassValue (tsNFT ts)
```

When adding tokens, we could check that the seller has signed the transaction, but this contract would be provided by the seller, and the seller doesn't mind if someone wants to give them a free gift! Therefore, once we have the *AddTokens* redeemer and  $n$  is greater than zero, we are happy to return the new state without constraints.

The state that we return is untouched, except for the unfortunate trick we need to do with the NFT, and the addition of the new tokens.

```
(v, p, AddTokens n) | n > 0 -> Just ( mempty
    , State p $
    v <>
    nft (negate 1) <>
    assetClassValue (tsToken ts) n
    )
```

For the *BuyTokens* redeemer, again we check the number of tokens is positive, and again we don't need any constraints, because anybody can buy tokens.

For the new state, we don't touch the price. We again correct for the NFT. Then we subtract the tokens that were bought, and we add the lovelace that were paid for them.

```
(v, p, BuyTokens n) | n > 0 -> Just ( mempty
    , State p $
    v <>
    nft (negate 1) <>
    assetClassValue (tsToken ts) (negate n) <>
    lovelaceValueOf (n * p)
    )
```

Finally, for *Withdraw*, we insist that the token amount and the lovelace amount are both nonnegative. This time we again add a constraint that the seller must sign the transaction. We modify the state in a similar way to the way we did for the *BuyTokens* redeemer, but this time we adjust the token and lovelace amounts according to how much has been withdrawn.

```
(v, p, Withdraw n l) | n >= 0 && l >= 0 -> Just ( Constraints.mustBeSignedBy (tsSeller ts)
    , State p $
    v <>
    nft (negate 1)
    assetClassValue (tsToken ts) (negate n)
    lovelaceValueOf (negate l)
    )
```

All other state transitions are illegal.

```
_ -> Nothing
```

In this example we are able to construct our state machine more simply than we could in the previous lecture. This is because, in the previous lecture we had one condition that could not be expressed in the regular constraints.

In these situations, there is a helper function called *mkStateMachine* that takes three arguments. The first one is the state token, the second is the transition function. The last one is to indicate which states are final. In this case, there is no final state. Once this token sale has been setup, it will always be there.

```
tsStateMachine :: TokenSale -> StateMachine Integer TSRedeemer
tsStateMachine ts = mkStateMachine (Just $ tsNFT ts) (transition ts) (const False)
```

We can now use the usual boilerplate to turn it into a Plutus smart contract.

```
type TS = StateMachine Integer TSRedeemer

tsInst :: TokenSale -> Scripts.ScriptInstance TS
tsInst ts = Scripts.validator @TS
    ($$(PlutusTx.compile [|| mkTSValidator ||]) `PlutusTx.applyCode` PlutusTx.liftCode_
    ↪ ts)
    $$(PlutusTx.compile [|| wrap ||])
    where
        wrap = Scripts.wrapValidator @Integer @TSRedeemer

tsValidator :: TokenSale -> Validator
tsValidator = Scripts.validatorScript . tsInst

tsAddress :: TokenSale -> Ledger.Address
tsAddress = scriptAddress . tsValidator

tsClient :: TokenSale -> StateMachineClient Integer TSRedeemer
tsClient ts = mkStateMachineClient $ StateMachineInstance (tsStateMachine ts) (tsInst ts)
```

There are two helper functions to convert specialised error types to *Text*.

```
mapErrorC :: Contract w s C.CurrencyError a -> Contract w s Text a
mapErrorC = mapError $ pack . show

mapErrorSM :: Contract w s SMContractError a -> Contract w s Text a
mapErrorSM = mapError $ pack . show
```

### 8.1.2 Off-chain code

For the off-chain code, we start by defining a constant for the token name of the NFT.

```
nftName :: TokenName
nftName = "NFT"
```

The first contract we define is to start the token sale. This contract is designed to be invoked by the seller.

This first argument is a *Maybe CurrencySymbol*. The idea here is that if you pass in *Nothing*, the contract will mint a new NFT. Alternatively, you can provide a *Just CurrencySymbol* if the token already exists. We have done it this way mainly to make testing easier.

The *AssetClass* argument is the token the seller wants to trade.

For the return type, we are using the writer monad type with the *Last* type. The idea is that once the token sale has been setup, it will get written here so that other contracts are able to discover it. In addition, we return the created token sale.

To begin, we lookup the seller's public key hash. We then need to get hold of the NFT. So, we determine if we need to mint the NFT, and, if we do, we mint it, otherwise we just use the one that was passed into the function.

```
startTS :: HasBlockchainActions s => Maybe CurrencySymbol -> AssetClass -> Contract_
  ↳ (Last TokenSale) s Text TokenSale
startTS mcs token = do

    pkh <- pubKeyHash <$> Contract.ownPubKey
    cs <- case mcs of
        Nothing -> C.currencySymbol <$> mapErrorC (C.forgeContract pkh [(nftName, 1)])
        Just cs' -> return cs'
```

And now we can define the *TokenSale* and create the state machine client.

```
let ts = TokenSale
    { tsSeller = pkh
    , tsToken   = token
    , tsNFT     = AssetClass (cs, nftName)
    }
client = tsClient ts
```

We then use the *runInitialise* function that we discussed in the last lecture, using the client, an initial price of zero, and no initial funds, except for the NFT which will be automatically added.

We write the *ts* into the log, then log a message, and return the *ts*.

```
void $ mapErrorSM $ runInitialise client 0 mempty
tell $ Last $ Just ts
logInfo $ "started token sale " ++ show ts
return ts
```

The functions for all the other operations are extremely short. This example is ideal for the state machine approach.

They are all very similar. They all invoke *runStep* and then invoke the correct transition from the state machine.

For example, for *setPrice*, we need the *TokenSale* argument to identify the correct contract and the new value of the price. Then we use *runStep* using the client and *SetPrice* as the redeemer. We wrap that using *mapErrorSM* to convert to *Text* error messages, and we ignore the result.

```
setPrice :: HasBlockchainActions s => TokenSale -> Integer -> Contract w s Text ()
setPrice ts p = void $ mapErrorSM $ runStep (tsClient ts) $ SetPrice p
```

The remaining three follow the same pattern.

```
addTokens :: HasBlockchainActions s => TokenSale -> Integer -> Contract w s Text ()
addTokens ts n = void (mapErrorSM $ runStep (tsClient ts) $ AddTokens n)

buyTokens :: HasBlockchainActions s => TokenSale -> Integer -> Contract w s Text ()
buyTokens ts n = void $ mapErrorSM $ runStep (tsClient ts) $ BuyTokens n
```

(continues on next page)

(continued from previous page)

```
withdraw :: HasBlockchainActions s => TokenSale -> Integer -> Integer -> Contract w s
  ↳ Text ()
withdraw ts n l = void $ mapErrorSM $ runStep (tsClient ts) $ Withdraw n l
```

Now we define three schemas.

One for the seller which just has one endpoint which takes the *CurrencySymbol* and the *TokenName* of the asset to be traded.

```
type TSStartSchema = BlockchainActions
  ↳ Endpoint "start" (CurrencySymbol, TokenName)
```

For testing purposes, we create *TSStartSchema'* which additionally takes the *CurrencySymbol* of the NFT.

```
type TSStartSchema' = BlockchainActions
  ↳ Endpoint "start" (CurrencySymbol, CurrencySymbol, TokenName)
```

Lastly we have a *use* schema, with endpoints for the four operations - set price, add tokens, buy tokens and withdraw.

```
type TSUseSchema = BlockchainActions
  ↳ Endpoint "set price" Integer
  ↳ Endpoint "add tokens" Integer
  ↳ Endpoint "buy tokens" Integer
  ↳ Endpoint "withdraw" (Integer, Integer)
```

Now to implement the start endpoint. It simply calls *startTs'* and recurses. *startTs'* blocks until the parameters are provided and then calls *startTs* with *Nothing*, indicating that the NFT has to be minted. We wrap it in *handleError* and if there is an error, we simply log that error.

```
startEndpoint :: Contract (Last TokenSale) TSStartSchema Text ()
startEndpoint = startTS' >> startEndpoint
  where
    startTS' = handleError logError $ endpoint @"start" >>= void . startTS Nothing .
  ↳ AssetClass
```

The *startEndpoint'* function is very similar, but we add the NFT parameter, as per *TSStartSchema'*.

```
startEndpoint' :: Contract (Last TokenSale) TSStartSchema' Text ()
startEndpoint' = startTS' >> startEndpoint'
  where
    startTS' = handleError logError $ endpoint @"start" >>= \(cs1, cs2, tn) -> void $
  ↳ startTS (Just cs1) $ AssetClass (cs2, tn)
```

No surprises in the *use* endpoints. We give a choice between the four endpoints and just call the functions we defined earlier with the arguments fed in from the endpoint call, and with everything wrapped inside an error handler so that the contract won't crash in the event of an error.

```
useEndpoints :: TokenSale -> Contract () TSUseSchema Text ()
useEndpoints ts = (setPrice' `select` addTokens' `select` buyTokens' `select` withdraw')
  ↳ >>> useEndpoints ts
  where
    setPrice' = handleError logError $ endpoint @"set price" >>= setPrice ts
    addTokens' = handleError logError $ endpoint @"add tokens" >>= addTokens ts
```

(continues on next page)

(continued from previous page)

```
buyTokens' = handleError logError $ endpoint @"buy tokens" >>= buyTokens ts
withdraw'   = handleError logError $ endpoint @"withdraw"   >>= uncurry (withdraw ts)
```

### 8.1.3 Testing

In order to try it out, let's run it in the emulator.

We define a `runMyTrace` function which uses `runEmulatorTraceIO'` with a custom emulator configuration and a `myTrace` function.

```
runMyTrace :: IO ()
runMyTrace = runEmulatorTraceIO' def emCfg myTrace
```

Let's first look at the `emCfg` function. Recall that this is where we can give custom initial distributions to wallets. Here we give 1000 Ada and 1000 of a custom token to three wallets.

**Note:** The ability to use underscores in large numbers such as 1000\_000\_000 is provided by a GHC extension *NumericUnderscores*

```
emCfg :: EmulatorConfig
emCfg = EmulatorConfig $ Left $ Map.fromList [(Wallet w, v) | w <- [1 .. 3]]
  where
    v :: Value
    v = Ada.lovelaceValueOf 1000_000_000 <> assetClassValue token 1000

currency :: CurrencySymbol
currency = "aa"

name :: TokenName
name = "A"

token :: AssetClass
token = AssetClass (currency, name)
```

For the trace, first we activate Wallet 1 using the non-primed `startEndpoint` function which mints the NFT is minted automatically. Then, we call the start endpoint, giving it the symbol and name of the token we want to sell, and then wait for five slots, although two would be enough in this case.

```
myTrace :: EmulatorTrace ()
myTrace = do
  h <- activateContractWallet (Wallet 1) startEndpoint
  callEndpoint @"start" h (currency, name)
  void $ Emulator.waitNSlots 5
  Last m <- observableState h
```

We then read the state, which we wrote using `tell`, and check to see if it is valid. If it is not, we log an error. If it is, we proceed with the test.

```
case m of
  Nothing -> Extras.logError @String "error starting token sale"
```

(continues on next page)

(continued from previous page)

```
Just ts -> do
  Extras.logInfo $ "started token sale " ++ show ts
```

We can now activate the endpoints for the three wallets. Recall that the *useEndpoints* function is parameterised by the *TokenSale* data, which is why we needed to get that value.

```
h1 <- activateContractWallet (Wallet 1) $ useEndpoints ts
h2 <- activateContractWallet (Wallet 2) $ useEndpoints ts
h3 <- activateContractWallet (Wallet 3) $ useEndpoints ts
```

Wallet 1 sets the price to 1 Ada and we again wait for a generous amount of time.

```
callEndpoint @"set price" h1 1_000_000
void $ Emulator.waitNSlots 5
```

Wallet 1 adds 100 tokens.

```
callEndpoint @"add tokens" h1 100
void $ Emulator.waitNSlots 5
```

Wallet 2 buys 20 tokens. So now the contract should contain 80 tokens and 20 Ada.

```
callEndpoint @"buy tokens" h2 20
void $ Emulator.waitNSlots 5
```

Wallet 3 buys 5 tokens. Now there should be 75 tokens in the contract and 25 Ada.

```
callEndpoint @"buy tokens" h3 5
void $ Emulator.waitNSlots 5
```

Finally, Wallet 1 calls the withdraw endpoint, taking out 40 tokens and 10 Ada. At this point, there should be 35 tokens and 10 Ada in the contract.

```
callEndpoint @"withdraw" h1 (40, 10_000_000)
void $ Emulator.waitNSlots 5
```

Let's run this in the REPL.

```
cabal repl plutus-pioneer-program-week08-tests
Ok, five modules loaded.
Prelude Main> :l Spec.Trace
Ok, one module loaded.
Prelude Spec.Trace> runMyTrace

Slot 00000: TxnValidate 2125c8770581c6140c3c71276889f6353830744191de0184b6aa00b185004500
Slot 00000: SlotAdd Slot 1
Slot 00001: 000000000-0000-4000-8000-0000000000000 {Contract instance for wallet 1}:
  Contract instance started
```

The first endpoint call is to *start*. This creates three transaction. Two of these are from the forge contract to create the NFT, and the third one is to set up our initial UTxO for the token sale.

```
Slot 00001: 000000000-0000-4000-8000-0000000000000 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "start"),("value",Object_
  ↳ (fromList [("unEndpointValue",Array [Object (fromList [("unCurrencySymbol",String "Ada",
  ↳ (""))],Object (fromList [("unTokenName",String "A"))]]))]]))
```



(continued from previous page)

```

Slot 00001: W1: TxSubmit:
  ↳ ccba8b2abc3e82a735735c2346aa3fcac58152f17854b1745306e5b63a0b965
Slot 00001: TxnValidate ccba8b2abc3e82a735735c2346aa3fcac58152f17854b1745306e5b63a0b965
Slot 00001: SlotAdd Slot 2
Slot 00002: W1: TxSubmit:
  ↳ e23e19192aea3304a989ab98f05e70bc01fe43f3ea940da78a92ab7cebec9bbb
Slot 00002: TxnValidate e23e19192aea3304a989ab98f05e70bc01fe43f3ea940da78a92ab7cebec9bbb
Slot 00002: SlotAdd Slot 3
Slot 00003: W1: TxSubmit:
  ↳ 4cae1c5115eb4128243ce029dcd4d6c23d6497d3ab5e71a79f4dc34e9b8cd763
Slot 00003: TxnValidate 4cae1c5115eb4128243ce029dcd4d6c23d6497d3ab5e71a79f4dc34e9b8cd763
Slot 00003: SlotAdd Slot 4
Slot 00004: *** CONTRACT LOG: "started token sale TokenSale {tsSeller =
  ↳ 21fe31dfa154a261626bf854046fd2271b7bed4b6abe45aa58877ef47f9721b9, tsToken = (aa,\"A\"),
  ↳ tsNFT = (65b4199f7d025bfb3b065b0fb88a77d694ffd849ff740b1a4cc453bfaab30f55,\"NFT\")}"
Slot 00004: SlotAdd Slot 5
Slot 00005: SlotAdd Slot 6

```

We successfully read the *TokenSale* value from the observable state, and start the three contract instances for the use contract.

```

Slot 00006: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
  Sending contract state to Thread 0
Slot 00006: SlotAdd Slot 7
Slot 00007: *** USER LOG: started token sale TokenSale {tsSeller =
  ↳ 21fe31dfa154a261626bf854046fd2271b7bed4b6abe45aa58877ef47f9721b9, tsToken = (aa,"A"),
  ↳ tsNFT = (65b4199f7d025bfb3b065b0fb88a77d694ffd849ff740b1a4cc453bfaab30f55,"NFT")}
Slot 00007: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 1}:
  Contract instance started
Slot 00007: 00000000-0000-4000-8000-000000000002 {Contract instance for wallet 2}:
  Contract instance started
Slot 00007: 00000000-0000-4000-8000-000000000003 {Contract instance for wallet 3}:
  Contract instance started

```

Then we set the price.

```

Slot 00007: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "set price"),("value",Object
  ↳ (fromList [("unEndpointValue",Number 1000000.0))]))
Slot 00007: W1: TxSubmit:
  ↳ 2de6dd820e6939b4b1f9e162c0e2cc878cc38ea1231a9be610315da4eda06714
Slot 00007: TxnValidate 2de6dd820e6939b4b1f9e162c0e2cc878cc38ea1231a9be610315da4eda06714
Slot 00007: SlotAdd Slot 8
Slot 00008: SlotAdd Slot 9
Slot 00009: SlotAdd Slot 10
Slot 00010: SlotAdd Slot 11
Slot 00011: SlotAdd Slot 12

```

Then add some tokens.

```

Slot 00012: 00000000-0000-4000-8000-000000000001 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "add tokens"),("value",Object
  ↳ (fromList [("unEndpointValue",Number 100.0))]))

```

(continues on next page)

(continued from previous page)

```

Slot 00012: W1: TxSubmit:␣
→42f1bebe285d1ea23bd90683d110866bb438eede8ef62eaf5e9e3d65eec18e90
Slot 00012: TxnValidate 42f1bebe285d1ea23bd90683d110866bb438eede8ef62eaf5e9e3d65eec18e90
Slot 00012: SlotAdd Slot 13
Slot 00013: SlotAdd Slot 14
Slot 00014: SlotAdd Slot 15
Slot 00015: SlotAdd Slot 16
Slot 00016: SlotAdd Slot 17

```

Then the two buys by Wallets 2 and 3.

```

Slot 00017: 00000000-0000-4000-8000-0000000000002 {Contract instance for wallet 2}:
  Receive endpoint call: Object (fromList [("tag",String "buy tokens"),("value",Object␣
→(fromList [("unEndpointValue",Number 20.0)])))]))
Slot 00017: W2: TxSubmit:␣
→30d28ca855a14accbb11deee682b174adffb548922e1d4257242880f28328f8e
Slot 00017: TxnValidate 30d28ca855a14accbb11deee682b174adffb548922e1d4257242880f28328f8e
Slot 00017: SlotAdd Slot 18
Slot 00018: SlotAdd Slot 19
Slot 00019: SlotAdd Slot 20
Slot 00020: SlotAdd Slot 21
Slot 00021: SlotAdd Slot 22
Slot 00022: 00000000-0000-4000-8000-0000000000003 {Contract instance for wallet 3}:
  Receive endpoint call: Object (fromList [("tag",String "buy tokens"),("value",Object␣
→(fromList [("unEndpointValue",Number 5.0)])))]))
Slot 00022: W3: TxSubmit:␣
→708b0c4117ad3b38b69254a714e4695c574af404c3fff0eda859b571218b003c
Slot 00022: TxnValidate 708b0c4117ad3b38b69254a714e4695c574af404c3fff0eda859b571218b003c
Slot 00022: SlotAdd Slot 23
Slot 00023: SlotAdd Slot 24
Slot 00024: SlotAdd Slot 25
Slot 00025: SlotAdd Slot 26
Slot 00026: SlotAdd Slot 27

```

And finally, the withdraw by Wallet 1.

```

Slot 00027: 00000000-0000-4000-8000-0000000000001 {Contract instance for wallet 1}:
  Receive endpoint call: Object (fromList [("tag",String "withdraw"),("value",Object␣
→(fromList [("unEndpointValue",Array [Number 40.0,Number 1.0e7]])))]))
Slot 00027: W1: TxSubmit:␣
→a42a06cc3e3b1653ec4aba5ab8304484d778adcbddac2ceb9f639f7e4bd1dfd2
Slot 00027: TxnValidate a42a06cc3e3b1653ec4aba5ab8304484d778adcbddac2ceb9f639f7e4bd1dfd2
Slot 00027: SlotAdd Slot 28
Slot 00028: SlotAdd Slot 29
Slot 00029: SlotAdd Slot 30
Slot 00030: SlotAdd Slot 31
Slot 00031: SlotAdd Slot 32
Slot 00032: SlotAdd Slot 33

```

All wallets initially owned 1000 tokens and 1000 Ada. Wallet 1 added 100 tokens to the contract, but then in the last step retrieved 40 tokens and 10 Ada, and so we see its final balance as 940 tokens and 1010 Ada minus transaction fees.

```
Final balances
Wallet 1:
  {aa, "A"}: 940
  {, ""}: 1009942570
```

```
{aa, "A"}: 940
{, ""}: 1009942570
```

Wallet 2 bought 20 tokens and paid 20 Ada for them, plus some transaction fees.

```
{aa, "A"}: 1020  
{, ""}: 979985260
```

Wallet 3 bought 5 tokens for 5 Ada.

```
{aa, "A"}: 1005  
{, ""}: 994985211
```

Finally, the script still contains the NFT, which will forever stay there, plus 35 tokens and 15 Ada. There were, at one point, 75 tokens and 25 Ada, before Wallet 1 made a withdrawal.

```
{65b4199f7d025bfb3b065b0fb88a77d694ffd849ff7f40b1a4cc453bfaab30f55, "NFT": 1
{aa, "A"}: 35
{, ""}: 150000000
```

## 8.2 Unit Testing

### 8.2.1 Tasty

You can find *tasty* on Hackage.

Hackage :: [Package]

Search

Browse

What's new

tasty: Modern and extensible testing framework

[ library, mit, testing ]

[ Propose Tags ]

Tasty is a modern testing framework for Haskell. It lets you combine your unit tests, golden tests, QuickCheck/SmallCheck properties, and any other types of tests into a single test suite.

Skip to Readme

Documentation

Available

Modules

[index]

[Quick Jump]

Test

Test.Tasty

Test.Tasty.Ingredients

Test.Tasty.Ingredients.Basic

Test.Tasty.Ingredients.ConsoleReporter

Test.Tasty.Options

Patterns

Test.Tasty.Patterns.Eval

Test.Tasty.Patterns.Parser

Test.Tasty.Patterns.Types

Test.Tasty.Providers

Test.Tasty.Providers.ConsoleFormat

Test.Tasty.Runners

Versions

[R5S]

[faq]

0.1, 0.1.1, 0.2, 0.3, 0.3.1, 0.4, 0.4.0.1, 0.4.1.1, 0.4.2, 0.5, 0.5.1, 0.5.2, 0.5.2.1, 0.6, 0.7, 0.8, 0.8.0.2, 0.8.0.4, 0.8.1.1, 0.8.1.2, 0.8.1.3, 0.9.0.1, 1.0, 1.0.0.1, 1.0.0.2, 1.0.0.3, 1.0.0.4, 1.0.1, 1.0.1.1, 1.0.1.2, 1.1, 1.1.0.1, 1.1.0.2, 1.1.0.3, 1.1.0.4, 1.1.1, 1.1.1.2, 1.1.2.1, 1.1.2.2, 1.1.2.3, 1.1.2.4, 1.1.2.5, 1.1.3, 1.1.3.1, 1.1.3.2, 1.1.3.3, 1.1.3.4, 1.1.3.5, 1.1.3.6, 1.1.3.7, 1.1.3.8, 1.1.3.9, 1.1.3.10, 1.1.3.11, 1.1.3.12, 1.1.3.13, 1.1.3.14, 1.1.3.15, 1.1.3.16, 1.1.3.17, 1.1.3.18, 1.1.3.19, 1.1.3.20, 1.1.3.21, 1.1.3.22, 1.1.3.23, 1.1.3.24, 1.1.3.25, 1.1.3.26, 1.1.3.27, 1.1.3.28, 1.1.3.29, 1.1.3.30, 1.1.3.31, 1.1.3.32, 1.1.3.33, 1.1.3.34, 1.1.3.35, 1.1.3.36, 1.1.3.37, 1.1.3.38, 1.1.3.39, 1.1.3.40, 1.1.3.41, 1.1.3.42, 1.1.3.43, 1.1.3.44, 1.1.3.45, 1.1.3.46, 1.1.3.47, 1.1.3.48, 1.1.3.49, 1.1.3.50, 1.1.3.51, 1.1.3.52, 1.1.3.53, 1.1.3.54, 1.1.3.55, 1.1.3.56, 1.1.3.57, 1.1.3.58, 1.1.3.59, 1.1.3.60, 1.1.3.61, 1.1.3.62, 1.1.3.63, 1.1.3.64, 1.1.3.65, 1.1.3.66, 1.1.3.67, 1.1.3.68, 1.1.3.69, 1.1.3.70, 1.1.3.71, 1.1.3.72, 1.1.3.73, 1.1.3.74, 1.1.3.75, 1.1.3.76, 1.1.3.77, 1.1.3.78, 1.1.3.79, 1.1.3.80, 1.1.3.81, 1.1.3.82, 1.1.3.83, 1.1.3.84, 1.1.3.85, 1.1.3.86, 1.1.3.87, 1.1.3.88, 1.1.3.89, 1.1.3.90, 1.1.3.91, 1.1.3.92, 1.1.3.93, 1.1.3.94, 1.1.3.95, 1.1.3.96, 1.1.3.97, 1.1.3.98, 1.1.3.99, 1.1.3.100, 1.1.3.101, 1.1.3.102, 1.1.3.103, 1.1.3.104, 1.1.3.105, 1.1.3.106, 1.1.3.107, 1.1.3.108, 1.1.3.109, 1.1.3.110, 1.1.3.111, 1.1.3.112, 1.1.3.113, 1.1.3.114, 1.1.3.115, 1.1.3.116, 1.1.3.117, 1.1.3.118, 1.1.3.119, 1.1.3.120, 1.1.3.121, 1.1.3.122, 1.1.3.123, 1.1.3.124, 1.1.3.125, 1.1.3.126, 1.1.3.127, 1.1.3.128, 1.1.3.129, 1.1.3.130, 1.1.3.131, 1.1.3.132, 1.1.3.133, 1.1.3.134, 1.1.3.135, 1.1.3.136, 1.1.3.137, 1.1.3.138, 1.1.3.139, 1.1.3.140, 1.1.3.141, 1.1.3.142, 1.1.3.143, 1.1.3.144, 1.1.3.145, 1.1.3.146, 1.1.3.147, 1.1.3.148, 1.1.3.149, 1.1.3.150, 1.1.3.151, 1.1.3.152, 1.1.3.153, 1.1.3.154, 1.1.3.155, 1.1.3.156, 1.1.3.157, 1.1.3.158, 1.1.3.159, 1.1.3.160, 1.1.3.161, 1.1.3.162, 1.1.3.163, 1.1.3.164, 1.1.3.165, 1.1.3.166, 1.1.3.167, 1.1.3.168, 1.1.3.169, 1.1.3.170, 1.1.3.171, 1.1.3.172, 1.1.3.173, 1.1.3.174, 1.1.3.175, 1.1.3.176, 1.1.3.177, 1.1.3.178, 1.1.3.179, 1.1.3.180, 1.1.3.181, 1.1.3.182, 1.1.3.183, 1.1.3.184, 1.1.3.185, 1.1.3.186, 1.1.3.187, 1.1.3.188, 1.1.3.189, 1.1.3.190, 1.1.3.191, 1.1.3.192, 1.1.3.193, 1.1.3.194, 1.1.3.195, 1.1.3.196, 1.1.3.197, 1.1.3.198, 1.1.3.199, 1.1.3.200, 1.1.3.201, 1.1.3.202, 1.1.3.203, 1.1.3.204, 1.1.3.205, 1.1.3.206, 1.1.3.207, 1.1.3.208, 1.1.3.209, 1.1.3.210, 1.1.3.211, 1.1.3.212, 1.1.3.213, 1.1.3.214, 1.1.3.215, 1.1.3.216, 1.1.3.217, 1.1.3.218, 1.1.3.219, 1.1.3.220, 1.1.3.221, 1.1.3.222, 1.1.3.223, 1.1.3.224, 1.1.3.225, 1.1.3.226, 1.1.3.227, 1.1.3.228, 1.1.3.229, 1.1.3.230, 1.1.3.231, 1.1.3.232, 1.1.3.233, 1.1.3.234, 1.1.3.235, 1.1.3.236, 1.1.3.237, 1.1.3.238, 1.1.3.239, 1.1.3.240, 1.1.3.241, 1.1.3.242, 1.1.3.243, 1.1.3.244, 1.1.3.245, 1.1.3.246, 1.1.3.247, 1.1.3.248, 1.1.3.249, 1.1.3.250, 1.1.3.251, 1.1.3.252, 1.1.3.253, 1.1.3.254, 1.1.3.255, 1.1.3.256, 1.1.3.257, 1.1.3.258, 1.1.3.259, 1.1.3.260, 1.1.3.261, 1.1.3.262, 1.1.3.263, 1.1.3.264, 1.1.3.265, 1.1.3.266, 1.1.3.267, 1.1.3.268, 1.1.3.269, 1.1.3.270, 1.1.3.271, 1.1.3.272, 1.1.3.273, 1.1.3.274, 1.1.3.275, 1.1.3.276, 1.1.3.277, 1.1.3.278, 1.1.3.279, 1.1.3.280, 1.1.3.281, 1.1.3.282, 1.1.3.283, 1.1.3.284, 1.1.3.285, 1.1.3.286, 1.1.3.287, 1.1.3.288, 1.1.3.289, 1.1.3.290, 1.1.3.291, 1.1.3.292, 1.1.3.293, 1.1.3.294, 1.1.3.295, 1.1.3.296, 1.1.3.297, 1.1.3.298, 1.1.3.299, 1.1.3.300, 1.1.3.301, 1.1.3.302, 1.1.3.303, 1.1.3.304, 1.1.3.305, 1.1.3.306, 1.1.3.307, 1.1.3.308, 1.1.3.309, 1.1.3.310, 1.1.3.311, 1.1.3.312, 1.1.3.313, 1

There is also some example code on the same page.

Basically you have a main program that references some *tests* of type *TestTree*. As the name suggests, this allows for a tree of tests, where you can have sub groups and sub-sub groups and so on.

```
main = defaultMain tests

tests :: TestTree
tests = testGroup "Tests" [properties, unitTests]
```

There is special support for tests in Plutus in the *plutus-contract* package in

```
module Plutus.Contract.Test
```

There are various types of tests that are supported, but here we will only look at two of those. One that works with emulator traces, and one which is much more sophisticated and uses so-called property-based testing.

This module gives us functions for checking predicates, for example

```
checkPredicate :: String -> TracePredicate -> EmulatorTrace () -> TestTree
```

Here we see the connection with Tasty. It takes, as arguments, the descriptive name of the test, then a *TracePredicate* which we will get to in a moment, and an *EmulatorTrace* like the one we have used to test our contracts previously. And the result is a *TestTree* which, as we have seen, is the type of tests that Tasty uses. So, using this *checkPredicate* function we can produce something that the Tasty framework can understand.

There's also a variant with one additional argument of *CheckOptions*

```
checkPredicateOptions :: CheckOptions -> String -> TracePredicate -> EmulatorTrace () ->
↳ TestTree
```

*CheckOptions* has no constructors. This is a bit unfortunate, as we are forced to interact with it via three operations that take a type *Lens*'. *Lens*' is related to something called *optics* in Haskell. Optics is a huge topic in itself, with whole books haven been written about it, so we will just touch on it for now and just learn how to use the emulator trace.

One of its operations is *emulatorConfig* which allows us to specify initial distributions of funds, in a way similar to that which we have done in previous testing examples.

```
emulatorConfig :: Lens' CheckOptions EmulatorConfig
```

Now let's look at *TracePredicate*. This specifies some condition that the emulator trace should satisfy. This is what will be tested when we run the test.

First of all we see some logical combinators - a logical *not* and a logical *and*.

```
not :: TracePredicate -> TracePredicate
```

```
(.&&.) :: TracePredicate -> TracePredicate -> TracePredicate
```

There are lots of functions for producing *TracePredicates*. A few example are

```
endpointAvailable :: forall (l :: Symbol) w s e a. ( HasType l Endpoints.ActiveEndpoint,
↳ (Output s), KnownSymbol l, ContractConstraints s, Monoid w )
=> Contract w s e a -> ContractInstanceTag -> TracePredicate
```

```
queryingUtxoAt :: forall w s e a. ( UtxoAt.HasUtxoAt s, ContractConstraints s, Monoid w )
=> Contract w s e a -> ContractInstanceTag -> Address -> TracePredicate
```

```
assertDone :: forall w s e a. ( ContractConstraints s, Monoid w )
=> Contract w s e a -> ContractInstanceTag -> (a -> Bool) -> String -> TracePredicate
```

For our example, we will only use one of the available checks, *walletFundsChange*, which checks funds.

```
-- | Check that the funds in the wallet have changed by the given amount, excluding fees.
walletFundsChange :: Wallet -> Value -> TracePredicate
```

The *walletFundsChange* creates a *TracePredicate* that checks whether the funds in a *Wallet* have changed by a given *Value*. Interestingly, here, fees are ignored. We would have a hard time writing precise tests if this were not the case - we would find ourselves needing to approximate the costs of fees without knowing exactly what they would be.

There is a variation *walletFundsExactChange*, which *does* take fees into account.

If we go back to our test module *Spec.Trace* there is a function that we have not looked at yet, *tests*, and it uses this *checkPredicateOptions*.

```
tests :: TestTree
tests = checkPredicateOptions
  (defaultCheckOptions & emulatorConfig .~ emCfg)
  "token sale trace"
  (
    walletFundsChange (Wallet 1) (Ada.lovelaceValueOf 10_000_000 <>
    ↪ assetClassValue token (-60))
    .&&. walletFundsChange (Wallet 2) (Ada.lovelaceValueOf (-20_000_000) <>
    ↪ assetClassValue token 20)
    .&&. walletFundsChange (Wallet 3) (Ada.lovelaceValueOf (- 5_000_000) <>
    ↪ assetClassValue token 5)
  )
  myTrace
```

The first argument, as we have seen is of type *CheckOptions*. This is where we have to use optics, but we won't go into the details of that here. It is sufficient for now to note that we use the same *EmulatorConfig* as we used for *runMyTrace*.

The second argument is the descriptive name of the trace.

For the third argument, we use the *(.&&.)* combinator to chain together three different trace predicates, each of which uses the *walletFundsChange* function we saw above. Here we specify the changes that we expect to see in each of the wallets at the end of the trace - for example, we expect Wallet 1 to have gained 10 Ada and lost 60 Tokens.

We can now run this in the REPL.

```
Prelude Spec.Trace> import Test.Tasty
Prelude Test.Tasty Spec.Trace> defaultMain tests
token sale trace: OK (1.22s)

All 1 tests passed (1.22s)
*** Exception: ExitSuccess
```

This passes. Let's see what happens if it doesn't pass. We can change one of the values.

```
( walletFundsChange (Wallet 1) (Ada.lovelaceValueOf 10_000_000 <> assetClassValue
↪ token (-50) )
```

```
Prelude Test.Tasty Spec.Trace> :! Spec.Trace
[1 of 1] Compiling Spec.Trace      ( test/Spec/Trace.hs, /home/chris/git/ada/pioneer-
↪ fork/code/week08/dist-newstyle/build/x86_64-linux/ghc-8.10.4.20210212/plutus-pioneer-
↪ program-week08-0.1.0.0/t/plutus-pioneer-program-week08-tests/build/plutus-pioneer
↪ program-week08-tests/plutus-pioneer-program-week08-tests-tmp/Spec/Trace.o )
```

(continued from previous page)

```

Ok, one module loaded.
Prelude Test.Tasty.Spec.Trace> defaultMain tests
token sale trace: FAIL (1.32s)
  Expected funds of W1 to change by
    Value (Map [(,Map [("",100000000))],(aa,Map [("A",-50))]))
    (excluding 57430 lovelace in fees)
  but they changed by
    Value (Map [(,Map [("",100000000))],(aa,Map [("A",-60))]))
  Test failed.
  Emulator log:

  [INFO] Slot 0: TxnValidate
  ↳ 2125c8770581c6140c3c71276889f6353830744191de0184b6aa00b185004500
  [INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
    Contract instance started
  [INFO] Slot 1: 00000000-0000-4000-8000-000000000000 {Contract instance for wallet 1}:
    Receive endpoint call: Object (fromList [("tag",String "start"),(
  ↳ "value",Object (fromList [("unEndpointValue",Array [Object (fromList [(
  ↳ "unCurrencySymbol",String "aa"))]),Object (fromList [("unTokenName",String "A"))]))]))))
  [INFO] Slot 1: W1: Balancing an unbalanced transaction:
  ...
  ...
  [INFO] Slot 27: W1: TxSubmit:
  ↳ a42a06cc3e3b1653ec4aba5ab8304484d778adcbddac2ceb9f639f7e4bd1dfd2
  [INFO] Slot 27: TxnValidate
  ↳ a42a06cc3e3b1653ec4aba5ab8304484d778adcbddac2ceb9f639f7e4bd1dfd2
    src/Plutus/Contract/Test.hs:245:
    token sale trace

1 out of 1 tests failed (1.32s)
*** Exception: ExitFailure 1

```

We see a nice error message, followed by the emulator log, which we didn't get when the tests passed.

This is probably the simplest way to write automated tests for Plutus contracts. You simply write one or more emulator traces, and then use *checkPredicate* in association with the appropriate test predicates, to check that the trace leads to the desired result. This lets us write more or less traditional unit tests.

## 8.3 Optics and Lenses

Before we get to the second way of testing Plutus contracts, we will take a brief look at optics and lenses.

There are various competing optics libraries on Hackage, but the most prominent, and the most infamous one, and the one that the Plutus team decided to use is called *Lens*.

*Lens* is authored by Edward Kmett, who is probably the most prolific contributor to Haskell libraries.

You can see on the Hackage page there is a scary diagram. There is a whole zoo of optics. There are lenses and prisms and traversals and isos and whatnot. This diagram shows some of the operations that the library provides.

Optics are all about reaching deeply into hierarchical data types to inspect parts that are hidden deeply in the data type and to manipulate them.

Let's look at a very simple example in

## Lens: Lenses, Folds and Traversals

[ bsd2, data, generics, lenses, library ] [ Propose Tags ]

This package comes "Batteries Included" with many useful lenses for the types commonly used from the Haskell Platform, and with tools for automatically generating lenses and isomorphisms for user-supplied data types.

The combinators in `Control.Lens` provide a highly generic toolbox for composing families of getters, folds, isomorphisms, traversals, setters and lenses and their indexed variants.

An overview, with a large number of examples can be found in the [README](#).

An introductory video on the style of code used in this library by Simon Peyton Jones is available from [Skills Matter](#).

A video on how to use lenses and how they are constructed is available on [youtube](#).

Slides for that second talk can be obtained from [comonad.com](#).

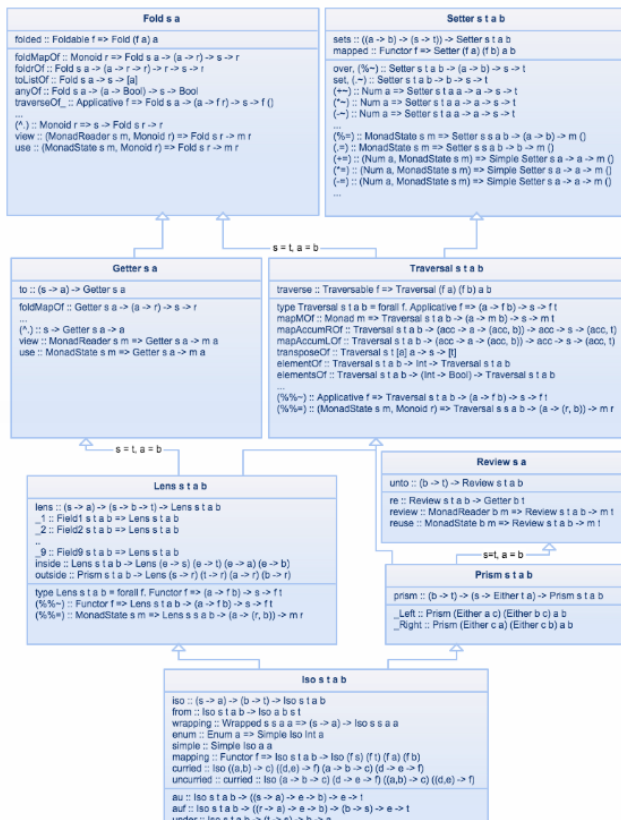
More information on the care and feeding of lenses, including a brief tutorial and motivation for their types can be found on the [lens wiki](#).

A small game of pong and other more complex examples that manage their state using lenses can be found in the [example folder](#).

### Lenses, Folds and Traversals

With some signatures simplified, the core of the hierarchy of lens-like constructions looks like:

With some signatures simplified, the core of the hierarchy of lens-like constructions looks like:



### Versions [RSS] [faq]

0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.0.1, 1.0.2, 1.0.3, 1.1, 1.1.1, 1.2, 1.3, 1.3.1, 1.4, 1.4.1, 1.5, 1.6, 1.7, 1.7.1, 1.8, 1.9, 1.9.1, 2.0, 2.1, 2.2, 2.3, 2.4, 2.4.0.2, 2.5, 2.6, 2.6.1, 2.7, 2.7.0.1, 2.8, 2.9, 3.0, 3.0.1, 3.0.2, 3.0.3, 3.0.4, 3.0.5, 3.0.6, 3.1, 3.2, 3.3, 3.4, 3.5, 3.5.1, 3.6, 3.6.0.1, 3.6.0.2, 3.6.0.3, 3.6.0.4, 3.7, 3.7.0.1, 3.7.0.2, 3.7.1, 3.7.1.1, 3.7.1.2, 3.7.2, 3.7.3, 3.7.4, 3.7.5, 3.7.6, 3.8, 3.8.0.1, 3.8.0.2, 3.8.1, 3.8.2, 3.8.3, 3.8.4, 3.8.5, 3.8.6, 3.8.7, 3.8.7.1, 3.8.7.2, 3.8.7.3, 3.9, 3.9.0.1, 3.9.0.2, 3.9.0.3, 3.9.1, 3.9.2, 3.10, 3.10.0.1, 3.10.1, 3.10.2, 3.10.3, 4.0, 4.0.1, 4.0.2, 4.0.3, 4.0.4, 4.0.5, 4.0.6, 4.0.7, 4.1, 4.1.1, 4.1.2, 4.1.2.1, 4.2, 4.3, 4.3.1, 4.3.2, 4.3.3, 4.4, 4.4.0.1, 4.4.0.2, 4.5, 4.6, 4.6.0.1, 4.7, 4.7.0.1, 4.8, 4.9, 4.9.1, 4.10, 4.11, 4.11.1, 4.12, 4.12.1, 4.12.2, 4.12.3, 4.13, 4.13.1, 4.13.2, 4.13.2.1, 4.14, 4.15, 4.15.1, 4.15.2, 4.15.3, 4.15.4, 4.16, 4.16.1, 4.17, 4.17.1, 4.18, 4.18.1, 4.19, 4.19.1, 4.19.2, 5, 5.0.1

### Change log

[CHANGELOG.markdown](#)

### Dependencies

array (>=0.5.0.0 && <0.6), assoc (>=1.0.2 && <1.1), base (>=4.7 && <5), base-orphans (>=0.5.2 && <1), bifunctors (>=5.1 && <6), bytestring (>=0.10.4.0 && <0.12), call-stack (>=0.1 && <0.5), comonad (>=5.0.7 && <6), containers (>=0.5.5.1 && <0.7), contravariant (>=1.3 && <2), distributive (>=0.3 && <1), exceptions (>=0.1.1 && <1), filepath (>=1.2.0.0 && <1.5),

containers (>=0.5.5.1 && <0.7), contravariant (>=1.3 && <2), distributive (>=0.3 && <1), exceptions (>=0.1.1 && <1), filepath (>=1.2.0.0 && <1.5), free (>=5.1.5 && <6), generic-deriving (>=1.10 && <2), ghc-prim, hashable (>=1.1.2.3 && <1.4), indexed-traversable (==0.1.\*), indexed-traversable-instances (==0.1.\*), kan-extensions (==5.\*), mtl (>=2.0.1 && <2.3), nats (>=0.1 && <1.2), parallel (>=3.1.0.1 && <3.3), profunctors (>=5.5.2 && <6), reflection (>=2.1 && <3), semigroupoids (==5.\*), semigroups (>=0.8.4 && <1), strict (==0.4.\*), tagged (>=0.4.4 && <1), template-haskell (>=2.9.0.0 && <2.18), text (>=1.2.3.0 && <1.3), th-abstraction (>=0.4.1 && <0.5), these (>=1.1.1.1 && <1.2), transformers (>=0.3.0.0 && <0.6), transformers-compat (>=0.4 && <1), unordered-containers (>=0.2.10 && <0.3), vector (>=0.9 && <0.13), void (>=0.5 && <1) [details]

### License

BSD-2-Clause

### Copyright

Copyright (C) 2012-2016 Edward A. Kmett

### Author

Edward A. Kmett

### Maintainer

Edward A. Kmett <ekmett@gmail.com>

### Revised

Revision 1 made by ryanglscott at 2021-05-16T11:55:20Z

### Category

Data, Lenses, Generics

### Home page

<http://github.com/ekmett/lens/>

### Bug tracker

<http://github.com/ekmett/lens/issues>



```
module Week08.Lens
```

We have a type *Company* which is a wrapper around a list of *Person*. There is a field *\_staff*. When dealing with lenses, it is convention to start field names with underscores.

```
newtype Company = Company {_staff :: [Person]} deriving Show

data Person = Person
  { _name    :: String
  , _address :: Address
  } deriving Show

newtype Address = Address {_city :: String} deriving Show
```

And we define two *Persons* and a *Company* with which these *Persons* are associated.

```
alejandro, lars :: Person
alejandro = Person
  { _name    = "Alejandro"
  , _address = Address {_city = "Zacateca"}
  }
lars = Person
  { _name    = "Lars"
  , _address = Address {_city = "Regensburg"}
  }

iohk :: Company
iohk = Company { _staff = [alejandro, lars] }
```

The task is to write a simple function, *goTo*, that gets a *String* as argument along with a *Company*. The function should create a new company which it gets by changing all the cities of all the staff of the company with the given string.

If we apply that to *iohk* with a string argument of “Athens”, then we should get a *Company* with the same two *Persons*, but now both of those *Persons* have a city of “Athens”.

You don’t need any advanced Haskell to achieve this, but it’s a bit messy, even in this simple example. The function below uses record syntax to modify specific fields of records, while leaving the other fields the same.

The helper function *movePerson* updates the *\_address* field of the *Person* *p*, and the *\_city* field of that *Address*, and the main part of the function maps the *movePerson* function over each member of *\_staff*.

```
goTo :: String -> Company -> Company
goTo there c = c {_staff = map movePerson (_staff c)}
  where
    movePerson p = p {_address = (_address p) {_city = there}}
```

We can look at the original company in the REPL.

```
Prelude Week08.Lens> iohk
Company {_staff = [Person {_name = "Alejandro", _address = Address {_city = "Zacateca"}},
↪ Person {_name = "Lars", _address = Address {_city = "Regensburg"}}]}
```

Now, let’s apply the *goTo* function to it, and see the changes.



```
Prelude Week08.Lens> goTo "Athens" iohk
Company {_staff = [Person {_name = "Alejandro", _address = Address {_city = "Athens"}},
↳ Person {_name = "Lars", _address = Address {_city = "Athens"}}]}
```

So, dealing with nested record types, even though it is quite simple conceptually, can be quite messy.

This is what optics try to make easier with the idea of providing first-class field accessors. In the end it's very similar to dealing with such data types in an imperative language such as C# or Java.

We saw in lecture four how monads can be viewed as a programmable semi-colon, where the semi-colon is the statement separator in many imperative languages. In a similar way, optics can be thought of as providing a programmable dot, where a dot is the accessor dot as in Python or Java.

You could implement lenses by hand, but the *lens* library provides some Template Haskell magic to do it automatically, so long as we follow the underscore convention mentioned above.

```
makeLenses ''Company
makeLenses ''Person
makeLenses ''Address
```

The names of the lenses will be the names of the original fields without the underscore.

There is a way, within the REPL, to inspect what code Template Haskell writes at compile time.

First, enable the following flag

```
Prelude Week08.Lens> :set -ddump-splices
```

Then, reload the module. If nothing happens, you'll need to make a minor change to the code, perhaps by adding some whitespace, before reloading.

```
Prelude Week08.Lens> :r
[4 of 4] Compiling Week08.Lens      ( src/Week08/Lens.hs, /home/chris/git/ada/pioneer-
↳ fork/code/week08/dist-newstyle/build/x86_64-linux/ghc-8.10.4.20210212/plutus-pioneer-
↳ program-week08-0.1.0.0/build/Week08/Lens.o )
src/Week08/Lens.hs:35:1-20: Splicing declarations
    makeLenses ''Company
=====>
    staff :: Iso' Company [Person]
    staff = (iso (\ (Company x_abB0) -> x_abB0)) Company
    {-# INLINE staff #-}
src/Week08/Lens.hs:36:1-19: Splicing declarations
    makeLenses ''Person
=====>
    address :: Lens' Person Address
    address f_abEJ (Person x1_abEK x2_abEL)
      = (fmap (\ y1_abEM -> (Person x1_abEK) y1_abEM)) (f_abEJ x2_abEL)
    {-# INLINE address #-}
    name :: Lens' Person String
    name f_abEN (Person x1_abEO x2_abEP)
      = (fmap (\ y1_abEQ -> (Person y1_abEQ) x2_abEP)) (f_abEN x1_abEO)
    {-# INLINE name #-}
src/Week08/Lens.hs:37:1-20: Splicing declarations
    makeLenses ''Address
=====>
    city :: Iso' Address String
```

(continues on next page)

(continued from previous page)

```
city = (iso (\ (Address x_abFw) -> x_abFw)) Address
{-# INLINE city #-}
```

This now shows us what Template Haskell does.

We see that *makeLenses* for *Company* creates a function *staff*, which returns an *Iso*' - a type of optic - from *Company* to *[Person]*.

For *makeLenses Person* we get an *address* function which returns a *Lens*' from *Person* to *Address*, and we also get a *name* lens from *Person* to *String*.

For *makeLenses Address* we get a *city* function which returns an *Iso*' from *Address* to *String*.

*Iso* and *Lens* are two different types of optics but the order of type arguments is always the same. You always have two type arguments, at least for these primed versions (there are more general optics which take four type parameters). The first argument is always the big data type and the second parameter is the part you are zooming into. The name optics relates to the mental image of zooming into a datatype.

Let's try them out in the REPL.

```
Prelude Week08.Lens> lars
Person {_name = "Lars", _address = Address {_city = "Regensburg"}}

Prelude Week08.Lens> import Control.Lens

Prelude Control.Lens Week08.Lens> lars ^. name
"Lars"

Prelude Control.Lens Week08.Lens> lars ^. address
Address {_city = "Regensburg"}
```

A very powerful feature of lenses is that you can compose them.

Where we have, above, something going from *Person* to *Address* and we have something else going from *Address* to *String*, then we can combine them using the function composition dot. There is some advanced type-level machinery going on behind the scenes to make that work, but it works.

```
Prelude Control.Lens Week08.Lens> lars ^. address . city
"Regensburg"
```

Not only can you view the contents of record types like this, but you can also manipulate them.

```
Prelude Control.Lens Week08.Lens> lars & name .~ "LARS"
Person {_name = "LARS", _address = Address {_city = "Regensburg"}}
```

The *&* symbol here is function application, but the other way around - the argument comes first and then the function.

Again, we can compose.

```
Prelude Control.Lens Week08.Lens> lars & address . city .~ "Munich"
Person {_name = "Lars", _address = Address {_city = "Munich"}}
```

There is another type of optics called *Traversable*s, that zooms not only into one field, but into many simultaneously. If you had a list it would zoom into each element. So, for example, we could use a list of integers, with the *each* traversable that works with many container types, including lists, and set every element to 42.

```
Prelude Control.Lens Week08.Lens> [1 :: Int, 3, 4] & each .~ 42
[42,42,42]
```

You may see a *type-defaults* warning when you run the above, but it is removed here.

A cool thing is that various types of lenses can be combined, again with the dot operator. For example

```
Prelude Control.Lens Week08.Lens> iohk & staff . each . address . city .~ "Athens"
Company {_staff = [Person {_name = "Alejandro", _address = Address {_city = "Athens"}},
  Person {_name = "Lars", _address = Address {_city = "Athens"}}]}
```

And this is exactly what our *goTo* function achieved, so we can write *goTo* as

```
goTo' :: String -> Company -> Company
goTo' there c = c & staff . each . address . city .~ there
```

And this is actually what we did when we configured our test.

```
tests :: TestTree
tests = checkPredicateOptions
  (defaultCheckOptions & emulatorConfig .~ emCfg)
```

The function *defaultCheckOptions* is of type *CheckOptions* and there is a lens from *CheckOptions* to *EmulatorConfig*, and this is the part that we wanted to change.

And that concludes our brief excursion into optics and lenses.

## 8.4 Property Based Testing

Property Based Testing is quite a revolutionary approach to testing that is much more powerful than simple unit testing. It originated from Haskell, which, with its pureness and immutable data structures is particularly suited to this approach. It has now been copied by almost all other programming languages.

### 8.4.1 QuickCheck

One of the inventors of *QuickCheck*, which is the most prominent and was the first library using this approach, is John Hughes, who is also one of the original inventors of Haskell. He and his company work with IOHK to provide special support of this approach to testing Plutus contracts.

Before we look at using QuickCheck for Plutus contracts, let's first look at its use for pure Haskell programs.

Property based testing subsumes unit tests. Let's write a very simple and silly unit test.

```
prop_simple :: Bool
prop_simple = 2 + 2 == (4 :: Int)
```

This function is available in the module.

```
module Week08.QuickCheck
```

After loading this module, and the *Test.QuickCheck* module, we can test our unit test in the REPL.

```
Prelude Control.Lens Test.QuickCheck Week08.QuickCheck> quickCheck prop_simple
+++ OK, passed 1 test.
```

This is not very exciting. For a more interesting example, the same module contains a buggy implementation of an insertion sort.

```
sort :: [Int] -> [Int] -- not correct
sort []      = []
sort (x:xs) = insert x xs

insert :: Int -> [Int] -> [Int] -- not correct
insert x []      = [x]
insert x (y:ys)  | x <= y      = x : ys
                  | otherwise  = y : insert x ys
```

To test it, a property that would could test would be that after applying sort to a list of integers, the resulting list is sorted.

```
isSorted :: [Int] -> Bool
isSorted []      = True
isSorted [_]     = True
isSorted (x : y : ys) = x <= y && isSorted (y : ys)
```

Using this, we can now provide a QuickCheck property that is not just simply of type *Bool*, but instead is a function from a list of *Ints* to *Bool*.

```
prop_sort_sorts :: [Int] -> Bool
prop_sort_sorts xs = isSorted $ sort xs
```

You can read that like a specification, which says “for all the lists of integers *xs*, if you apply *sort* to it, then the result should be sorted.”

QuickCheck can deal with such properties.

In the REPL

```
Prelude Control.Lens.Test.QuickCheck Week08.QuickCheck> quickCheck prop_sort_sorts
*** Failed! Falsified (after 8 tests and 4 shrinks):
[0,0,-1]
```

It fails, and gives us an example where the property does not hold. We can test that example.

```
Prelude Control.Lens.Test.QuickCheck Week08.QuickCheck> sort [0, 0, -1]
[0,-1]
```

And can see that, indeed, it is not correct.

How does QuickCheck do this? If you provide a function with one or more arguments, it will generate random arguments for the function. In our example, QuickCheck has generated 100 random lists of integers and, for each of those lists, has checked whether the property holds, until it hit a failure.

Note that the failure was reported as

```
*** Failed! Falsified (after 8 tests and 4 shrinks):
```

This means that after 8 tests the property was falsified, but at this point, rather than just report the failure, it has tried to shrink it - to simplify it.

This is a powerful feature of QuickCheck, because the random counter examples that QuickCheck finds are very complicated - long lists with long numbers. But once a counter example has been found, QuickCheck tries to simplify it,

perhaps by dropping some elements from the list, or by making some of the numbers smaller, until it doesn't find a way to get an even simpler example.

It is this combination of random test generation and shrinking that makes QuickCheck so tremendously useful.

We can see what type of random lists QuickCheck generates.

```
Prelude Control.Lens Test.QuickCheck Week08.QuickCheck> sample (arbitrary :: Gen [Int])
[]
[0]
[2,4,-1,3]
[3,-1,4,3,-5]
[3,-1,-8,-4,-6]
[4,5,-1,4,-7,2,8,4,-5]
[-8,-8,-11,-12,2,-4,-12,2,4]
[7,9,3,-5,5,-9,3,1,11]
[12,-7,-9,9,-11,-15,5,-10,-7,4,8,8,-12,-6,16]
[-11,11,-1,-6]
[14,2,-5,9,13,-8,-8,-17,-1,-11,-19,15,9,8,-19,-4,16,4,4,19]
```

The way QuickCheck does this random generation is by using a type class called *Arbitrary*

```
Prelude Control.Lens Test.QuickCheck Week08.QuickCheck> :i Arbitrary
type Arbitrary :: * -> Constraint
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
```

There are many more lines to the above output, but the important ones are shown. We can see that it has two methods. One is called *arbitrary* and one is called *shrink*.

*Gen* is yet another monad. The monad provides various methods that allow for random number generation for values of type *a*.

The second method is *shrink*, which, when given an *a* will provide a list of simpler versions of *a*. This, of course, depends on the type of *a*.

If we look at the output above that provides some random integer lists, we see something interesting. The further we go down the list, the more complicated the list becomes. The first is just the empty list, then we get single-element lists, then some longer lists, and it tends towards greater complexity over time.

In addition to just providing random generation in the *Gen* monad, there is also a concept of complexity. If you implement an instance of *Gen* you are expected not only to generate a random *a* but also a random *a* of some given complexity.

When *QuickCheck* checks a property, it starts with simple, random arguments, then makes them more complex over time. By default it tests 100 random arguments, but this can be configured.

Now that we know that our code fails, let's try to fix it.

```
sort :: [Int] -> [Int] -- not correct
sort []      = []
sort (x:xs) = insert x xs
```

The problem is that all we do for a non-empty list is to insert the first element into the tail, but we don't recursively sort the tail.

Our first attempt to fix...

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = insert x $ sort xs
```

Now, when we test this...

```
Prelude Control.Lens.Test.QuickCheck> :r
[1 of 1] Compiling Week08.QuickCheck ( src/Week08/QuickCheck.hs, /home/chris/git/ada/
↳pioneer-fork/code/week08/dist-newstyle/build/x86_64-linux/ghc-8.10.4.20210212/plutus-
↳pioneer-program-week08-0.1.0.0/build/Week08/QuickCheck.o )
Ok, one module loaded.
Prelude Control.Lens.Test.QuickCheck Week08.QuickCheck> quickCheck prop_sort_sorts
+++ OK, passed 100 tests.
```

It passes. However, if we test specifically for the case that failed previously...

```
Prelude Control.Lens.Test.QuickCheck Week08.QuickCheck> sort [0, 0, -1]
[-1,0]
```

It is clearly not correct. Even though the list has been sorted, the length of the list has changed. This leads to an important point. QuickCheck can't do magic - its results are only as good as the properties we provide. What we see here is that our property *prop\_sort\_sorts* is not strong enough to test if the function is correct.

We can add a second property that checks the length.

```
prop_sort_preserves_length :: [Int] -> Bool
prop_sort_preserves_length xs = length (sort xs) == length xs
```

And we find that this property is not satisfied by our code.

```
Prelude Control.Lens.Test.QuickCheck Week08.QuickCheck> quickCheck prop_sort_preserves_
↳length
*** Failed! Falsified (after 4 tests and 3 shrinks):
[0,0]
```

The bug in our code is in the *insert* function.

```
insert :: Int -> [Int] -> [Int] -- not correct
insert x [] = [x]
insert x (y:ys) | x <= y = x : ys
                 | otherwise = y : insert x ys
```

We say here that, if *x* is less or equal to *y*, then we append *x* to *ys*, but we have forgotten about the *y*. It should read:

```
insert x (y:ys) | x <= y = x : y : ys
```

This should fix it.

```
Prelude Control.Lens.Test.QuickCheck Week08.QuickCheck> :r
Prelude Control.Lens.Test.QuickCheck Week08.QuickCheck> quickCheck prop_sort_preserves_
↳length
+++ OK, passed 100 tests.
```

Of course, this is still not proof that our function is correct, because these two properties are still not enough to specify a sorting function fully. For example, the sorting function could return a list of the same length containing only zeroes. This would pass all tests. It is quite an art to find properties to guarantee that, if they are all satisfied, there is no bug.

Even so, this approach to testing is often more effective than unit testing as it can test a huge number of random cases and can find examples of failure which a programmer writing a unit test may not have thought of.

### Using QuickCheck with Plutus

Now that we have seen what QuickCheck can do, we will turn our attention to using it to test Plutus contracts.

Here we hit a problem - how do you use QuickCheck to test side-effected code? This problem does not only arise with blockchain, it arises with all systems that use IO.

John Hughes always uses the example of the file system. How would you test file system operations, i.e. reading, writing, opening and closing files, using QuickCheck.

The approach to use is very similar to the one you can use with Plutus. The idea is that you start with a model.



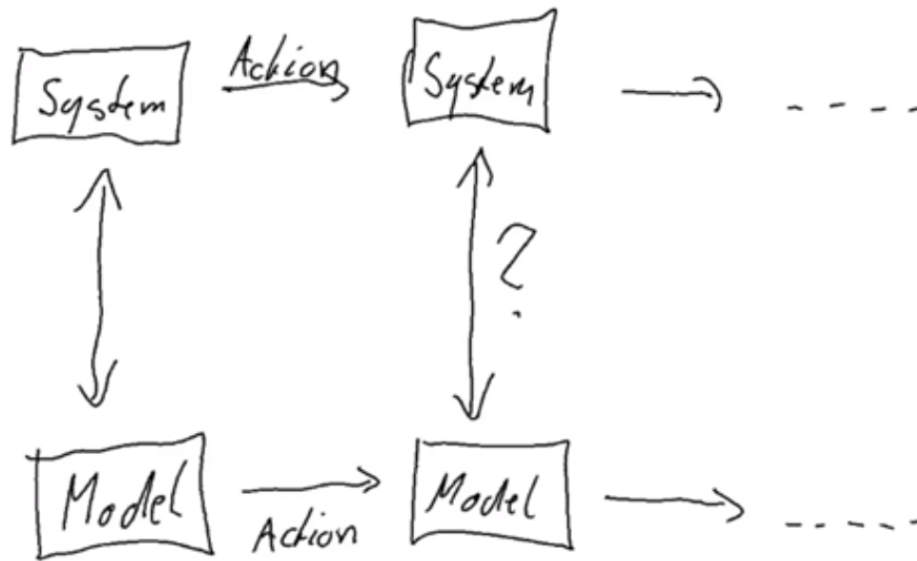
The model is basically an idealised model of how the real world system should work. There must be some sort of relation between the model and the real system.

If the real system is a file system, then you could, in the model, have an idealised version of how you think files should work. And then, what QuickCheck does, in its random generation, is to generate a random sequence of actions that you can perform on the system. In the example of a file system, it would randomly generate a sequence of opening files, closing files, writing to files, reading files and so on. Now you can basically step this model and the system in parallel.

You have some sort of action that you perform in the real world, and you apply the same type of action to your model. Then your real system has progressed into a new state, and your model has also been updated. After this step, you can compare the two and check that they are still in sync. You can then continue this for several steps.

While our first QuickCheck example generated a random list of *Ints*, the idea for testing a real world system is to generate random lists of actions and then to apply those actions both to a model and to the real system and to check that the model and the real system stay in sync.

Shrinking in this example would be that, if you have a list of actions that show that there is a bug, then you can, for example, drop some of the actions and see whether the problem still arises. This can be repeated until you cannot drop any further actions from the list and still reproduce the bug.



This is exactly how the QuickCheck support for Plutus works. In order to test a Plutus contract, we have to come up with a model and define our expectations of how the various endpoints, for example, would change the model. We would then need to provide a link between the model and the real system (the emulator), and then apply the QuickCheck machinery.

The code to do this is in

```
module Spec.Model
```

We notice that we import two Plutus test modules, with the QuickCheck support being provided by the *ContractModel*, which has all the machinery to define a model and to link it to a real contract.

```
import Plutus.Contract.Test
import Plutus.Contract.Test.ContractModel
```

And we import three more test modules. One for Tasty, one for QuickCheck, and one that allows for using QuickCheck properties in Tasty test suites.

```
import Test.QuickCheck
import Test.Tasty
import Test.Tasty.QuickCheck
```

To define a model, we first define a datatype that represents the state of one *TokenSale* instance.

```
data TSSState = TSSState
  { _tssPrice    :: !Integer
  , _tssLovelace :: !Integer
  , _tssToken    :: !Integer
  } deriving Show
```

It has three fields - the current price, the current supply of lovelace in the contract, and the current supply of tokens in the contract.



Then our model *TModel* is a map from wallets to *TokenSale* states.

```
newtype TModel = TModel {_tsModel :: Map Wallet TSState}
deriving Show
```

The idea in this test is that we have two wallets and each wallet runs its own *TokenSale* contract, and the two wallets will trade different tokens.

We create lenses for the model. We need optics to interact with the *ContactModel* library.

```
makeLenses ''TModel
```

All the logic that defines how our model should behave, and how it is linked to the real contract is in

```
instance ContractModel TModel where
```

First we have an associated datatype. This is quite an advanced Haskell feature. In type classes, as well as methods, you can have data types. We have seen this before in validators where we define a dummy type that provides a link between the datum type and the redeemer type.

Here, we associate an *Action* type, which represents the actions that QuickCheck will generate. In principal, we just have one constructor in the *Action* type for each of the endpoints we saw earlier. We have additional arguments because now there are additional wallets at play and we must keep track of which wallet performs an action.

```
data Action TModel =
  Start Wallet
  | SetPrice Wallet Wallet Integer
  | AddTokens Wallet Wallet Integer
  | Withdraw Wallet Wallet Integer Integer
  | BuyTokens Wallet Wallet Integer
deriving (Show, Eq)
```

*Start Wallet* means that this wallet starts the token sale contract.

*SetPrice Wallet Wallet Integer* means that the second wallet sets the price for the token sale contract operated by the first wallet. We know from the contract logic that this should only work if both the wallets are the same, because only the owner of the contract can set the price.

*AddTokens* is very similar to *SetPrice*.

For *Withdraw*, the second wallet attempts to withdraw a certain number of lovelace and a certain number tokens (respectively) from the token sale run by the first wallet. Again, this should fail if the two wallets are not the same.

In *BuyTokens*, the second wallet will try to buy a certain number of tokens from the token sale run by the first wallet.

So, the *Action* type is the first ingredient.

The second ingredient is another associated datatype. For each instance of a contract that we are running, we want a key that identifies the instance.

```
data ContractInstanceKey TModel w s e where
  StartKey :: Wallet          -> ContractInstanceKey TModel (Last TokenSale) ⊥
  ⊔TSStartSchema' Text
  UseKey    :: Wallet -> Wallet -> ContractInstanceKey TModel ()      ⊥
  ⊔TSUseSchema Text
```

This is a generalised, algebraic datatype (GADT), so it's a little different to usual data declarations in Haskell. Instead of just providing the constructors, you provide the constructors with a type signature.

In *ContractInstanceKey*, we have a constructor *StartKey* that takes a *Wallet* as an argument and then produces something of type

**ContractInstanceKey TSMModel (Last TokenSale) TSSStartSchema' Text**

The point of GADTs is that with normal datatypes, the type parameters are the same for all constructors, for example, *Action TSMModel* has five constructors, but the type is always *TSMModel*. But with GADTs, we are able to provide a more generalised type parameter - in this case *TSMModel w s e*.

We need this feature in this case because our contracts can have different type parameters.

There are two types of instances. Recall we have the *start* contract and the *use* contract, which have different type signatures.

*StartKey* returns a type that consists of our model and then the parameters that come from the contract itself - the state type, the schema, and the error type. We used the primed version of *TSSStartSchema* - *TSSStartSchema'* because we don't want to create the NFT, we want to pass it in ourselves because it makes it easier to write the test if we know what NFT we will be using in advance.

We also provide a key for the use contract which takes two *Wallets* as parameters. The first is the one that owns the token sale that we are interacting with and the second is the one that actually runs the contract. As for the type parameters, there is no state parameter, and it uses a different schema - *TSUseSchema*, but the error type is the same.

Next we need to provide the *instanceTag* method which, given an instance key and a wallet, will provide a so-called contract instance tag. As we already know the wallet that runs the instance, because that was one of the arguments to the instance key constructor we can ignore it as an argument.

```
instanceTag key _ = fromString $ "instance tag for: " ++ show key
```

The *instanceTag* function doesn't have an accessible constructor, but it implements the *IsString* class. We haven't seen the *IsString* class explicitly but we have used it when we used the *OverloadedStrings* GHC extension - it allows a type class that implements it to be represented by a string literal. In particular, it has a method *fromString* which, given a string, will create an instance of the type.

The "instance tag for: " literal in the function above isn't necessary - all that is necessary is for the whole string to be unique for each instance that we will ever run in our tests.

There is a default implementation for the *instanceTag* method of the *ContractModel* class, so you normally don't have to implement it yourself. However, it only works if you have at most one contract instance per wallet. This is not the case for us, as we will have three instances per wallet - one *start* instance and two *use* instances (one for the own wallet's token sale, and one for the other wallet's token sale).

The next method that we need to implement is *arbitraryAction* which is how we tell the system how to generate a random action.

```
arbitraryAction _ = oneof $
  (Start <$> genWallet) :
  [ SetPrice <$> genWallet <*> genWallet <*> genNonNeg ] ++
  [ AddTokens <$> genWallet <*> genWallet <*> genNonNeg ] ++
  [ BuyTokens <$> genWallet <*> genWallet <*> genNonNeg ] ++
  [ Withdraw <$> genWallet <*> genWallet <*> genNonNeg <*> genNonNeg ]
```

As an argument it gets the model state. We will come to this later, but we don't need it here and so ignore it in the method declaration.

The function *oneof* is one of the combinators provided by QuickCheck. Given a list of arbitrary actions, it randomly picks one of those.

Here we are using something else that we have not seen before - the applicative style. Recall that when we looked at monads, we saw that *Monad* has *Applicative* as a superclass. *Applicative* is often useful to write more compact monadic code.

First let's look at the *genWallet* function.

```
genWallet :: Gen Wallet
genWallet = elements wallets
```

In the random generation monad *Gen*, it generates a random wallet. It uses another combinator provided by QuickCheck, *elements*, which simply takes a list of the type that we wish to generate, and randomly picks one of those elements.

This is using another helper function *wallets*.

```
wallets :: [Wallet]
wallets = [w1, w2]
```

Which, in turn, uses

```
w1, w2 :: Wallet
w1 = Wallet 1
w2 = Wallet 2
```

So *genWallet* will randomly pick either *Wallet 1* or *Wallet 2*.

Getting back to the *arbitraryAction* code.

```
Start <$> genWallet
```

What this means is that we first use *genWallet* to generate a random wallet and then return *Action Start w*, where *w* is the wallet we have just picked.

The right-hand side is of type *Gen Wallet* and *Start* takes a *Wallet* and returns an action. If we *fmap* (<\$>) this, we get a type of *Gen Wallet -> Gen Action*, which is what we want.

For the other four actions, we use an additional helper function *genNonNeg* which generates a nonnegative number.

```
genNonNeg :: Gen Integer
genNonNeg = getNonNegative <$> arbitrary
```

Now, when we want to generate a random action for *SetPrice*, this is where the applicative style really shines.

```
SetPrice <$> genWallet <*> genWallet <*> genNonNeg
```

If we wanted to write this in a *do* block, we would do something like

```
w1 <- genWallet
w2 <- genWallet
p <- genNonNeg
return (SetPrice w1 w2 p)
```

You can use the applicative style if the actions in the monad you are invoking don't depend on the result of previous actions. In a *do* block, you could do inspect the the result in *w1* and make some choice based upon it. This is not possible in *Applicative*, but often monadic code doesn't make use of this power, and in these situations, we have this more compact way of writing it.

We can try out the *arbitraryAction* function in the REPL.

```
Prelude Test.QuickCheck Plutus.Contract.Test.ContractModel Spec.Model> sample_
↳ (arbitraryAction undefined :: Gen (Action TSMModel))
Start (Wallet 1)
AddTokens (Wallet 1) (Wallet 1) 1
AddTokens (Wallet 1) (Wallet 1) 3
SetPrice (Wallet 1) (Wallet 2) 3
SetPrice (Wallet 1) (Wallet 1) 2
AddTokens (Wallet 1) (Wallet 1) 1
SetPrice (Wallet 2) (Wallet 1) 12
Withdraw (Wallet 2) (Wallet 1) 14 3
AddTokens (Wallet 2) (Wallet 1) 9
AddTokens (Wallet 2) (Wallet 1) 18
SetPrice (Wallet 2) (Wallet 1) 17
```

We see that it generates a sample of random actions with random arguments.

The next method to implement is *initialState* which, as the name suggests, is the initial state of our model.

```
initialState = TSMModel Map.empty
```

Now comes the most complex function that we must implement to set this up. You will recall from when we looked at the diagram that we must know what effect performing an action will have on the model. This is exactly what the *nextState* function does.

If we look at the type of *nextState*, we see that it takes an action and returns something in yet another monad, this time the *Spec* monad. The *Spec* monad allows us to inspect the current state of our model, and also to transfer funds within our model.

```
nextState :: ContractModel state => Action state -> Spec state ()
```

Let's look at an example for *Start*. This should tell us the effect on our model if wallet *w* starts a token sale.

```
nextState (Start w) = do
  withdraw w $ nfts Map.! w
  (tsModel . at w) $= Just (TSSState 0 0 0)
  wait 1
```

Here we see a function from the *Spec* monad called *withdraw*. Using *withdraw* means that some funds go from a wallet to a contract - it doesn't matter which contract. So, this says that the effect of *Start* will be that Wallet *w* loses the NFT.

The NFT is again something that is defined in a helper function. Let's quickly look at the helper functions that define the NFTs and tradable tokens.

Each wallet will trade its own token and each wallet will have its own NFT.

```
tokenCurrencies, nftCurrencies :: Map Wallet CurrencySymbol
tokenCurrencies = Map.fromList $ zip wallets ["aa", "bb"]
nftCurrencies   = Map.fromList $ zip wallets ["01", "02"]

tokenNames :: Map Wallet TokenName
tokenNames = Map.fromList $ zip wallets ["A", "B"]

tokens :: Map Wallet AssetClass
tokens = Map.fromList [(w, AssetClass (tokenCurrencies Map.! w, tokenNames Map.! w)) | w
  <- wallets]
```

(continues on next page)

(continued from previous page)

```
nftAssets :: Map Wallet AssetClass
nftAssets = Map.fromList [(w, AssetClass (nftCurrencies Map.! w, nftName)) | w <- wallets]
```

```
nfts :: Map Wallet Value
nfts = Map.fromList [(w, assetClassValue (nftAssets Map.! w) 1) | w <- wallets]
```

Wallet 1 will trade the A token and Wallet 2 will trade the B token. Wallet one will have the 01 NFT and Wallet two will have the 02 NFT.

While we are here, we can look at the *tss* helper which exists alongside the above helper functions and maps the wallets to their *TokenSale* parameters.

```
tss :: Map Wallet TokenSale
tss = Map.fromList
  [ (w, TokenSale { tsSeller = pubKeyHash $ walletPubKey w
                    , tsToken  = tokens Map.! w
                    , tsNFT    = nftAssets Map.! w
                    })
    | w <- wallets
  ]
```

Now, back to the *nextState* function. The first line of the *do* block says that the effect of calling *Start* will be that the wallet will lose the NFT to the contract. Remember that the NFT is locked in the contract when we start the token sale.

```
nextState (Start w) = do
  withdraw w $ nfts Map.! w
  (tsModel . at w) $= Just (TSSState 0 0 0)
  wait 1
```

Secondly, there will be an effect on the model state. Remember that the model state is a map from *Wallet* to *TSSState*, where *TSSState* is a triple of price, tokens and Ada.

The second line of the *do* block says that after the contract has started, there will be an entry in the map at key *w* with 0 price, 0 tokens and 0 Ada.

The left hand side of the expression is another example of an optic, this time allowing us to access the map *\_tsModel* from *TSMModel*. The *at* lens allows us to reference a map entry at a given key. The type returned by this optic is a *Maybe* as the key may or may not be there.

The *\$=* comes from the Spec monad and it takes a lens on the left-hand side and then a new value on the right-hand side.

The *wait* function comes from the Spec monad and says here that the *Start* will take one slot.

Now we do something similar for all the other operations. Firstly, *SetPrice*.

```
nextState (SetPrice v w p) = do
  when (v == w) $
    (tsModel . ix v . tssPrice) $= p
  wait 1
```

In this function, we only do something if the wallet that invokes *SetPrice* is the same as the wallet that is running the token sale. If it is then the funds don't move, but we must update the model.

We use a different optic - instead of *at* we use *ix* which is a *Traversal*. It is similar to *at*, but whereas *at* returned a *Maybe*, *ix* does not. It also uses the *tssPrice* lens to access the first element of the *TSSState* triple, which it sets to the price. In the event that *ix* does not find an entry, the line will have no effect.

Whether or not the wallets match, and whether or not the price update succeeds, we wait one slot.

The model state change for *AddTokens* is more complex.

```
nextState (AddTokens v w n) = do
  started <- hasStarted v
  -- has the token sale started?
  ...
```

First we check the the token sale for wallet *v* has actually started, and this is yet another helper function.

```
getTSSState' :: ModelState TSMModel -> Wallet -> Maybe TSSState
getTSSState' s v = s ^. contractState . tsModel . at v
```

Given a *ModelState* (which is of type *TSMModel* but with additional information such as current funds and current time) and given a *Wallet*, we want to extract the *TSSState* which is the state of the token sale contract for that wallet, which may or may not have started yet.

This is again performed using optics. There is a lens called *contractState* which, here is the *TSMModel* type. We then zoom into the map and use the *at* lens, which will return *Nothing* if the wallet key *v* does not exist, or a *Just TSSState* if it is there.

Using this, we can write a slight variety of this function which doesn't have the first argument. Instead it takes just the *Wallet* argument, but then returns the *Maybe TSSState* in the *Spec* monad. In order to do that, we use a feature of the *Spec* monad, a function called *getModelState*, which will return the model state, which we then pass to the primed version of the function along with the *Wallet* argument.

```
getTSSState :: Wallet -> Spec TSMModel (Maybe TSSState)
getTSSState v = do
  s <- getModelState
  return $ getTSSState' s v
```

And then another variation, this time called *hasStarted*, which will tell us, within the *Spec* monad, whether the token sale has started or not.

```
hasStarted :: Wallet -> Spec TSMModel Bool
hasStarted v = isJust <$> getTSSState v
```

This just checks whether the return value from *getTSSState v* is a *Just* or a *Nothing*. The *isJust* function returns *True* if it is a *Just*, and we need to use *fmap* to lift it into the *Spec* monad.

Continuing the *nextState* function for *AddTokens*

```
nextState (AddTokens v w n) = do
  started <- hasStarted v
  when (n > 0 && started) $ do
    bc <- askModelState $ view $ balanceChange w
```

If the token sale has not started, we don't do anything because *AddTokens* shouldn't have any effect in that case.

We also check that the number of tokens to be added is greater than zero. If not, again we do nothing. Otherwise, we continue.

We now see another function from the *Spec* monad called *askModelState*, which is similar to *getModelState* but it doesn't return the complete model state but instead takes a function and applies it to the the model state. The function

`view` comes from the `lens` library and is just another name for the `^.` operator for viewing the result of zooming into a lens.

And there is a `balanceChange w` lens which is a lens to the balance change of wallet `w`. The balance change refers to how much the funds of the wallet have changed since the start of the simulation.

At this point we have the balance change bound to `bc`. The reason we are doing this is because we want to make sure that the wallet has enough funds to add the requested number of tokens, which we now do. First we look up the token.

```
let token = tokens Map.! v
```

Then we check whether the wallet has enough of them.

```
when (tokenAmt + assetClassValueOf bc token >= n) $ do -- does the wallet have the
  tokens to give?
  withdraw w $ assetClassValue token n
  (tsModel . ix v . tssToken) $~ (+ n)
wait 1
```

The number in `tokenAmt` is the number of tokens the wallet had at the start, so by adding this to the balance change for the token, we get the number of tokens currently in the wallet.

If we have enough tokens, then we withdraw the correct number of tokens from the wallet, and we update the model to show that the tokens should now be in the contract. Note that instead of using `$=` to set the value, we use the `$~` function which applies a function to a value.

Again, we wait one slot.

Next, we write a `nextState` function for `BuyTokens`.

```
nextState (BuyTokens v w n) = do
when (n > 0) $ do
  m <- getTSState v
  case m of
    Just t
      | t ^. tssToken >= n -> do
        let p = t ^. tssPrice
            l = p * n
        withdraw w $ lovelaceValueOf l
        deposit w $ assetClassValue (tokens Map.! v) n
        (tsModel . ix v . tssLovelace) $~ (+ l)
        (tsModel . ix v . tssToken) $~ (+ (- n))
      _ -> return ()
wait 1
```

First we check the the number of tokens we are attempting to buy is positive. If so, then we get the state of the token sale.

If the state is a `Just` then we know that the token sale has started.

```
m <- getTSState v
case m of
  Just t
```

If so, then we use optics to check that the number of tokens available in the contract is at least enough for us to buy what we are asking for.

```
t ^ . tssToken >= n -> do
```

If we are still going, we then lookup the current price and calculate how much the requested number of tokens will cost.

```
let p = t ^ . tssPrice
l = p * n
```

The effect should then be that our wallet loses that number of lovelace, and gains the tokens we buy. Here we see the *deposit* function for the first time. It is the opposite of the *withdraw* function.

```
withdraw w $ lovelaceValueOf l
deposit w $ assetClassValue (tokens Map.! v) n
```

Finally we update the model state by adding the lovelace and removing the bought tokens.

```
(tsModel . ix v . tssLovelace) $~ (+ l)
(tsModel . ix v . tssToken)    $~ (+ (- n))
```

And we wait for one slot.

Finally, the *Withdraw* action.

```
nextState (Withdraw v w n l) = do
when (v == w) $ do
  m <- getTSState v
  case m of
    Just t
      | t ^ . tssToken >= n && t ^ . tssLovelace >= l -> do
        deposit w $ lovelaceValueOf l <> assetClassValue (tokens Map.! w) n
        (tsModel . ix v . tssLovelace) $~ (+ (- l))
        (tsModel . ix v . tssToken) $~ (+ (- n))
      _ -> return ()
wait 1
```

This is only possible if the wallet wanting to withdraw is the same as the wallet running the sale. We check this first, then get the contract state.

We check both that there are enough tokens for us to withdraw the tokens that we are requesting, and also that there are enough lovelace for us to withdraw the lovelace that we are requesting. If this is satisfied, the effect is that we add the lovelace and the tokens to the wallet, and the model is updated to reflect the fact that the tokens and the lovelace have been removed.

That completes the *nextState* function declarations.

Right now, the model is just a conceptual model that has nothing to do with the contracts we wrote earlier. The names are suggestive because they have same names as we used in the redeemer, but there is no link yet between the model and the actual contracts.

The link is provided by yet another method in the *ContractModel* class that we have to implement, and that's the *perform* function.

```
perform
  :: ContractModel state =>
    HandleFun state
  -> ModelState state
  -> Action state
  -> Plutus.Trace.Emulator.EmulatorTrace ()
```



It takes something called *HandleFun* and then it takes the *ModelState* and the *Action*.

The *HandleFun* parameter gives us access to the contract handles.

Let's look at our implementation of this method. We don't need access to the *ModelState* for this example.

```
perform h _ cmd = case cmd of
  (Start w)      -> callEndpoint @"start"      (h $ StartKey w) (nftCurrencies Map.! w, tokenCurrencies Map.! w, tokenNames Map.! w) >> delay 1
  (SetPrice v w p) -> callEndpoint @"set price" (h $ UseKey v w) p
  (AddTokens v w n) -> callEndpoint @"add tokens" (h $ UseKey v w) n
  (BuyTokens v w n) -> callEndpoint @"buy tokens" (h $ UseKey v w) n
  (Withdraw v w n l) -> callEndpoint @"withdraw" (h $ UseKey v w) (n, l)
```

Here we are linking actions to contract endpoints. Recall that we wrote functions that create keys that uniquely identify contracts. The functions were called *StartKey* and *UseKey*.

The *StartKey* function takes one *Wallet* as an argument, and you can see that we give that argument here in the first line of the body of the function. Then we apply the function *h* to it. The function *h* is the *HandleFun* parameter and it is the job of this function to get a handle to the contract instance associated with a given key.

We also pass in the parameters. So in the example of the start action, we pass in a pre-computed values for the NFT, the token currencies and the token names. We will say later how the *instanceSpec* function links the *StartKey* to *startEndpoint'*, the primed version of the function, which takes those three parameters.

The *delay* function used in *perform* is another simple helper function to wait for a number of slots.

```
delay :: Int -> EmulatorTrace ()
delay = void . waitNSlots . fromIntegral
```

All the other actions are very similar, but note that they all use *UseKey* instead of *StartKey*.

Finally, the last method we must provide for the *ContractModel* instance is *precondition*. This allows us to define the conditions under which it is acceptable to provide each action.

```
precondition :: ContractModel state => ModelState state -> Action state -> Bool
```

The precondition for *Start* is that the token sale has not yet started. It says that, given a certain state *s* and the *Start w* action, check that the return value of *getTSSState' s w* is *Nothing*.

```
precondition s (Start w) = isNothing $ getTSSState' s w
```

And for the others, we do the opposite. They are only possible if the token sale has started.

```
precondition s (SetPrice v _ _) = isJust $ getTSSState' s v
precondition s (AddTokens v _ _) = isJust $ getTSSState' s v
precondition s (BuyTokens v _ _) = isJust $ getTSSState' s v
precondition s (Withdraw v _ _ _) = isJust $ getTSSState' s v
```

One last thing, we must link the keys to actual contracts. We do this with the *instanceSpec* function.

```
instanceSpec :: [ContractInstanceSpec TModel]
instanceSpec =
  [ContractInstanceSpec (StartKey w) w startEndpoint' | w <- wallets] ++
  [ContractInstanceSpec (UseKey v w) w $ useEndpoints $ tss Map.! v | v <- wallets, w
    <- wallets]
```

(continues on next page)

The *instanceSpec* function returns a list of *ContractInstanceSpec* types.

A *ContractInstanceSpec* takes three arguments - the first is the key, the second is the wallet, and the third is the contract that is supposed to be invoked.

For the start endpoint, we generate a *ContractInstanceSpec* for each wallet.

For the use endpoint, we generate a *ContractInstanceSpec* for all combinations of two wallets. Note also that the *useEndpoints* function takes an argument of type *TokenSale*, so we need to get this from Wallet *v* and pass it in.

And finally (honestly), we can define a QuickCheck property.

There's a function in `Plutus.Contract.Test` called *propRunActionsWithOptions*.

```
propRunActionsWithOptions
:: ContractModel state =>
  Plutus.Contract.Test.CheckOptions
  -> [ContractInstanceSpec state]
  -> (ModelState state -> Plutus.Contract.Test.TracePredicate)
  -> Actions state
  -> Property
```

First it takes the *CheckOptions* type that we have seen before when we did emulator trace testing. Next it takes the list of *ContractInstanceSpecs* that we defined above. Then it takes a function from *ModelState* to *TracePredicate*, which allows us to insert additional tests. And finally, it produces a function from a list of *Actions* to *Property*. *Property* is like a beefed-up *Bool*, which has additional capabilities, mostly for logging and debugging.

We use this in the *prop\_TS* function. For options we use the same as before which allows us to specify the initial coin distributions. We give each wallet 1,000 Ada, the wallet's NFT and 1,000 of both tokens, A and B.

For the second argument we provide the *instanceSpec* function. For the third argument, we don't add any additional checks.

```
prop_TS :: Actions TModel -> Property
prop_TS = withMaxSuccess 100 . propRunActionsWithOptions
  (defaultCheckOptions & emulatorConfig .~ EmulatorConfig (Left d))
  instanceSpec
  (const $ pure True)
  where
    d :: InitialDistribution
    d = Map.fromList $ [ ( w
                        , lovelaceValueOf 1000_000_000 <>
                          (nfts Map.! w) <>
                          mconcat [assetClassValue t tokenAmt | t <- Map.elems tokens])
                      | w <- wallets
                      ]
```

This results in a type

```
Actions TModel -> Property
```

And this is something that QuickCheck can handle.

Let's look at a sample of *Actions TModel*

```

Prelude Test.QuickCheck Plutus.Contract.Test.ContractModel Spec.Model> sample (arbitrary_
↳ :: Gen (Actions TSMModel))
Actions []
Actions []
Actions []
Actions []
Actions []
Actions
  [Start (Wallet 1),
   AddTokens (Wallet 1) (Wallet 2) 8,
   Withdraw (Wallet 1) (Wallet 2) 5 1,
   Withdraw (Wallet 1) (Wallet 1) 7 2,
   SetPrice (Wallet 1) (Wallet 1) 0,
   Start (Wallet 2),
   BuyTokens (Wallet 2) (Wallet 1) 2]
Actions
  [Start (Wallet 1)]
Actions
  [Start (Wallet 2),
   Withdraw (Wallet 2) (Wallet 1) 4 5,
   SetPrice (Wallet 2) (Wallet 2) 5,
   BuyTokens (Wallet 2) (Wallet 2) 3,
   Start (Wallet 1),
   BuyTokens (Wallet 1) (Wallet 1) 14,
   Withdraw (Wallet 1) (Wallet 1) 11 7,
   AddTokens (Wallet 2) (Wallet 1) 12]
Actions
  [Start (Wallet 1),
   AddTokens (Wallet 1) (Wallet 2) 1,
   BuyTokens (Wallet 1) (Wallet 1) 11,
   SetPrice (Wallet 1) (Wallet 2) 5,
   Withdraw (Wallet 1) (Wallet 1) 10 6,
   Withdraw (Wallet 1) (Wallet 2) 13 0,
   BuyTokens (Wallet 1) (Wallet 1) 8,
   Withdraw (Wallet 1) (Wallet 2) 6 14,
   SetPrice (Wallet 1) (Wallet 2) 7,
   BuyTokens (Wallet 1) (Wallet 2) 4,
   AddTokens (Wallet 1) (Wallet 2) 3]
Actions
  [Start (Wallet 1),
   BuyTokens (Wallet 1) (Wallet 2) 10]
Actions
  [Start (Wallet 1),
   SetPrice (Wallet 1) (Wallet 2) 14,
   BuyTokens (Wallet 1) (Wallet 1) 20,
   BuyTokens (Wallet 1) (Wallet 2) 15,
   Start (Wallet 2),
   Withdraw (Wallet 2) (Wallet 2) 14 1,
   AddTokens (Wallet 2) (Wallet 2) 4,
   Withdraw (Wallet 2) (Wallet 1) 21 2,
   SetPrice (Wallet 2) (Wallet 1) 8,
   Withdraw (Wallet 1) (Wallet 2) 15 17,
   SetPrice (Wallet 1) (Wallet 1) 2,

```

(continues on next page)

(continued from previous page)

```
BuyTokens (Wallet 2) (Wallet 1) 4]
```

We notice here a similar pattern to before, where things start quite simply and get more complex as the list goes on.

So what will be tested? As we saw in the diagram back at the beginning, for all these randomly-generated action sequences, it will test that the properties we specified in the model - how the funds flow - corresponds to what actually happens in the emulator. If there is a discrepancy, the test will fail.

Let's use it!

```
Prelude Test.QuickCheck Plutus.Contract.Test.ContractModel Spec.Model> test
(21 tests)
```

It takes quite a while.

```
Prelude Test.QuickCheck Plutus.Contract.Test.ContractModel Spec.Model> test
(27 tests)
```

But it will run 100 if you let it complete.

What might be more interesting would be to implement a bug in the code and see if these tests will find it.

In the *transition* function of our *TokenSale* code, let's forget to check that only the seller can change the price.

```
transition :: TokenSale -> State Integer -> TSRedeemer -> Maybe (TxConstraints Void Void,
↳ State Integer)
transition ts s r = case (stateValue s, stateData s, r) of
  (v, _, SetPrice p) | p >= 0 -> Just ( mempty -- Just ( Constraints.mustBeSignedBy_
↳ (tsSeller ts)
  , State p $
    v <>
    nft (negate 1)
  )
  ...
```

We need to reload the code.

```
Prelude Test.QuickCheck Plutus.Contract.Test.ContractModel Spec.Model> :l test/Spec/
↳ Model.hs
```

Ok, one module loaded.

```
Prelude Test.QuickCheck Plutus.Contract.Test.ContractModel Spec.Model> test
```

```
*** Failed! Assertion failed (after 13 tests and 2 shrinks)...
```

You will see a whole bunch of output, but at the top, you will see clearly the action sequence that led to the bug.

```
Actions
[Start (Wallet 2),
 SetPrice (Wallet 2) (Wallet 1) 12,
 AddTokens (Wallet 2) (Wallet 2) 11,
 BuyTokens (Wallet 2) (Wallet 2) 1]
Expected funds of W2 to change by
Value (Map [(02,Map [("NFT",-1)]),(bb,Map [("B",-10)])])
(excluding 29466 lovelace in fees)
but they changed to
```

(continues on next page)

(continued from previous page)

```
Value (Map [(Map [("",-12)]),(02,Map [("NFT",-1)]),(bb,Map [("B",-10)])])
Test failed.
```

And we see that Wallet 1 has tried to set the price of the token sale that was started by Wallet 2. This should result in no change, because Wallet 1 is not allowed to do this.

The model believes that the price should still be zero, but in the emulator the price has been set to 12.

Then Wallet 2 adds 11 tokens, and then buys 1 token from itself.

According to the model, the tokens should be free. The model expects that the wallet loses the NFT, that the wallet also loses 10 “B” because it gave 11 and then bought 1 back, and that there is no change in Ada in the wallet, because the token price is zero.

But in the emulator, setting the price did have an effect, and so it reports that the wallet lost 12 lovelace.

So the discrepancy in the flow of funds has been found, and QuickCheck reports the error.

By default this is all the QuickCheck test do. It only checks the flow of funds, whether the emulator and the model agree at each point. It is, however, possible to add additional checks. And it is also possible to influence the action sequences so that we can specify certain flows of actions to steer the tests in certain directions. That is called Dynamic Logic, and that is yet another monad.

Even though this is very powerful, it also has its limitations. For one, it only tests the contracts that we provide. It doesn’t test all possible off-chain code. It is possible that some party could write their own off-chain code that would allow them to steal funds from our contract, and this QuickCheck model can’t test for that.

The second problem is concurrency. We added this delay of one slot to each action to make sure that everything is nicely sequenced. Of course, in a real blockchain or in an emulator, wallets can have concurrent submissions of transactions. In principle we could try to do that with this model as well, but then we would need to somehow specify in the model what should happen in each case and that could get very complicated.

We should quickly look at how this integrates with Tasty.

There is a function in the Tasty library called *testProperty* that takes, as one its arguments, a QuickCheck property.

```
tests :: TestTree
tests = testProperty "token sale model" prop_TS
```

You will see an additional stanza in this week’s cabal file

```
test-suite plutus-pioneer-program-week08-tests
type: exitcode-stdio-1.0
main-is: Spec.hs
...
```

And, if we look at the referenced Spec.hs

```
main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "token sale"
  [ Spec.Trace.tests
  , Spec.Model.tests
  ]
```

We can see that it specifies a list of test modules. And these can be run from the command line with the following command.

```
cabal test
```

## WEEK 09 - MARLOWE

---

**Note:** This is a written version of [Lecture #9](#).

In this lecture we cover Marlowe - a special-purpose language for financial contracts on Cardano.

---

### 9.1 Overview

In the previous lectures we have learnt about all the important ingredients for writing a Plutus application.

We have first looked at the extended UTxO model - the accounting model that Cardano uses - and the additions that Plutus brings to it.

Then we have talked about on-chain validation, minting policies, writing off-chain code, we have seen how to deploy smart contracts and also how to test them.

Plutus is a very powerful language. So powerful, in fact, that you can implement other languages on top of it - you can write an interpreter in Plutus for other languages.

One such language is Marlowe. Marlowe is a Domain Specific Language (DSL) for smart contracts.

For this lecture, Professor Simon Thompson, a very prominent figure in the Haskell community who leads the Marlowe team, and his colleague Alex Nemish will give guest lectures to tell us a bit about Marlowe.

Afterwards we will look at the [Marlowe playground](#) and play with a simple smart contract.

### 9.2 Lecture by Prof. Simon Thompson

Marlowe is a special-purpose language for writing financial contracts on Cardano.

#### 9.2.1 Why do we build DSLs?

One reason is that we want to build languages that are closer to the language of the user and not so much the language of the system. They are designed to be in the specific domain of the application. A financial language will talk about payments, for example.

When we write a DSL, we get some advantages. We can write down things in that domain, but we can't perhaps write as much as we could in a general purpose language. And, if we do work in this more specialised context, we have the advantage of being able to give people better feedback and better error messages. We can also give more guarantees on program behaviour. That's one of the things that will be stressed in this lecture.



## Marlowe

### A SPECIAL-PURPOSE LANGUAGE FOR FINANCIAL CONTRACTS

Designed for users, as well as developers.

Designed for maximum assurance.

## Assurance

### CONTRACTS DO WHAT THEY SHOULD ... ... AND NOT WHAT THEY SHOULDN'T

Language as *simple* as it can be.

Contracts can be *read* and *simulated*.

Before running, can explore *all behaviour*.

System can be *proved safe* in various ways.





## 9.2.2 What kind of assurance can we give?

We can give two kinds of assurance. We can make sure that contracts do what they are supposed to do, but we can also make sure that they don't do what they shouldn't. We will see both aspects of that as we go along.

We've designed the language to be as simple as possible and the implementation reflects that, and we'll talk a bit about that later on. Contracts are nice and readable, and also we can easily simulate them, so we can present to users a very clear picture of how their contract in Marlowe will behave.

In fact, we can do more than that. Because they are particularly restricted, we can explore every possible behavior path that a contract can take, before it is executed. So, we can give complete guarantees about how a contract will behave, not just on one or two tests, but on every possible execution sequence.

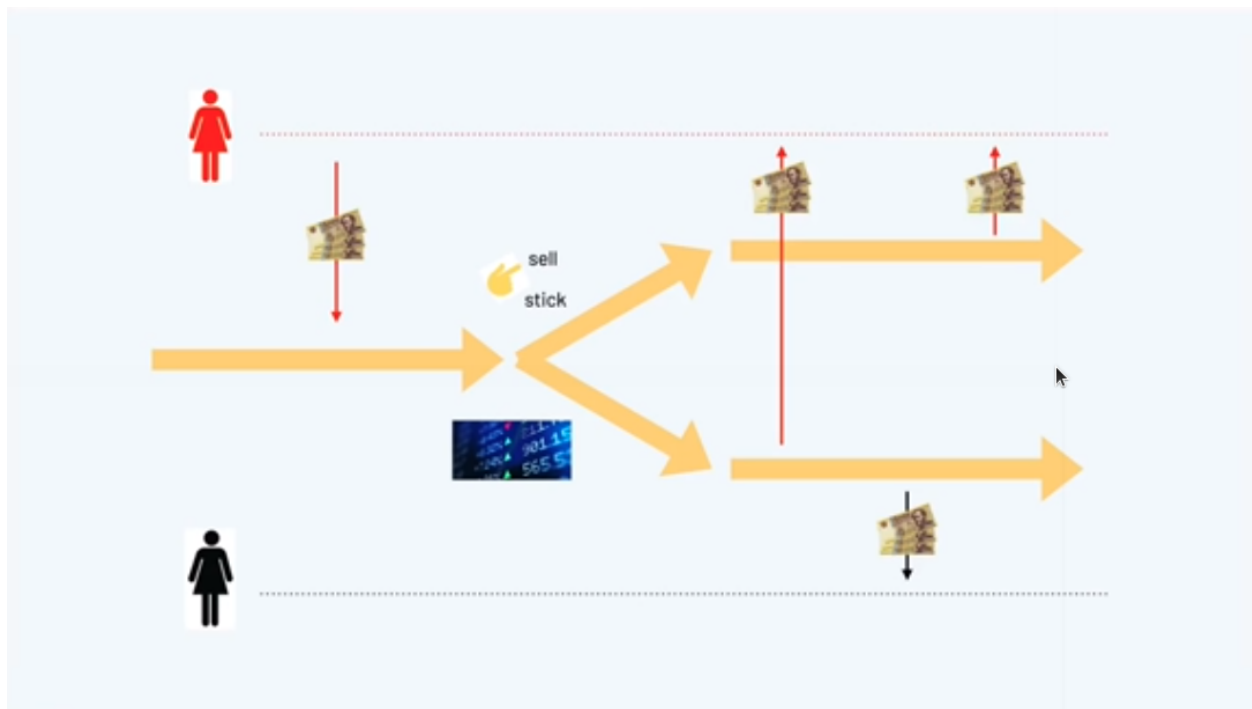
It's also more straightforward to write mathematical proofs of various kinds of safety, so that is the strongest criteria that we can hit in this kind of world; a mathematical proof that the system will do certain things and won't do others.

## 9.2.3 What does a financial contract do?

Let's start by looking at what a financial contract can do.

A contract can accept payments from participants in the contract.

Depending on choices made by one of the participants, it can evolve in different directions.



It can make decisions based on external information such as the information coming from a stock exchange. So, information coming from an oracle can determine the future behaviour of a contract.

A contract can also make payments out. If money has been deposited in the contract, that money can be deposited out to participants.

So we have flows of money and choices according to external factors.

One final thing that we have is that the roles in a contract are things that themselves can be owned. We represent that in Marlowe by minting tokens that represent those roles. That means that we can use those tokens as evidence that

somebody is meant to be playing a role. They are a form of security that a person submitting a transaction is allowed to submit that transaction, but also it means that these roles are tradable. A role could be traded by another person or another contract.

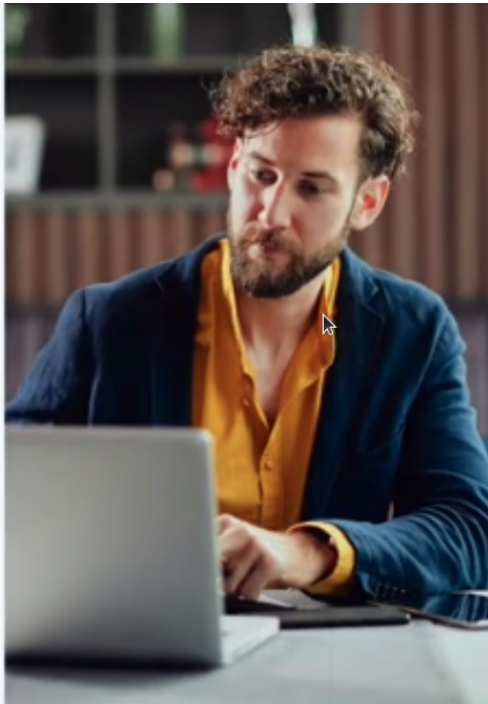
### 9.2.4 Language design

Now let's think about how to design a language based on these ingredients.

A contract could ...

**A CONTRACT IS JUST A PROGRAM  
RUNNING ON A BLOCKCHAIN**

- ... run forever.
- ... wait for an input forever.
- ... terminate holding assets.
- ... "double spend" assets.



When we design a language of contracts, what we are really doing is designing a programming language. A smart contract is just a program running on a blockchain.

A contract could, in principle, run forever. And also, more subtly, it could get stuck waiting for an input forever.

It could terminate while holding assets, locking them up forever.

So there's a whole lot of security issues that a program might have.

## Designed for safety

What we chose to do was to design for safety.

## Contracts are finite

Firstly, contracts are designed to be finite. Their life will be finite, there is no recursion or looping in Marlowe. We will come back to that a bit later on when we talk about Marlowe being embedded in other languages.

## Contracts will terminate

We can be sure that contracts will terminate. We do that by putting timeouts on every external action. Every choice, every deposit of money into the contract comes with a deadline. Marlowe contracts cannot wait forever for somebody to make a choice or for an action to happen. If you hit the timeout then an alternative course is taken.

## No assets retained on close

We've designed the semantics of the language so that when a contract reaches its close, at the end of its lifetime, any money left in the contract will be refunded to participants.

## Conservation of value

Conservation of value is something that we get for free from the underlying blockchain. The blockchain guarantees that we can't double spend and because we are using the transaction mechanisms of the underlying blockchain, we can be sure that we are getting conservation of value.

So this is giving us a lot of guarantees out of the box. These are not guarantees that you get from Plutus contracts in general. A Plutus contract could go on forever, it need not terminate and it could terminate while holding a whole collection of assets which then become unreachable.

## 9.2.5 The Marlowe Language

So what does the language look like? Let's cut to the chase.

Marlowe, at heart, is represented as a Haskell datatype.

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

We have a *Pay* construct. In that a *Party* in the contract makes a payment to a *Payee* of a particular *Value*, and then the contract continues with what we call the continuation contract.

```
Pay Party Payee Value Contract
```

We can go in two separate directions. We can observe *If* a particular *Observation* is true or not. If the observation is true we follow the first *Contract*, if it is false we follow the second *Contract*.

## The Marlowe language

Contracts are finite.

Contracts will terminate ...

... with a defined lifetime.

No assets retained on close.

Conservation of value.

```
data Contract = Close
              | Pay Party Payee Value Contract
              | If Observation Contract Contract
              | When [Case Action Contract]
                    Timeout Contract
              | Let ValueId Value Contract
              | Assert Observation Contract
```

### If Observation Contract Contract

The most complex construct in Marlowe is the *When* construct. It takes three arguments. The first of those is a list of *Contract/Action* pairs - a list of *Cases*.

### When [Case Action Contract] Timeout Contract

What the *When* construct does is wait for one of a number of *Actions*. When one of those *Actions* happens, it performs the corresponding *Contract*. For example, it could be waiting for a deposit. If we have a case where the first part of the pair is a deposit, then we execute the corresponding second part of the pair. Similarly with making a choice or with getting a value from an oracle.

Here we are waiting for external actions and, of course, the contract can't make those actions happen. A contract can't force somebody to make a choice. It can't force somebody to make a deposit. But what we can do is say that if none of these actions takes place then we will hit the *Timeout*, and when we hit the *Timeout*, we will perform the *Contract* represented by the final argument to the *When* construct.

So, we can guarantee that something will happen in the *When* construct, either by one of the actions triggering a successive contract, or we hit the timeout and go to that continuation.

Finally we have the *Close* construct which has the semantics defined so that nothing is retained when we close.

That is the Marlowe language, and we will see that we can use these to construct Marlowe contracts in a variety of ways.

## 9.2.6 The Marlowe Product

So that is the language. What is the Marlowe product itself?

We have a suite of things. First we'll look at the overall vision for Marlowe and then look at where we are in terms of fulfilling that vision.



We have a prototype for Marlowe Run. That is the system through which an end user will interact with contracts running on the Cardano blockchain. You can think of Marlowe Run as the Marlowe dApp. It's the things that allows Marlowe contracts to be executed.

We're also building a market where contracts can be uploaded, downloaded, and where we can provide various kinds of assurance about those contracts.

We allow contracts to be simulated interactively and we call that Marlowe Play. We allow contracts to be built in various different ways and we call that Marlowe Build. In fact what we've done at the moment is bundle those two - Marlowe Play and Build - into what we call the Marlowe Playground.

So as things stand at the moment you can use the Marlowe Playground to simulate and construct Marlowe contracts we're in the process of redesigning the user experience based on what we've done with Marlowe Run.

What we're releasing very shortly is the prototype of Marlowe Run and this is the prototype of how end users will interact with Marlowe on the blockchain. Our intention is that we'll have all these products available running on the Cardano blockchain when we have the full support for this which will involve having the Plutus Application Backend and the wallet back end and so on working as they should.

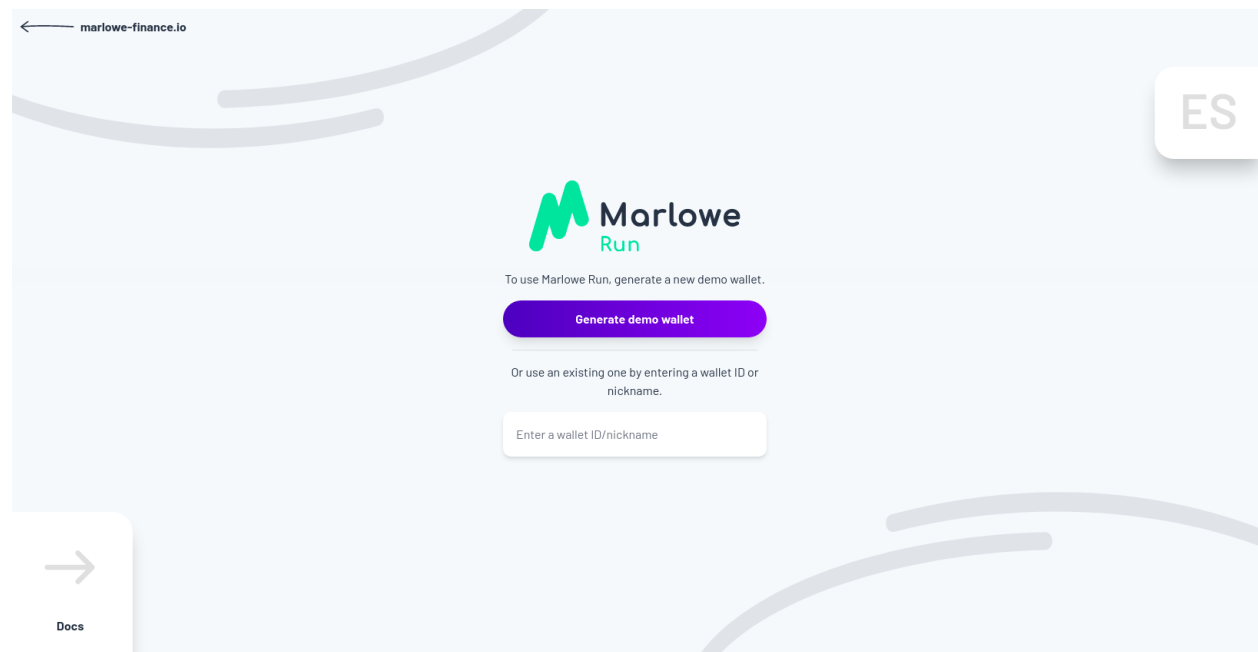
### 9.2.7 Demonstration

We'll now look at a demo of what we have in Marlowe Run to give you a sense of what we can do at the moment in terms of giving users the experience that they will have when Marlowe is running on blockchain. This will be the app that is going to provide that experience.

At the moment it's running locally but in a few weeks' time we will be releasing a version that runs in a distributed fashion on the simulated blockchain. Then, as we go into the end of the year we expect to have it running for real on the Cardano blockchain itself.

You can find the Marlowe Playground at

<https://staging.marlowe-dash.iohkdev.io/>



Marlowe run runs in the browser and what it does is provide the end user interaction with contracts running on the blockchain.

For the moment we're simulating that blockchain inside the browser but eventually this will be the tool you'll use to run contracts for real on Cardano.

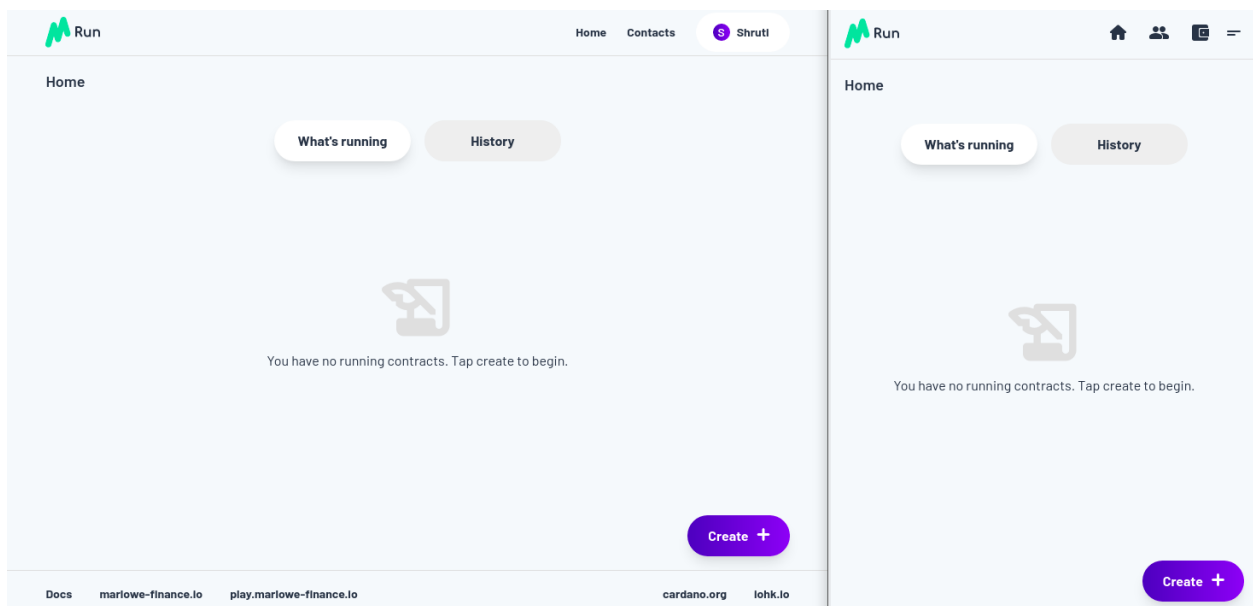
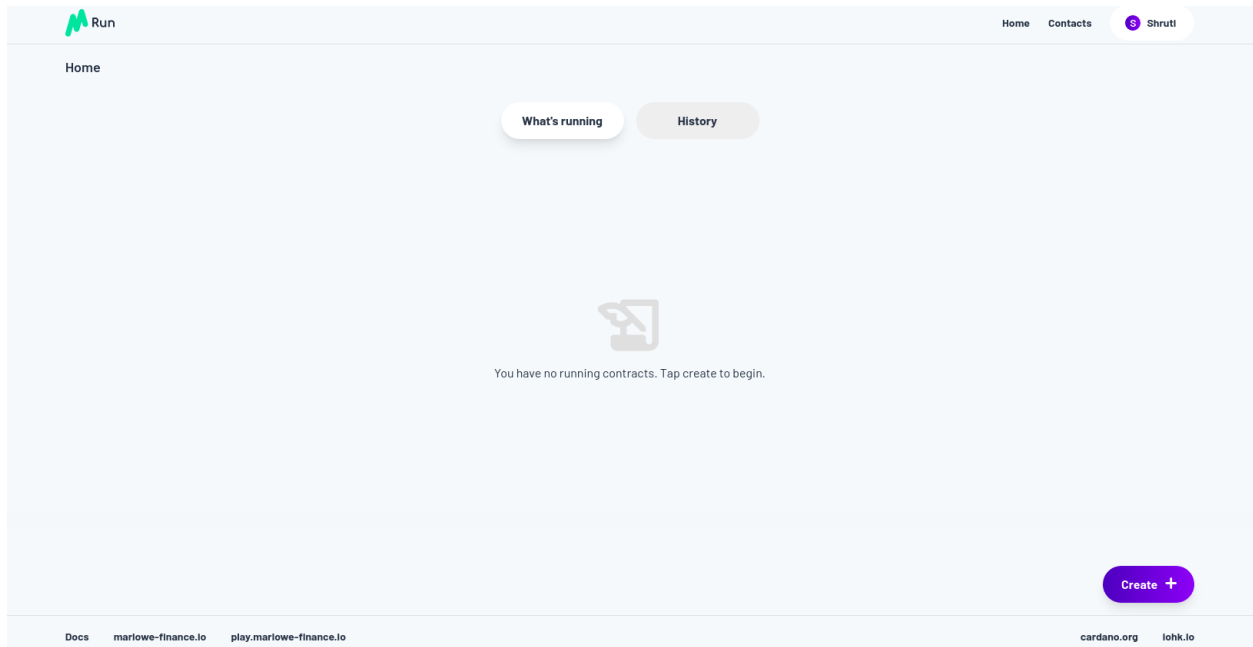
To interact with the contract your wallet needs to be involved to control your signature and to control your assets, so we link up Marlowe to run with a wallet. Let's link it up with Shruti's wallet. You can do this by creating a demo wallet, or by selecting an existing wallet.

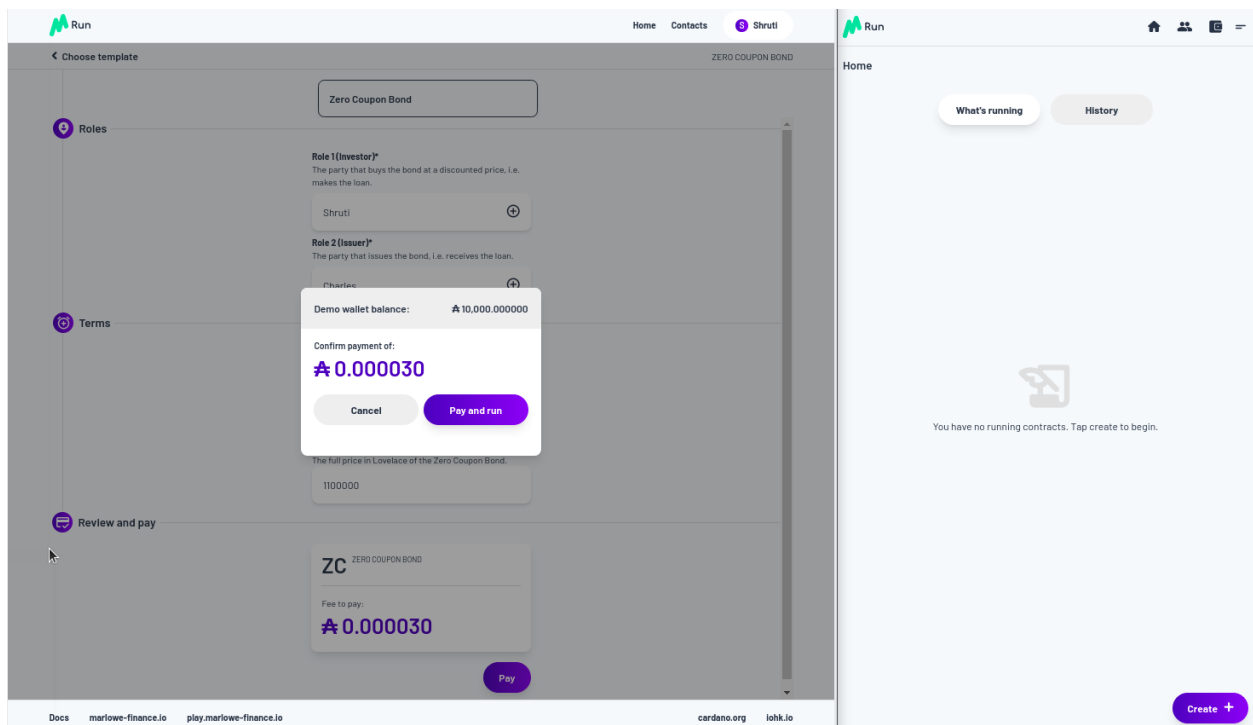
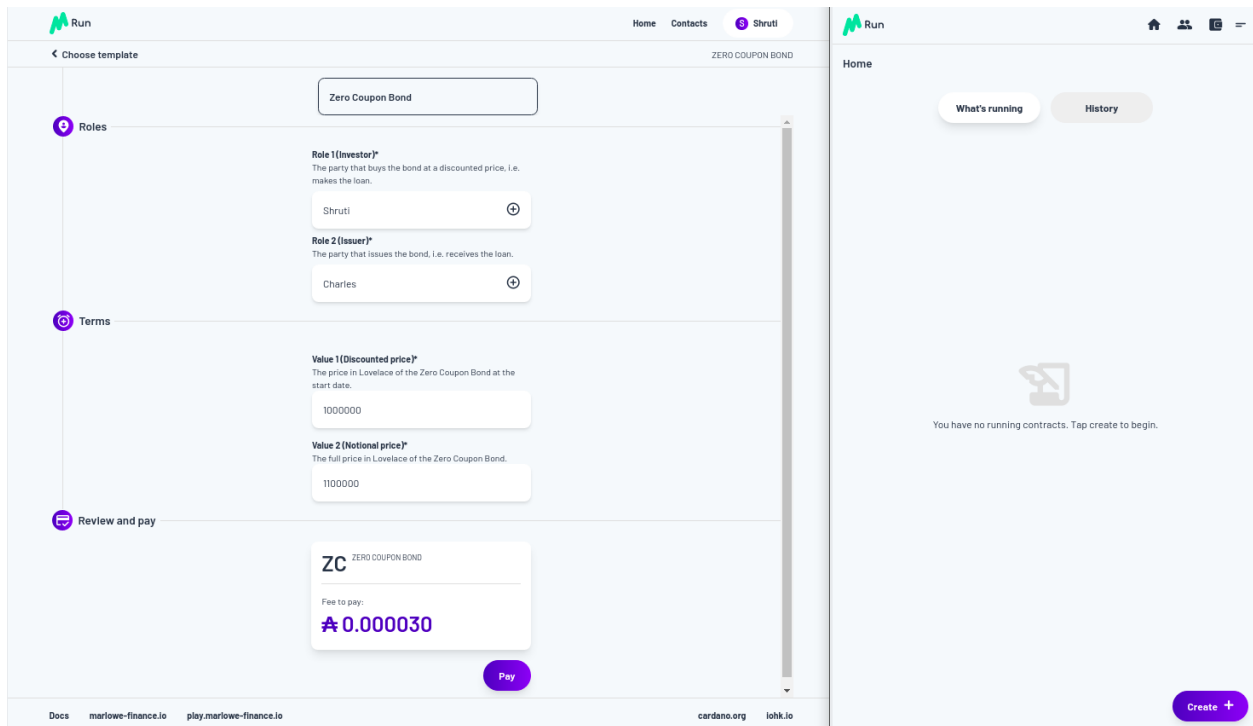
In this window we see the world from Shruti's perspective. Let's open up another window and link that window to the world from Charles's perspective.

At the moment neither of them has any contracts running. They have a blank space, but let's start a contract up. Let's set up a zero coupon bond which is a fancy name for a loan. You can do this by clicking *Create* and selecting the *Zero Coupon Bond* option.

Let's suppose that Shruti is making a loan to Charles. She's the investor he's the issuer of the bond.

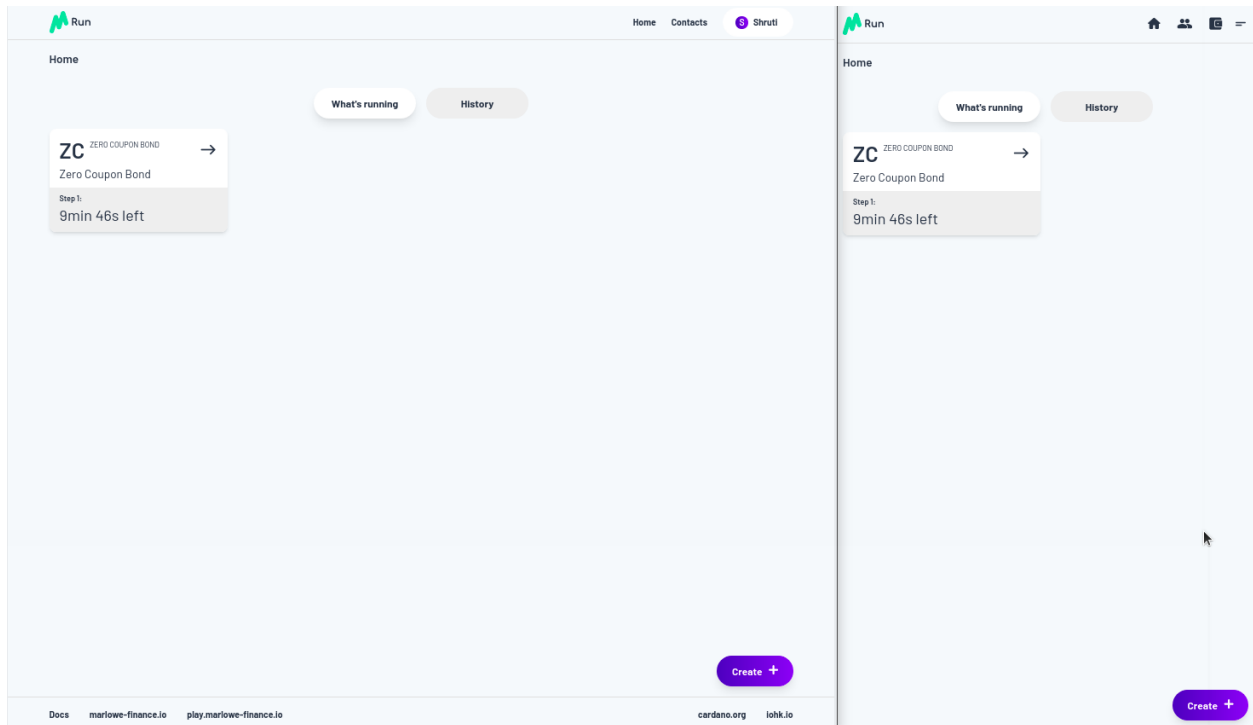
Charles wants to borrow one Ada from Shruti and he's promised to pay back 1.1 Ada. So we've said who the issuer and investor are we said what the price and the eventual value will be and we're now going to create the contract. In order to do that we have to make a payment of 30 lovelace to get the contract started.







So let's pay. We are asked to approve and the payment goes through. You can see now in Shruti's Marlowe Run we've got the Zero Coupon Bond running, but also, if you look at Charles's view of the world, it's running there too for him.



We're at the first step. If we click through on Charles's contract, it's saying that it's waiting for something from the investor, who is Shruti.

So let's see what's happening in her view.

She's being asked to make a deposit so let's click on that to make the deposit.

And click to confirm with a fee of 10 lovelace.

Then you can see her view has changed now she's waiting for the issuer to pay her back.

We look in Charles's view, which is incidentally the mobile view, of Marlowe Run, and he's asked to pay his 1 Ada.

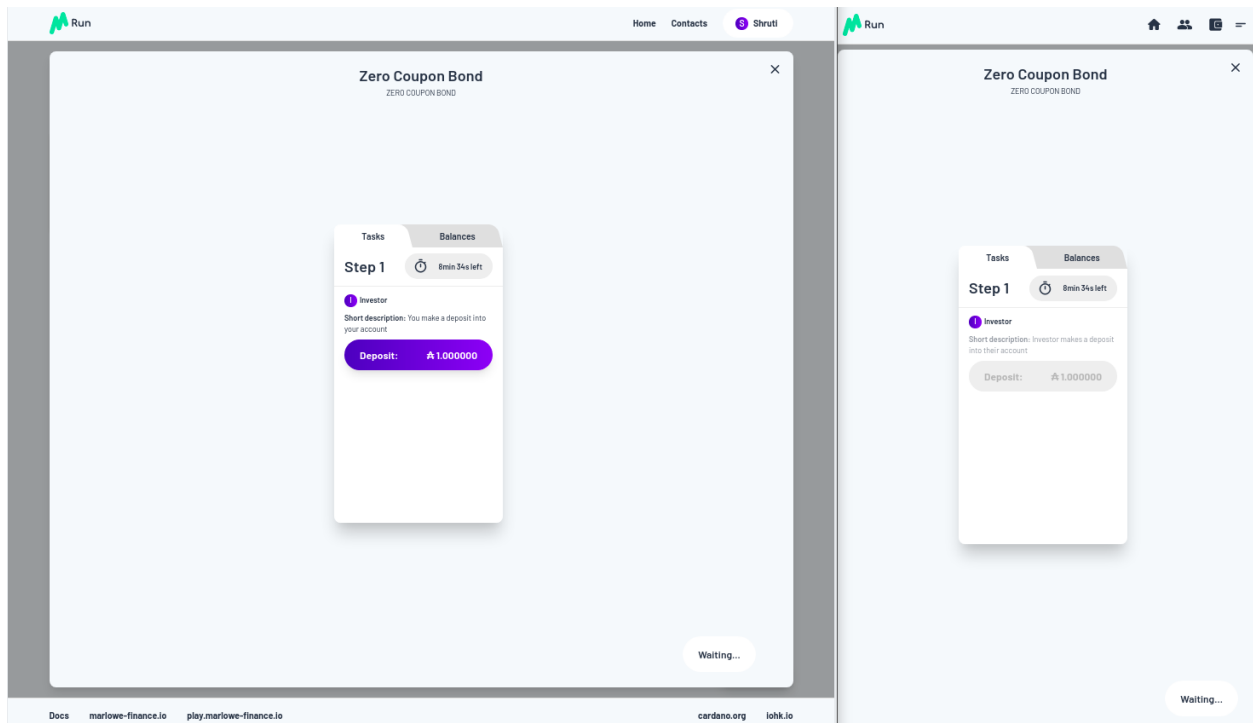
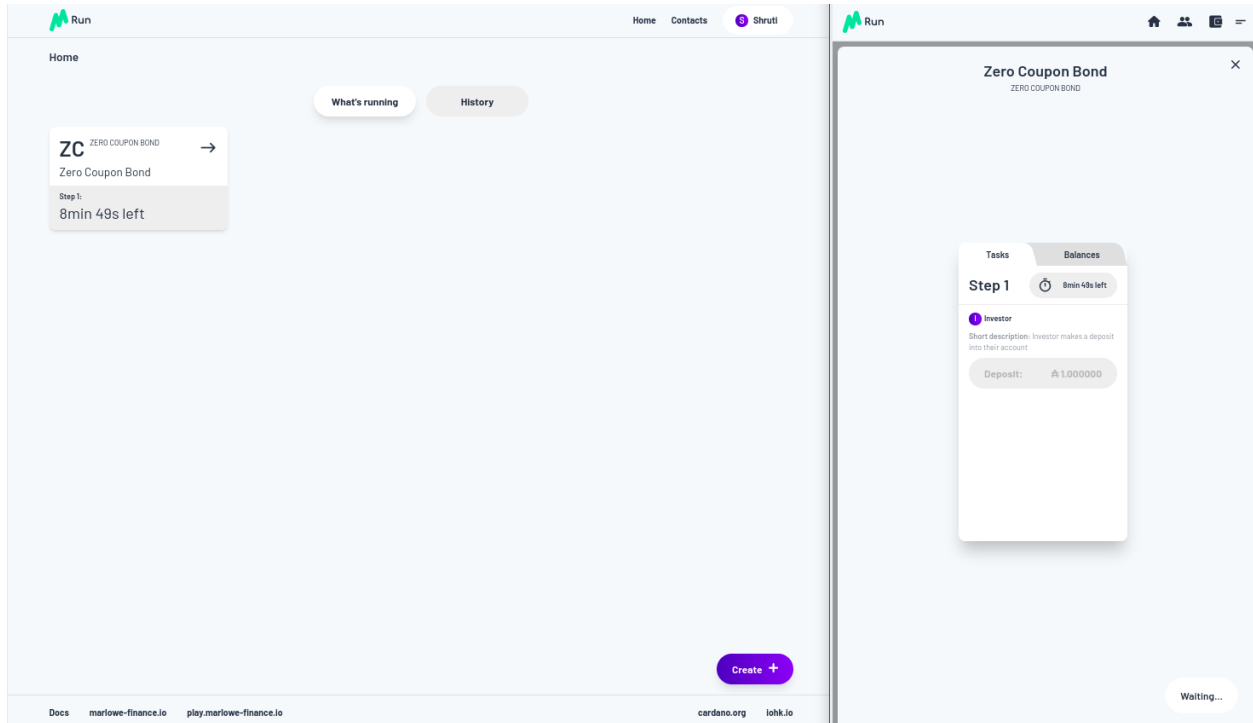
Let's make him do that now. He'll also have to pay a 10 lovelace transaction fee.

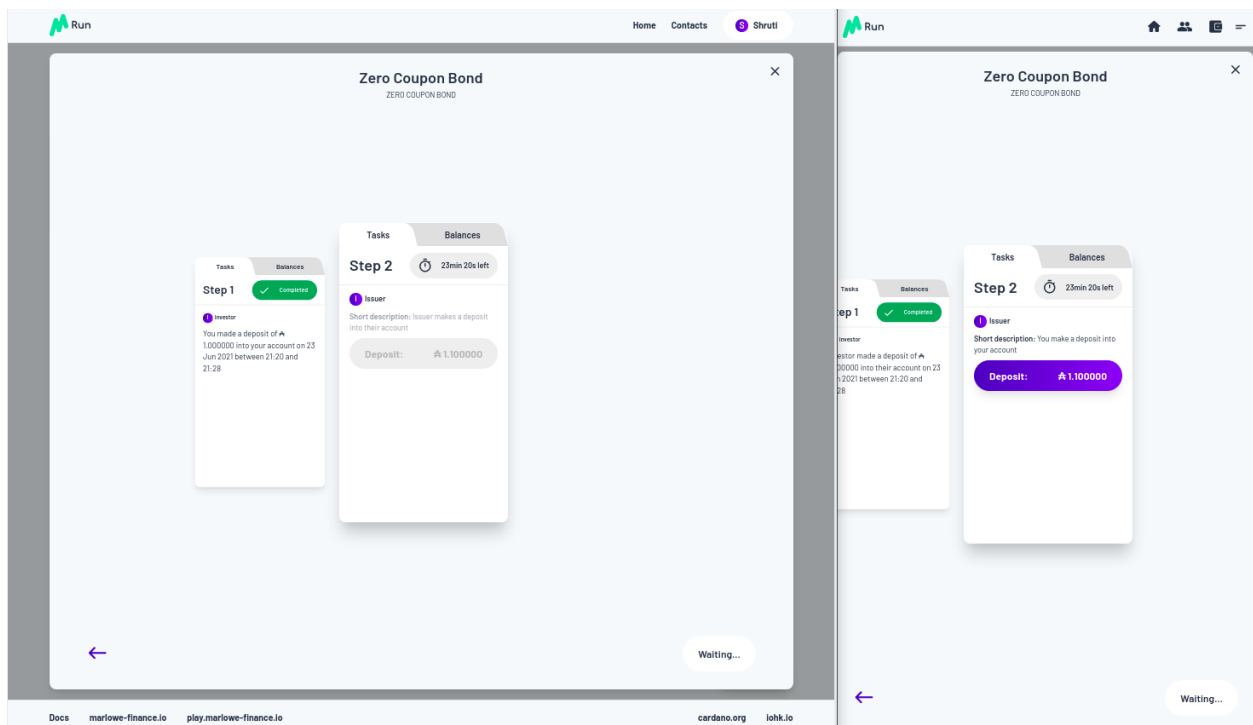
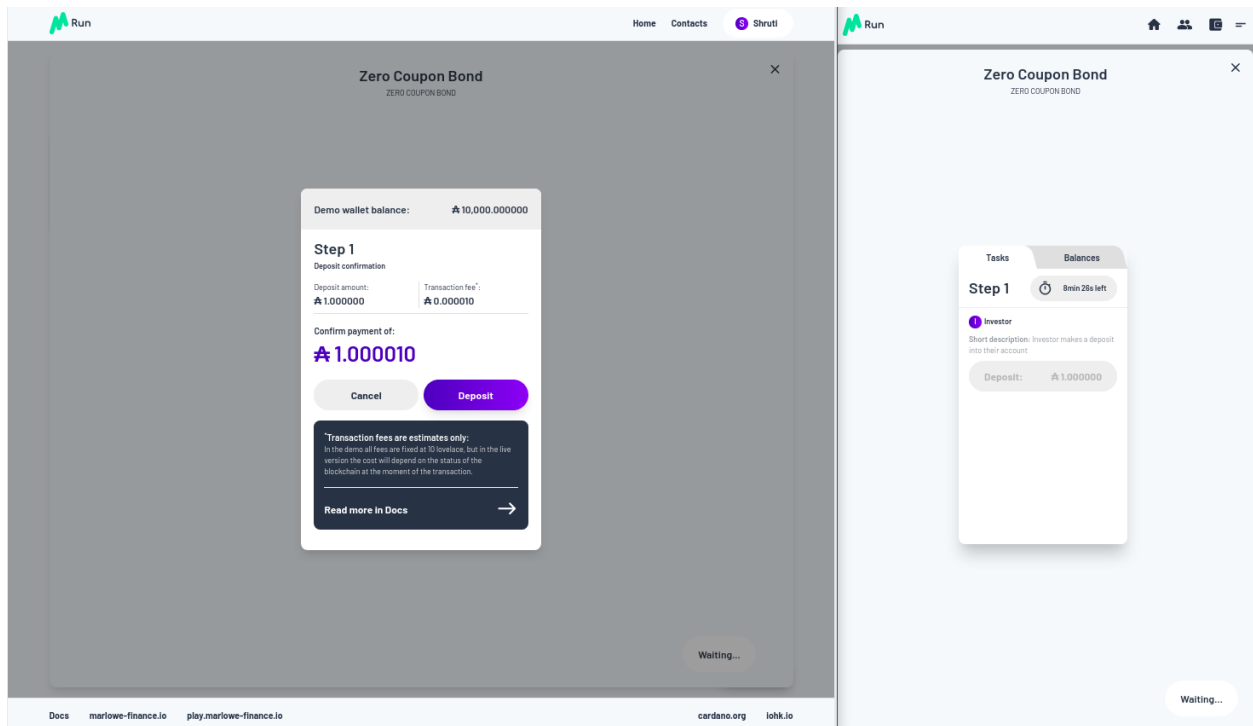
Let's make that deposit.

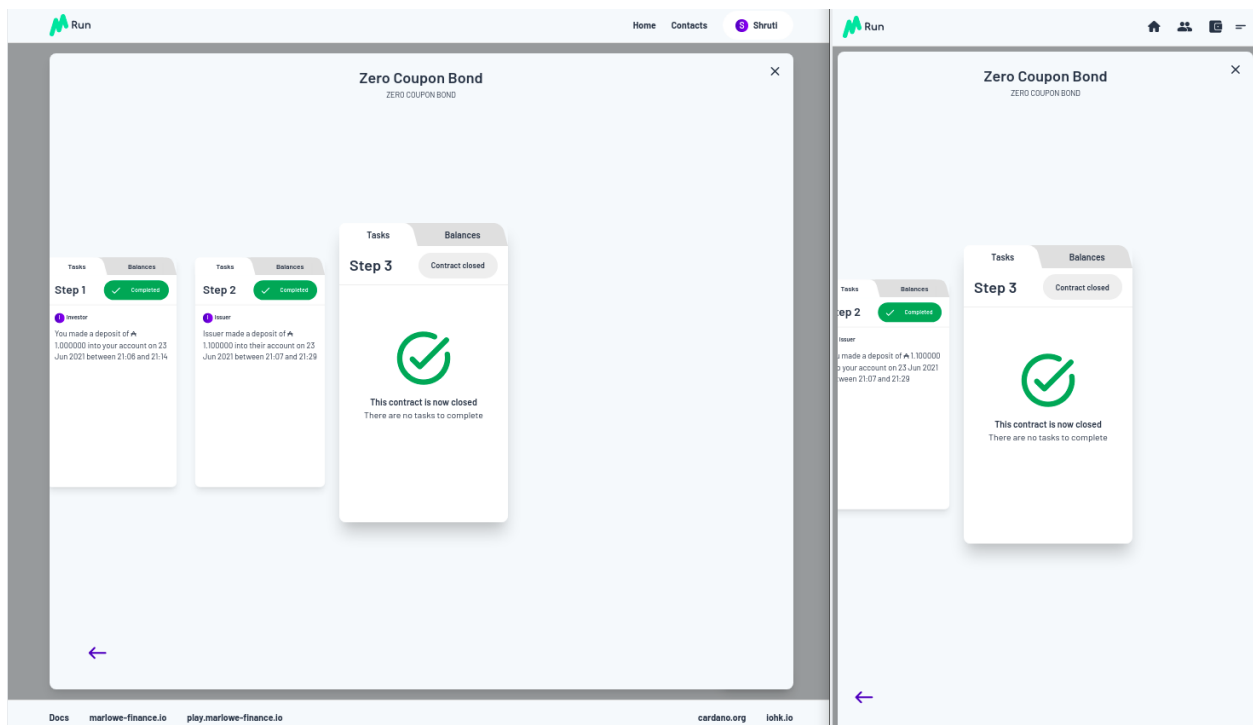
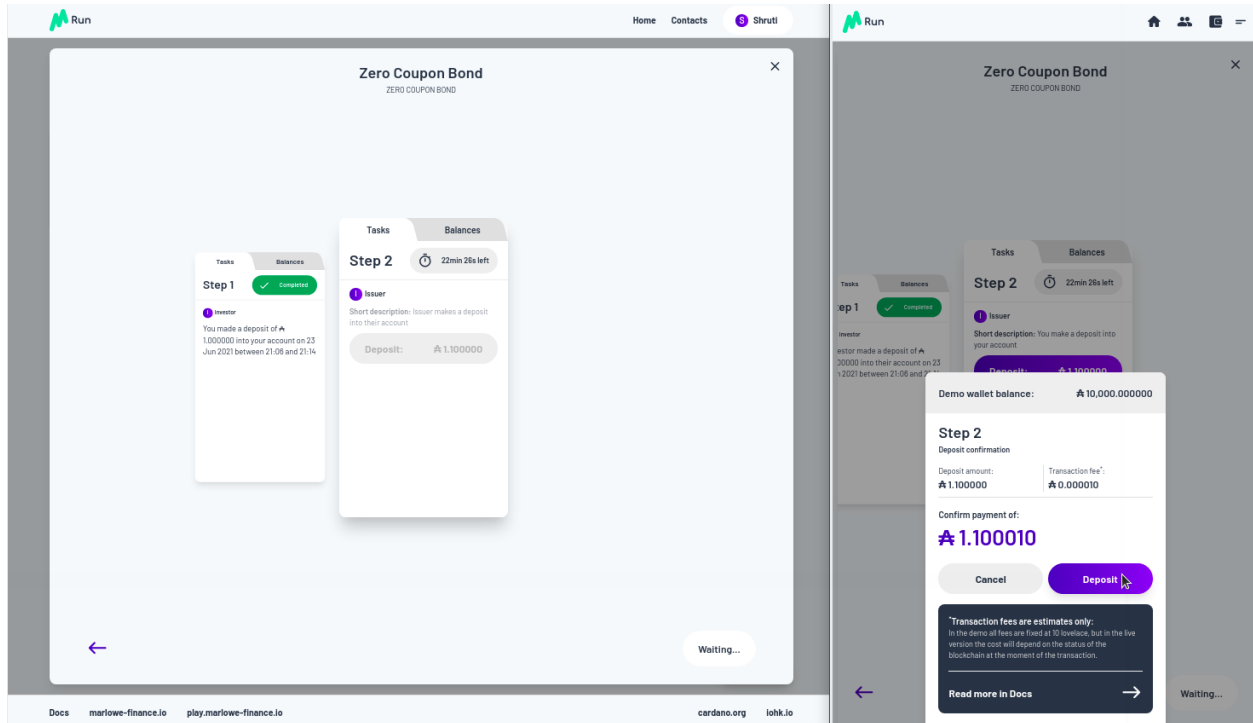
And you see now from both their perspectives that loan is completed you can see the history of what's gone on. You can see, at particular points, the balances that the contract holds.

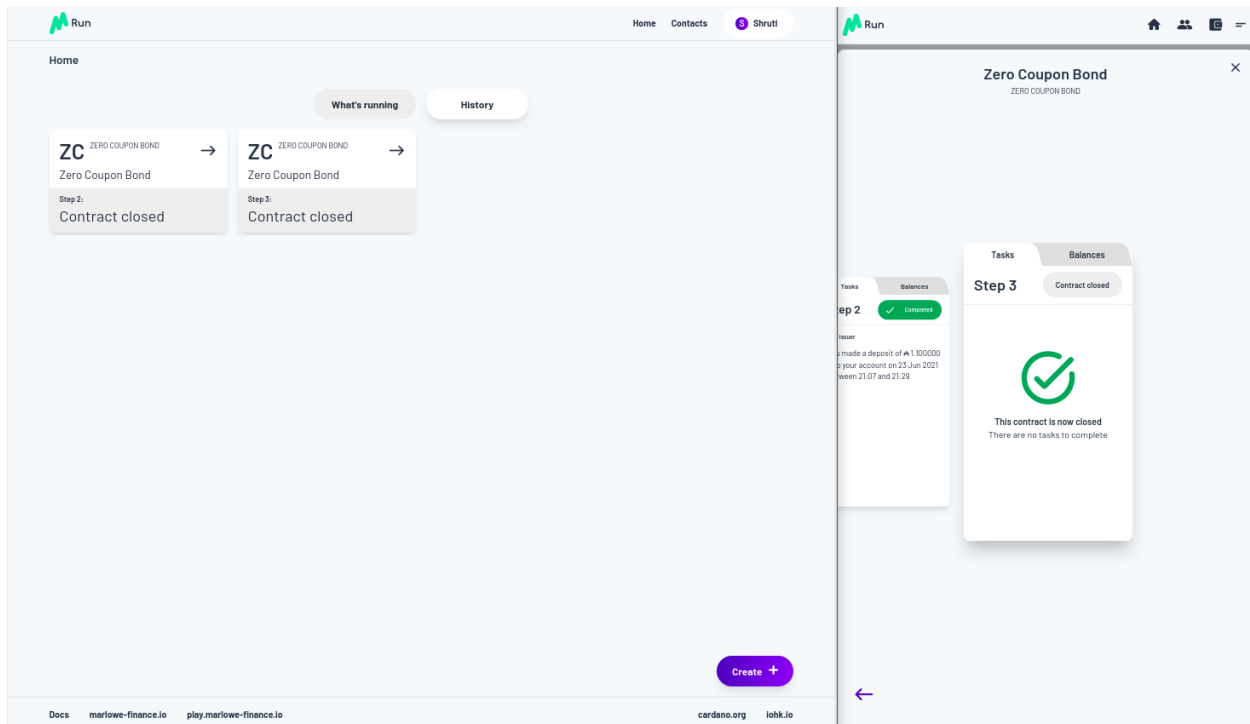
If we close that and select *History*, we can see the history of all the contracts that Shruti has taken part in.

That pretty much covers the basics of what you get from Marlowe Run. It's an intuitive interface to a contract running on the blockchain. You see that each participant in the contract gets their view of the contract in real time, updated from what is, in this case in the browser, but eventually what's on the blockchain.









## 9.2.8 Engineering

Let's now take a look under the hood and see how Marlowe will be executed on Cardano.

Here's a diagram just to give you the context. You'll understand most parts of this diagram already. We have a Cardano root node on which Plutus is running, and as you know, Plutus is a dialect of Haskell, more or less.

Marlowe is embedded in Haskell and Marlowe is executed using Plutus. So Marlowe sits on top of Plutus, but it's also linked to Marlowe Run and has an attachment to a wallet you'll be able to interact with as an end user with a running Marlowe contract.

Also it gets linked to Oracles and so on sitting out there in the real world.

Now, what does it mean to execute a Marlowe contract?

Again this will be familiar to you from Plutus but let's just talk through precisely how it works.

Executing a Marlowe contract will produce a series of transactions on the blockchain. Obviously Plutus running on Cardano checks the validity of transactions. We have a validation function.

The validation function for these Marlowe transactions is essentially a Marlowe interpreter. It checks that the transactions indeed conform to the steps of the Marlowe contract. That's done using the (E)UTxO model, so we pass the current state of the contract and some other information through as datum.

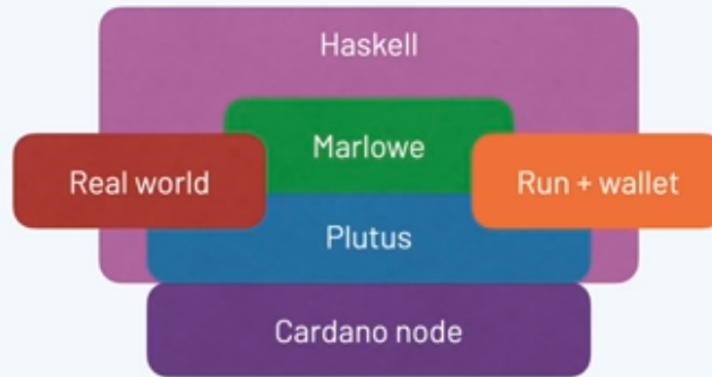
The Marlowe interpreter uses that to ensure that the transactions that are submitted meet the criteria for the particular Marlowe contract.

So that's the on chain part.

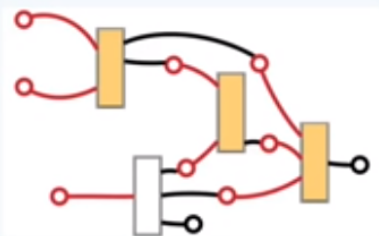
Obviously off chain there's a component as well. So we have to have Marlowe Run and we'll have to build the transactions that meet the validation step on chain.

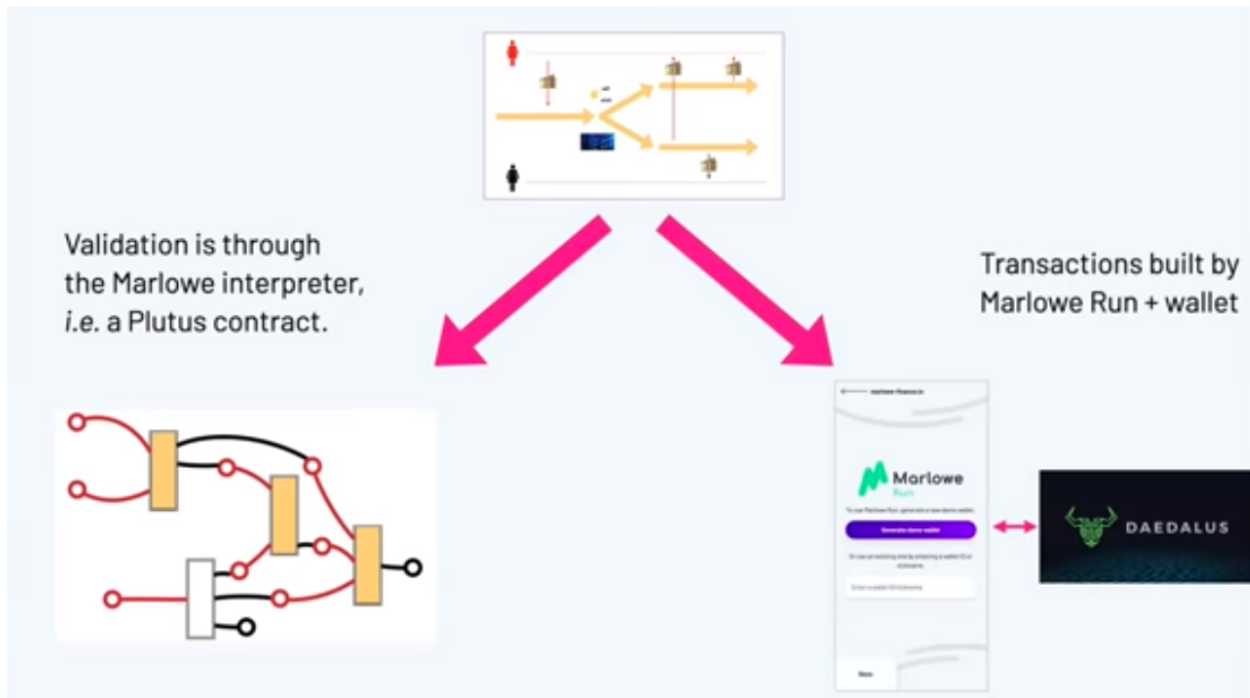
And, if and when the contract requires crypto assets it will have off chain code to ensure that transactions are appropriately signed so that we will have authorization for spending crypto assets.

Using Marlowe run and an associated wallet, we construct the transactions.



Validation is through the Marlowe interpreter, i.e. a Plutus contract.





We get a flow of information in both directions. Marlowe run will submit transactions to the blockchain that then can be validated by the Marlowe interpreter, which is itself a Plutus contract. It's one of the largest Plutus contracts that exists.

But there's also information flow another way because suppose that the transaction I've submitted is a deposit of money into a running contract, and suppose the contract also involves Charles Hoskinson, so my instance of Marlowe Run has submitted that transaction, but Charles also has to be notified about that.

The information flows in the other direction using the companion contract to ensure that every instance of Marlowe Run gets informed about activity in that contract.

Alex will talk some more about the details of the implementation but here you're seeing an outline of how it all how it all works.

Transactions are validated on chain through the interpreter, but they have to be built off chain and in some cases have to be authorized. Essentially the blockchain is the central synchronization point for the distributed system that is the collection of instances of Marlowe Run that are interacting to execute the contract/

You saw in the demo that, in two separate windows, we were sharing information. That was simulating it locally but in production this will be information that's stored on the blockchain.

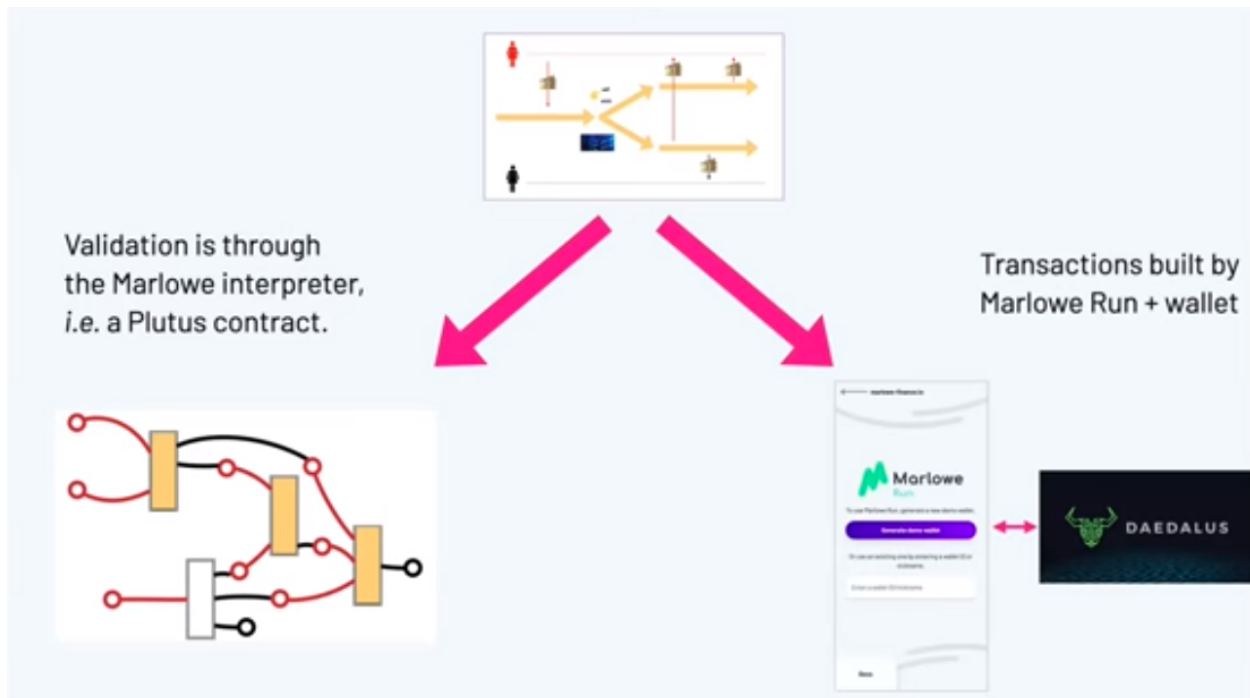
## 9.2.9 System Design

Let's talk a little bit about how the system is designed in a high-level way.

Here's a piece of the semantics of Marlowe, and as you can see it's a Haskell function.

We take an environment, the current state and a contract we executed, and based on what contract that is - a *close* perhaps, or a *pay*, we can reduce we can take some steps of computing the results of that contract.

We do that in a way that uses Haskell in a quite straightforward way to advance the contract. This specification in Haskell is an executable specification of the semantics and this gives us some very nice consequences.



```
-- | Carry a step of the contract with no inputs
reduceContractStep :: Environment -> State -> Contract -> ReduceStepResult
reduceContractStep env state contract = case contract of

  Close -> case refundOne (accounts state) of
    Just ((party, money), newAccounts) -> let
      newState = state { accounts = newAccounts }
      in Reduced ReduceNoWarning (ReduceWithPayment (Payment party money)) newState Close
    Nothing -> NotReduced

  Pay accId payee val cont -> let
    amountToPay = evalValue env state val
    in if amountToPay <= 0
      then Reduced (ReduceNonPositivePay accId payee amountToPay) ReduceNoPayment state cont
      else let
        balance      = moneyInAccount accId (accounts state) -- always positive
        moneyToPay    = Lovelace amountToPay -- always positive
        paidMoney     = min balance moneyToPay -- always positive
        newBalance    = balance - paidMoney -- always positive
        newAccs       = updateMoneyInAccount accId newBalance (accounts state)
        warning       = if paidMoney < moneyToPay
          then ReducePartialPay accId payee paidMoney moneyToPay
          else ReduceNoWarning
        (payment, finalAccs) = giveMoney payee paidMoney newAccs
        in Reduced warning payment (state { accounts = finalAccs }) cont
```



We've got a high level description of what the semantics are, and we're doing that through something that is effectively an interpreter. So we're defining at a high level this interpreter in Haskell for Marlowe contracts.

One really nice thing about writing it in this sort of way is that we can be sure we cover all cases because it's obvious if we're missing some cases. Writing it as an interpreter ensures that we will hit cases we need to in describing the semantics.

Also it really helps us to understand the semantics. When you're designing a language you have an abstract idea about what it's going to mean, but there's nothing like having an implementation of it so you can actually run the semantics.

What would it mean if we were to add this construct? What would it mean if we were to modify the semantics in this way?

If we'd written it in a purely logical format, it's difficult to unscramble just from the rules as they're laid out what, precisely, a change in rule might mean.

## Semantics = executable specification in Haskell

Denotational semantics

Definitional interpreter

Completeness

Must cover *all* cases

Engagement

Can run the semantics

What's even nicer is that we can reuse the semantics in a number of different ways.

In the theorem prover Isabelle, we can use the semantics for reasoning and proof and we use pretty much the same semantics because Isabelle uses a functional language as its subject.

We can run the semantics in Plutus. Plutus is more or less Haskell, perhaps not with all the libraries, but we can, in principle at least, build our implementation on blockchain from our semantics, and also we can translate the semantics into PureScript for simulation in the browser.

Now pure script is not the same exactly the same as Haskell. Isabelle's language is not exactly the same as Haskell. How can we be sure that all these versions are the same?

One way of doing it is to extract Haskell code from Isabelle and test the original against this extracted code. We do that on random contracts and that gives us a pretty high level of assurance that the two are the same.

Down the line in our road map we certainly expect to be using a Haskell and Javascript implementation at some point to replace PureScript in the front end so we don't have to write a PureScript version of the semantics when we're doing the off chain interpretation building the transactions to be submitted. We can use the real Haskell implementation by compiling it into Javascript and running that in Marlowe Run in the client code.

## Repurpose the semantics

In Isabelle

For reasoning and proof

In Plutus ( $\approx$  Haskell)

For implementation on blockchain

In PureScript

For browser-based simulation

## Aside: how to verify that these versions are the same?

Extract Haskell code from the Isabelle version.

Test this against the original Haskell version on random contracts.

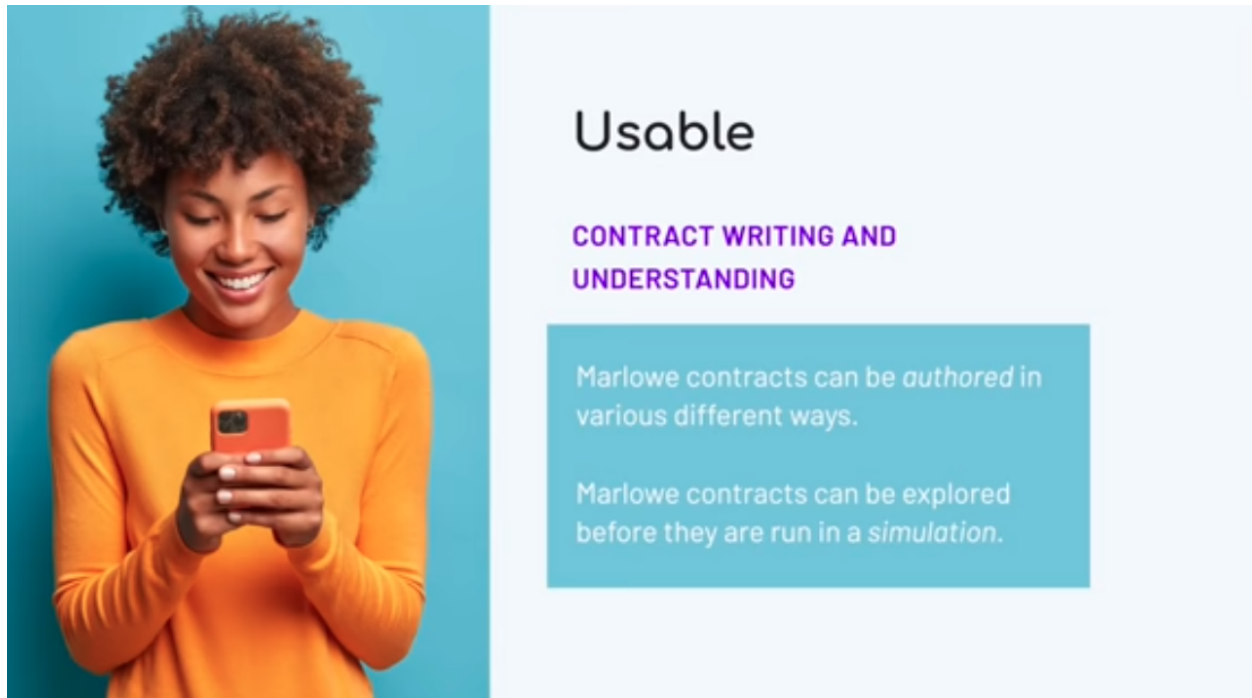
Eventually use a Haskell in JS implementation to replace the PureScript.

So, building the language in Haskell means that though we use various different versions of the semantics, we can get a high level of assurance that these are the same and indeed we can in some situations replace things like the PureScript by Javascript.

### 9.2.10 Usability

That gives us a picture about how the system is put together. Let's go to another aspect of Marlowe. We talked about it being a special purpose language, and that being a DSL promoted usability.

Let's say a bit more about that.



One way we promote usability is that we provide different ways of writing contracts. Another way we promote usability is to allow people to explore interactively how contracts behave before they're actually run in the simulation.

So let's talk about those now.

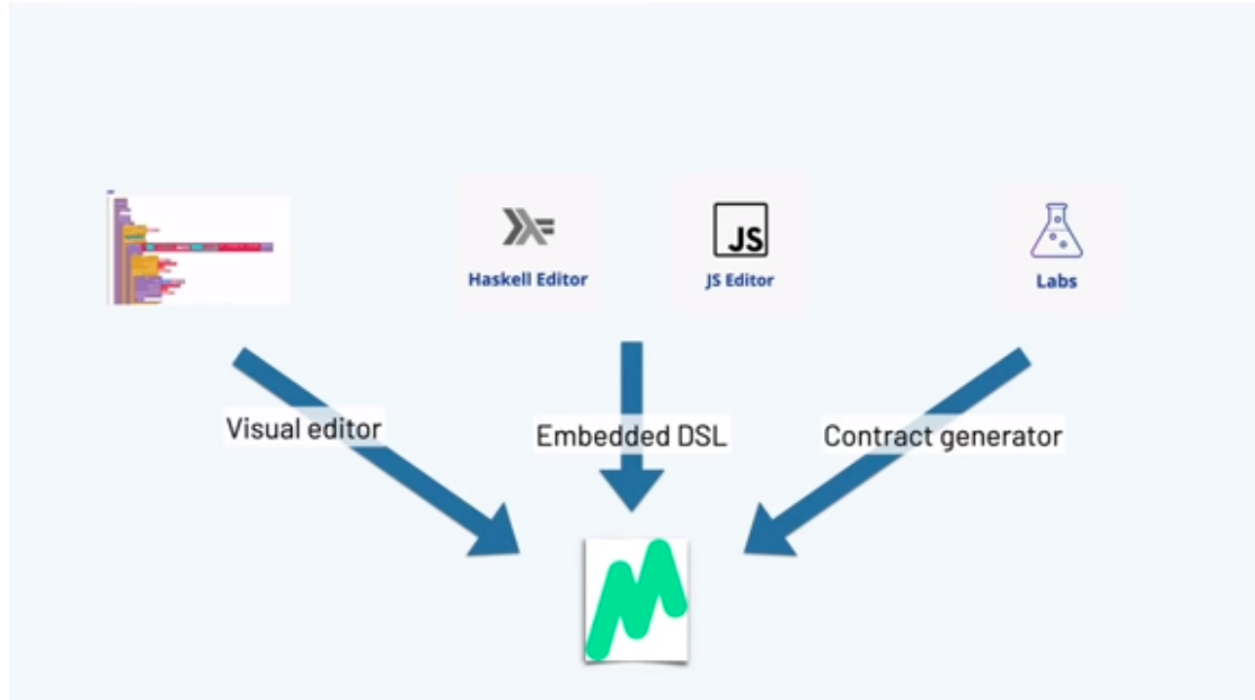
We want to write a Marlowe contract, how can we do it? Well, we can write Haskell using the Marlowe data type as text. That's one way we can do it and that's fine. We have an editor for that inside the playground that supports code completion and will make suggestions and so on.

So we can build the contracts as pure Marlowe, but there are other routes as well.

We have a visual editor for Marlowe so that you can produce Marlowe contracts visually, putting together blocks in a way that doesn't require you to be a confident programmer. You can start off by using the visual version as a way of learning to engage with Marlowe if you are a coder.

Marlowe is embedded in Haskell and in Javascript so we can use facilities like recursion to describe Marlowe contracts. We can say, in Haskell, let's do this particular pattern of behavior a certain number of times. We can write that in Haskell and then for a particular contract we convert the Haskell into Marlowe, and we can also do that for Javascript.

Finally, something we're not going to talk about anymore in this talk is that we can generate contracts from initial conditions. We've been looking at that for the actor standard of financial contracts. On the basis of the contract terms we generate code in Marlowe. We write functions whose output is Marlowe code.



We provide users with a variety of different approaches, leveraging knowledge of Javascript, for example, or leveraging a non-code-based approach for describing the contracts

We also allow people to simulate the behavior of contracts. This is something that you can see in the current version of the Marlowe Playground.

That's something you can play with yourselves. We are looking at different ways of describing the results of a simulation. So at the moment we have a transaction log. We are allowed to choose an next action to perform, you can undo the last step to take you back and then try another path so you can step interactively backwards and forwards through the source code through the application of the contract.

What we're looking at is changing the user interface Marlowe Playground so that we'll use something rather more like the Marlowe Run run description of a running contract.

### 9.2.11 Assurance

We've talked about usability. What about the sort of assurance that Marlowe can give users?

We've seen we've seen that making the system transparent, that making code readable is itself an advantage. We've seen that there's simulation to give people to validate their intuition about a contract.

But rather more formally we can use the power of logic to do two things for us. We can do what's called *static analysis* so we can automatically verify properties of individual contracts. That means we can guarantee this contract will behave as it should, checking every route through the contract.

Also we can do machine-supported proof so, not automatic any longer, written by a user, but we can prove properties of the overall system.

**MARLOWE PLAYGROUND** Escrow with collateral

current slot: 0 expiration slot: 17

**ACTIONS**

Participant **Buyer** "The party that pays for the item on sale..."

Deposit 100,000,000 units of ADA into account of **Seller** as **Buyer**

Other Actions

Move to slot: 10

**TRANSACTION LOG**

Action	Slot
Deposit 1,000,000 units of ADA into account of <b>Seller</b> as <b>Seller</b>	0
Deposit 1,000,000 units of ADA into account of <b>Buyer</b> as <b>Buyer</b>	0

© 2020 IOHK Ltd

**Escrow with collateral**

ESCROW

**Step 1** **Completed**

**Seller**

Seller made a deposit of A 1,000,000 into their account on 31 May 2021 between 08:44 and 08:45

**Step 2** **Completed**

**Buyer**

You made a deposit of A 1,000,000 into your account on 31 May 2021 between 08:45 and 08:46

**Step 3** **Completed**

**Buyer**

You made a deposit of A 1,000,000,000 into Seller's account on 31 May 2021 between 08:45 and 08:47

**Step 4** **Completed**

**Buyer**

You chose 0 for "Everything is alright" on 31 May 2021 between 08:45 and 08:46

**Step 5** **Contract closed**

This contract is now closed. There are no funds to complete.

Next →

# Assurance

## USING THE POWER OF LOGIC

*Static analysis:* automatic verification of properties of individual contracts.

*Verification:* machine-supported proof of system and contract properties.



## Static Analysis

What static analysis allows us to do is check all execution paths through a Marlowe contract. All choices, all choices of slots for a submission of a transaction so we examine every possible way in which the contract might be executed.

The canonical example here is the example of whether a pay construct might fail. Is it possible a pay construct could fail? The answer is that we will use what's called an SMT solver. An SMT is an automatic logic tool - the one we use is called Z3, although others are available. The SMT solver effectively checks all execution parts.

If a property is satisfied that's fine, we get the result. If it's not satisfied, we get a counter example. We get told that there's a way through this contract that leads to a failed payment - a payment that can't be fulfilled. So it gives an example of how it can go wrong, and that's really helpful. It means that if you really want to make sure that a failed payment can't happen, then this gives you a mechanism to understand and to debug how that eventuality can happen, and so gives you a chance to think about how to avoid it.

So, very powerful and entirely push button. You push a button and you get the results.

Here you see a fragment of a Marlowe contract. It's an escrow contract where the contract starts with a deposit of 450 lovelace.

Checking the analysis in the playground, we've got the results. Static analysis could not find any any execution that results in any warning, so that's saying that you're okay - it's not going to give you a warning whatever you do.

But if we change that deposit of 450 lovelace to a deposit of 40 and analyze we then get this warning.

We get a transaction partial payment. We're told that we get to a payment where we're meant to pay 450 units of lovelace but there are only 40 available, and we get given a list of transactions that take us there.

So we're able to see from that how we got to that point, and the problem is that we didn't put enough money in and then we reached a place where we needed to make a payment of 450 lovelace.

So it's easy for us to see that we need to either make the payment smaller or make the initial deposit bigger. As it's entirely push button, we get that sort of assurance for free, as it were.

## Static analysis

Can check *all* execution paths through a Marlowe contract.

All choices, all choices of slots for transaction submission.

Example: is it possible there may not be enough to fulfil a Pay construct?

Constructive: if it is, then here's a counter-example.

The screenshot shows the Marlowe IDE interface. On the left, a code editor displays a Marlowe contract snippet. A pink arrow points to the line `(Constant 450)` within a `(Pay ...)` construct. The contract code includes a `(When ...)` block with a `(Choice ...)` and a `(Pay ...)` construct. The right sidebar shows the contract state for participant 'alice', including a deposit of 450 units of A and a 'Move to slot' field set to 10. Below the code editor, a 'Warning Analysis Result: Pass' message is displayed, stating 'Static analysis could not find any execution that results in any warning.' Two buttons, 'Analyse for warnings' and 'Analyse reachability', are visible below the message.

Participant **alice**  
Deposit 40 units of ADA (Role "alice") as (Role "alice")

Other Actions  
Move to slot:   
Undo

Modelling with Marlowe

Marlowe is designed to execute financial contracts on the blockchain, and specifically on Cardano. Contracts are written in a high-level language, and together a small number of contracts in combination can be used to create many different kinds of contracts.

Warning Analysis Result: Warnings Found

Static analysis found the following counterexample:

- Warnings issued:
  - TransactionPartialPay** - The contract is supposed to make a payment of **450** units of **ADA** from account of (Role "alice") to party (Role "bob") but there is only **40**.
- Initial slot: **0**
- Offending transaction list:
  - Transaction with slot interval **0 to 3** and inputs:
    - IDeposit** - Party (Role "alice") deposits **40** units of **ADA** into account of (Role "alice").
  - Transaction with slot interval **1 to 2** and inputs:
    - IChoice** - Party (Role "alice") chooses number **0** for choice "choice".
  - Transaction with slot interval **1 to 1** and inputs:
    - IChoice** - Party (Role "bob") chooses number **0** for choice "choice".

Analyse for warnings   Analyse reachability

## The system is safe

Prove properties of the Marlowe system once and for all.

*Theorem:* Accounts are never -ve.

*Theorem:* Money preservation:

$$\text{money\_in} = \text{money\_in\_accounts} + \text{money\_out}$$

*Theorem:* Close produces no warnings.

*Theorem:* Static analysis is sound and complete.

And we can do the same for individual contracts and templates too.



But thinking about verification, we can do rather more than that. We can prove properties of the system once and for all.

So, for example, we can prove from the semantics that accounts inside a Marlowe contract never go negative. You can't ever overdraw an account in a Marlowe contract.

We can also prove this theorem of money preservation. We can prove that if we look at all the money that's gone into the contract so far, that's equal to the sum of two things - the amount of money that's held inside the contract plus the amount of money that has been paid out. That gives a clear picture of money preservation.

We're also able to prove other more technical things about the system. For example, that a *Close* construct will never produce any warnings. So, if we're analyzing for warnings, we don't need to worry about *Close* constructs. That allows us to optimize the static analysis.

We're also able to prove that the static analysis, which makes a number of simplifications to speed things up, is sound and complete. That means the static analysis will give us an error warning when the real contract can generate an error warning and it won't give us an error warning if the real contract can't do that.

One thing that we haven't done but is on our road map is to do these sorts of proofs for individual contracts or individual contract templates. Things that we can't necessarily prove with static analysis, we can prove by proving them by hand.

The system is amenable to having these proofs written about it, and they give us the highest level of assurance about how it works.

We've said enough for the moment about Marlowe. Where can you go to find out more?

## More information about Marlowe

The marlowe and plutus github repositories.

The IOHK research library: search for "Marlowe".

Online tutorial in the Marlowe Playground.

Alex's presentation coming up next.

There's a Marlowe GitHub repository that has the semantics and the basics about Marlowe.

<https://github.com/input-output-hk/marlowe>

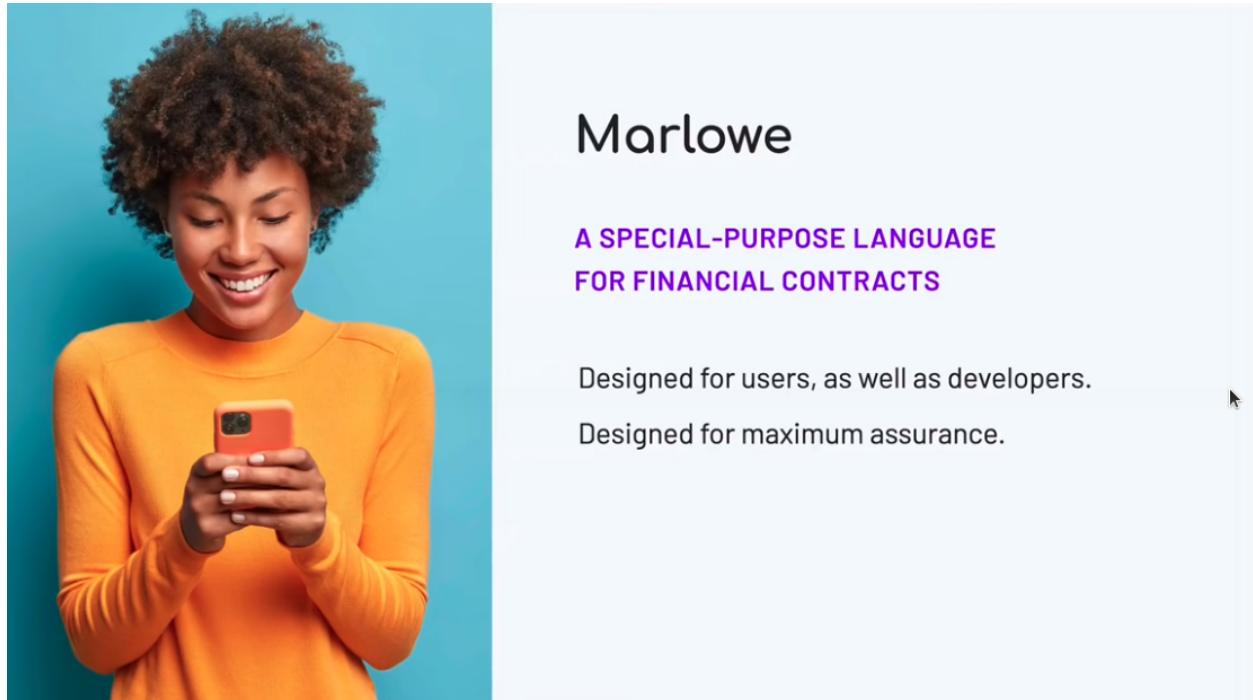
Quite a lot of the implementation of the tools from Marlowe is in the Plutus repository because it has that repository as a dependency.

If you look in the [IOHK online research library](#) and search for Marlowe you'll find a number of research papers we've written about how the system works.

You'll also find an online tutorial in the Marlowe Playground.

Finally, Alex is going to give some more information in his presentation coming up next.

### 9.2.12 Summary



Just to summarize, what we have in Marlowe is a DSL, a special-purpose language for financial contracts, running on top of Plutus. Because it's a DSL it allows us to give assurance that is harder to give for a general purpose language. And we get assurance of the way contracts should and shouldn't behave.

It also allows us to orient its design around users as well as developers. The language is simple and therefore we get readability.

We also get simulatability and we get these stronger assurances of static analysis and verification.

## 9.3 Lecture by Alex Nemish

Alex Nemish is one of the Marlowe developers and in this presentation, he shows us a bit of Marlowe semantics and Marlowe PAB (Plutus Application Backend) contracts.

We'll start with a brief description of Marlowe Semantics that's implemented in the [Semantics.hs](#) file. Then we'll look at the PAB contracts.

Here are the main data types for Marlowe.

It's a contract. Essentially those are six constructors that you can start to model a contract with and here's the state that is going to be stored on a blockchain.

So we have a state of balances of accounts by party, we have a map of choices, we have bound values which come from the *Let* constructor, and a *minSlot* which is the first slot that the contract sees.

The *Input* data type essentially contains actions for a Marlowe contract. It is either a deposit, a choice, or a notification.

```

Client.hs  Semantics.hs X
marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics > Contract > Close
218 {- Plutus doesn't support mutually recursive data types yet.
219 datatype Case is mutually recursive with @Contract@
220 -}
221 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 2 authors (You and others)
222 data Case a = Case Action a
223 deriving stock (Haskell.Show,Generic,Haskell.Eq,Haskell.Ord)
224 deriving anyclass (Pretty)
225
226 {-| Marlowe has six ways of building contracts.
227 Five of these - 'Pay', 'Let', 'If', 'When' and 'Assert' -
228 build a complex contract from simpler contracts, and the sixth, 'Close',
229 is a simple contract.
230 At each step of execution, as well as returning a new state and continuation contract,
231 it is possible that effects - payments - and warnings can be generated too.
232 -}
233 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 4 authors (You and others)
234 data Contract = Close You, 15/10/19, 15:13.+03:00 via PR #1588 • Marlowe: rename Refund to Close (#1588)
235 | Pay AccountId Payee Token (Value Observation) Contract
236 | If Observation Contract Contract
237 | When [Case Contract] Timeout Contract
238 | Let ValueId (Value Observation) Contract
239 | Assert Observation Contract
240 deriving stock (Haskell.Show,Generic,Haskell.Eq,Haskell.Ord)
241 deriving anyclass (Pretty)
242
243 {-| Marlowe contract internal state. Stored in a /Datum/ of a transaction output.
244 -}
245 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 3 authors (Pablo Lamela and others)
246 data State = State { accounts :: Accounts
247                    , choices :: Map ChoiceId ChosenNum
248                    , boundValues :: Map ValueId Integer
249                    , minSlot :: Slot }
250 deriving stock (Haskell.Show,Generic)

```

```

Client.hs  Semantics.hs X
marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics > State > State > accounts
227 build a complex contract from simpler contracts, and the sixth, 'Close',
228 is a simple contract.
229 At each step of execution, as well as returning a new state and continuation contract,
230 it is possible that effects - payments - and warnings can be generated too.
231 -}
232 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 4 authors (You and others)
233 data Contract = Close
234 | Pay AccountId Payee Token (Value Observation) Contract
235 | If Observation Contract Contract
236 | When [Case Contract] Timeout Contract
237 | Let ValueId (Value Observation) Contract
238 | Assert Observation Contract
239 deriving stock (Haskell.Show,Generic,Haskell.Eq,Haskell.Ord)
240 deriving anyclass (Pretty)
241
242 {-| Marlowe contract internal state. Stored in a /Datum/ of a transaction output.
243 -}
244 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 3 authors (Pablo Lamela and others)
245 data State = State { accounts :: Accounts You, 14/01/20, 0:03.+02:00 via PR #1603 • Multicurrency support for Marlowe (#1603)
246                    , choices :: Map ChoiceId ChosenNum
247                    , boundValues :: Map ValueId Integer
248                    , minSlot :: Slot }
249 deriving stock (Haskell.Show,Generic)
250
251 {-| Execution environment. Contains a slot interval of a transaction.
252 -}
253 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 2 authors (Evgenii Akentev and others)
254 newtype Environment = Environment { slotInterval :: SlotInterval }
255 deriving stock (Haskell.Show,Haskell.Eq,Haskell.Ord)

```

```

marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics > State > State
227 Five of these - 'Pay', 'Let', 'If', 'When' and 'Assert' -
228 build a complex contract from simpler contracts, and the sixth, 'Close',
229 is a simple contract.
230 At each step of execution, as well as returning a new state and continuation contract,
231 it is possible that effects - payments - and warnings can be generated too.
232 -}

Evgenii Akentev, 24/05/21, 14:15.+03:00 | 4 authors (You and others)
233 data Contract = Close
234 | Pay AccountId Payee Token (Value Observation) Contract
235 | If Observation Contract Contract
236 | When [Case Contract] Timeout Contract
237 | Let ValueId (Value Observation) Contract
238 | Assert Observation Contract
239 deriving stock (Haskell.Show,Generic,Haskell.Eq,Haskell.Ord)
240 deriving anyclass (Pretty)
241
242
243 {-| Marlowe contract internal state. Stored in a /Datum/ of a transaction output.
244 -}

Evgenii Akentev, 24/05/21, 14:15.+03:00 | 3 authors (Pablo Lamela and others)
245 data State = State { accounts :: Accounts
246                      , choices :: Map ChoiceId ChosenNum
247                      , boundValues :: Map ValueId Integer
248                      , minSlot :: Slot }
249 deriving stock (Haskell.Show,Generic)
250
251 {-| Execution environment. Contains a slot interval of a transaction.
252 -}

Evgenii Akentev, 24/05/21, 14:15.+03:00 | 2 authors (Evgenii Akentev and others)
253 newtype Environment = Environment { slotInterval :: SlotInterval }
254 deriving stock (Haskell.Show,Haskell.Eq,Haskell.Ord)
255

```

```

marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics > Input > INotify
242
243 {-| Marlowe contract internal state. Stored in a /Datum/ of a transaction output.
244 -}

Evgenii Akentev, 24/05/21, 14:15.+03:00 | 3 authors (Pablo Lamela and others)
245 data State = State { accounts :: Accounts
246                      , choices :: Map ChoiceId ChosenNum
247                      , boundValues :: Map ValueId Integer
248                      , minSlot :: Slot }
249 deriving stock (Haskell.Show,Generic)
250
251 {-| Execution environment. Contains a slot interval of a transaction.
252 -}

Evgenii Akentev, 24/05/21, 14:15.+03:00 | 2 authors (Evgenii Akentev and others)
253 newtype Environment = Environment { slotInterval :: SlotInterval }
254 deriving stock (Haskell.Show,Haskell.Eq,Haskell.Ord)
255
256
257 {-| Input for a Marlowe contract. Correspond to expected 'Action's.
258 -}

Evgenii Akentev, 24/05/21, 14:15.+03:00 | 3 authors (Pablo Lamela and others)
259 data Input = IDeposit AccountId Party Token Integer
260 | IChoice ChoiceId ChosenNum
261 | INotify
262 deriving stock (Haskell.Show,Haskell.Eq,Generic)
263 deriving anyclass (Pretty)
264
265
Evgenii Akentev, 24/05/21, 14:15.+03:00 | 2 authors (Pablo Lamela and others)
265 instance FromJSON Input where
266   parseJSON (String "input_notify") = return INotify
267   parseJSON (Object v) =
268     (IDeposit <$> (v .: "into_account")
269      <$> (v .: "input_from_party")

```

Here is the *TransactionInput* datatype. This is what we give as an input. Every transaction has a defined slot interval and a list of inputs.

```

393 prettyFragment tInp = text "TransactionInput" < space < lbrace < line < txIntl > TransactionInput
394
395     where
396       txIntlLine = hang 2 $ text "txInterval = " < prettyFragment (txInterval tInp) < comma
397       txInLine   = hang 2 $ text "txInputs = " < prettyFragment (txInputs tInp) < rbrace
398
399 {-| Marlowe transaction output.
400 -|}
401 data TransactionOutput =
402   TransactionOutput
403     { txOutWarnings :: [TransactionWarning]
404     , txOutPayments  :: [Payment]
405     , txOutState     :: State
406     , txOutContract  :: Contract }
407   deriving stock (Haskell.Show)
408   deriving anyclass (ToJSON, FromJSON)
409
410 {-|
411   This data type is a content of a contract's /Datum/.
412 -|}
413
414 data MarloweData = MarloweData {
415   marloweState :: State,
416   marloweContract :: Contract
417 } deriving stock (Haskell.Show, Generic)
418 deriving anyclass (ToJSON, FromJSON)
419
420 data MarloweParams = MarloweParams {

```

And we have *TransactionOutput* which contains the payments that we expect to happen, the output state and the output contract.

We also see *MarloweData* which is essentially what is going to be stored on the blockchain. It's the current state of a contract as well as the actual contract.

The entrance to the semantics is the *computeTransaction* function. It gets the transaction input, the current state and the current contract and returns the transaction output.

First of all we check the slot interval for errors. For example, we do not allow the slot interval to contain any timeouts. If you have a contract with a *When* construct of 10, you cannot produce a contract with a slot interval of 5..15 because it will contain a timeout.

Then we apply all inputs and if this is successful we return the transaction output with any warnings we have found, the payments we expect, the new state and the continuation contract.

So what happens in *applyAllInputs*?

First of all, it's a loop. It uses the *reduceContractUntilQuiescent* function which reduces the contract until it reaches a quiescent state. Once we reach a quiescent state, we take the first input and try to apply it, and then continue with the loop, until we get an empty input list. Then we return the current state and the continuation contract.

The *reduceContractUntilQuiescent* function goes through a loop and tries to apply *reduceContractStep* which essentially evaluates a contract.

If we get a *Close* then we are in a quiescent state. If we get a payment, then we evaluate it, update the balances and then return the reduced contract.

We do the same for *If*, *Let* and *Assert*. But for *When*, we only evaluate it if it's timed out, otherwise we say that it's not reduced, and that the contract is quiescent.

In a nutshell, Marlowe contract evaluation consists of two steps.

- We reduce the contract until it is quiescent - it's either closed or we get to a *When* that's not timed out yet.

```

Client.hs Semantics.hs X
marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics
715         warnings
716         ++ convertReduceWarnings reduceWarns
717         ++ convertApplyWarning applyWarn)
718         (payments ++ pays)
719         ApplyNoMatchError → ApplyAllNoMatchError
720     in applyAllLoop env state contract inputs [] []
721 where
722     convertApplyWarning :: ApplyWarning → [TransactionWarning]
723     convertApplyWarning warn =
724     case warn of
725     ApplyNoWarning → []
726     ApplyNonPositiveDeposit party accId tok amount →
727     [TransactionNonPositiveDeposit party accId tok amount]
728
729
730 | Try to compute outputs of a transaction given its inputs, a contract, and it's @State@
731 computeTransaction :: TransactionInput → State → Contract → TransactionOutput
732 computeTransaction tx state contract = let
733     inputs = txInputs tx
734     in case fixInterval (txInterval tx) state of
735     IntervalTrimmed env fixState → case applyAllInputs env fixState contract inputs of
736     ApplyAllSuccess warnings payments newState cont →
737     if (contract == cont) && ((contract /= Close) || (Map.null $ accounts state))
738     then Error TEUselessTransaction
739     else TransactionOutput { txOutWarnings = warnings
740     , txOutPayments = payments
741     , txOutState = newState
742     , txOutContract = cont }
743     ApplyAllNoMatchError → Error TApplyNoMatchError
744     ApplyAllAmbiguousSlotIntervalError → Error TEAmbiguousSlotIntervalError
745     IntervalError error → Error (TEIntervalError error)
746

```

```

Client.hs Semantics.hs X
marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics
715         warnings
716         ++ convertReduceWarnings reduceWarns
717         ++ convertApplyWarning applyWarn)
718         (payments ++ pays)
719         ApplyNoMatchError → ApplyAllNoMatchError
720     in applyAllLoop env state contract inputs [] []
721 where
722     convertApplyWarning :: ApplyWarning → [TransactionWarning]
723     convertApplyWarning warn =
724     case warn of
725     ApplyNoWarning → []
726     ApplyNonPositiveDeposit party accId tok amount →
727     [TransactionNonPositiveDeposit party accId tok amount]
728
729
730 | Try to compute outputs of a transaction given its inputs, a contract, and it's @State@
731 computeTransaction :: TransactionInput → State → Contract → TransactionOutput
732 computeTransaction tx state contract = let
733     inputs = txInputs tx
734     in case fixInterval (txInterval tx) state of
735     IntervalTrimmed env fixState → case applyAllInputs env fixState contract inputs of
736     ApplyAllSuccess warnings payments newState cont →
737     if (contract == cont) && ((contract /= Close) || (Map.null $ accounts state))
738     then Error TEUselessTransaction
739     else TransactionOutput { txOutWarnings = warnings
740     , txOutPayments = payments
741     , txOutState = newState
742     , txOutContract = cont }
743     ApplyAllNoMatchError → Error TApplyNoMatchError
744     ApplyAllAmbiguousSlotIntervalError → Error TEAmbiguousSlotIntervalError
745     IntervalError error → Error (TEIntervalError error)
746

```



```

marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics > applyAllInputs
688 -- | Apply a list of inputs to the contract
689 applyAllInputs :: Environment -> State -> Contract -> [Input] -> ApplyAllResult
690 applyAllInputs env state contract inputs = let
691     applyAllLoop
692     :: Environment
693     -> State
694     -> Contract
695     -> [Input]
696     -> [TransactionWarning]
697     -> [Payment]
698     -> ApplyAllResult
699     applyAllLoop env state contract inputs warnings payments =
700     case reduceContractUntilQuiescent env state contract of
701         RRAmbiguousSlotIntervalError -> ApplyAllAmbiguousSlotIntervalError
702         ContractQuiescent reduceWarns pays curState cont -> case inputs of
703             [] -> ApplyAllSuccess
704                 (warnings ++ convertReduceWarnings reduceWarns)
705                 (payments ++ pays)
706                 curState
707                 cont
708             (input : rest) -> case applyInput env curState input cont of
709                 Applied applyWarn newState cont ->
710                     applyAllLoop
711                         env
712                         newState
713                         cont
714                         rest
715                         (warnings
716                             ++ convertReduceWarnings reduceWarns
717                             ++ convertApplyWarning applyWarn)
718                         (payments ++ pays)
719                 ApplyNoMatchError -> ApplyAllNoMatchError

```

```

marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics > reduceContractUntilQuiescent
614 Assert obs cont -> let
615     warning = if evalObservation env state obs
616               then ReduceNoWarning
617               else ReduceAssertionFailed
618     in Reduced warning ReduceNoPayment state cont
619
620 -- | Reduce a contract until it cannot be reduced more
621 reduceContractUntilQuiescent :: Environment -> State -> Contract -> ReduceResult
622 reduceContractUntilQuiescent env state contract = let
623     reductionLoop
624     :: Environment -> State -> Contract -> [ReduceWarning] -> [Payment] -> ReduceResult
625     reductionLoop env state contract warnings payments =
626     case reduceContractStep env state contract of
627         Reduced warning effect newState cont -> let
628             newWarnings = if warning == ReduceNoWarning then warnings
629                           else warning : warnings
630             newPayments = case effect of
631                 ReduceWithPayment payment -> payment : payments
632                 ReduceNoPayment           -> payments
633             in reductionLoop env newState cont newWarnings newPayments
634         AmbiguousSlotIntervalReductionError -> RRAmbiguousSlotIntervalError
635         -- this is the last invocation of reductionLoop, so we can reverse lists
636         NotReduced -> ContractQuiescent (reverse warnings) (reverse payments) state contract
637     in reductionLoop env state contract [] []
638
639
640 -- | Apply a single Input to the contract (assumes the contract is reduced)
641 applyCases :: Environment -> State -> Input -> [Case Contract] -> ApplyResult
642 applyCases env state input cases = case (input, cases) of
643     (IDeposit accId1 party1 tok1 amount,
644      Case (Deposit accId2 party2 tok2 val) cont : rest) ->

```

```

marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics
559 newAccs = addMoneyToAccount accId (Token cur tok) amount accounts
560 in (ReduceNoPayment, newAccs)
561
562
563 | Carry a step of the contract with no inputs
564 reduceContractStep :: Environment -> State -> Contract -> ReduceStepResult
565 reduceContractStep env state contract = case contract of
566
567   Close -> case findOne (accounts state) of
568     Just ((party, money), newAccounts) -> let
569       newState = state { accounts = newAccounts }
570       in Reduced ReduceNoWarning (ReduceWithPayment (Payment party money)) newState Close
571     Nothing -> NotReduced
572
573   Pay accId payee tok val cont -> let
574     amountToPay = evalValue env state val
575     in if amountToPay <= 0
576       then let
577         warning = ReduceNonPositivePay accId payee tok amountToPay
578         in Reduced warning ReduceNoPayment state cont
579       else let
580         balance = moneyInAccount accId tok (accounts state)
581         paidAmount = min balance amountToPay
582         newBalance = balance - paidAmount
583         newAccs = updateMoneyInAccount accId tok newBalance (accounts state)
584         warning = if paidAmount < amountToPay
585                   then ReducePartialPay accId payee tok paidAmount amountToPay
586                   else ReduceNoWarning
587         (payment, finalAccs) = giveMoney payee tok paidAmount newAccs
588         newState = state { accounts = finalAccs }
589         in Reduced warning payment newState cont

```

- We try to apply inputs and evaluate the contract further.

Let's see how it works from the client side.

As you may have noticed, the Marlowe semantics code is quite abstract and it doesn't depend on the Cardano system's actions. So let's take a look at the actual Marlowe validator that's being executed on-chain.

Here's the *scriptInstance* which calls the *mkMarloweValidator* code, which in turn calls *mkValidator*, which uses a state machine library function *mkStateMachine* to which is provides two functions - a transition function and a finality check.

The finality check is very simple - we just check the the contract contract is constructed with *Close*.

The transition function is the meat of the validator.

It takes *MarloweParams* - which we'll talk about later, it takes the state of the state machine *MarloweData*, it takes *MarloweInput* which is essentially transaction input expressed in Cardano types. It will then return either *Nothing* in the case of error, or a new state along with zero or more constraints.

We check that the balances are valid - we require balances to be positive.

Then we create input constraints based on the inputs. So, in the case of deposits we expect that money will go into a contract. In the case of choices, we expect witnesses of the respective parties. We calculate that the contract contains the correct balance.

We construct a *TransactionInput* given the slot interval and list of inputs, and we call the *computeTransaction* function that we saw in *semantics.hs*.

With the computed result we construct a *MarloweData* with a new contract continuation and updated state. We produce output constraints that produce payouts to the respective parties, and we calculate the new balance. Then we combine all the constraints with range validation.

To validate inputs, we check that the required signatures and role tokens are present.

Payments to parties go either to a public key, or go to the validator *rolePayoutValidatorHash*, which simply checks, given a currency, that a transaction spends a role token.



```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client
586 mkValidator :: MarloweParams → Scripts.ValidatorType MarloweStateMachine
587 mkValidator p = SM.mkValidator $ SM.mkStateMachine Nothing (mkMarloweStateMachineTransition p) isFinal
588
589
590 mkMarloweValidatorCode
591   :: MarloweParams
592   → PlutusTx.CompiledCode (Scripts.ValidatorType MarloweStateMachine)
593 mkMarloweValidatorCode params =
594   $(PlutusTx.compile [|| mkValidator ||]) `PlutusTx.applyCode` PlutusTx.liftCode params
595
596
597 type MarloweStateMachine = StateMachine MarloweData MarloweInput
598
599 scriptInstance :: MarloweParams → Scripts.ScriptInstance MarloweStateMachine
600 scriptInstance params = Scripts.validator @MarloweStateMachine
601   (mkMarloweValidatorCode params)
602   $(PlutusTx.compile [|| wrap ||])
603   where
604     wrap = Scripts.wrapValidator @MarloweData @MarloweInput
605
606
607 mkMachineInstance :: MarloweParams → SM.StateMachineInstance MarloweData MarloweInput
608 mkMachineInstance params =
609   SM.StateMachineInstance
610     (SM.mkStateMachine Nothing (mkMarloweStateMachineTransition params) isFinal)
611     (scriptInstance params)
612
613
614 mkMarloweClient :: MarloweParams → SM.StateMachineClient MarloweData MarloweInput
615 mkMarloweClient params = SM.mkStateMachineClient (mkMachineInstance params)
616
617

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > mkMarloweStateMachineTransition
481
482 defaultMarloweParams :: MarloweParams
483 defaultMarloweParams = marloweParams adaSymbol
484
485
486 {-# INLINABLE mkMarloweStateMachineTransition #-}
487 mkMarloweStateMachineTransition
488   :: MarloweParams
489   → SM.State MarloweData
490   → MarloweInput
491   → Maybe (TxConstraints Void Void, SM.State MarloweData)
492 mkMarloweStateMachineTransition params SM.State{ SM.stateData=MarloweData{..}, SM.stateValue=scriptInValue}
493   (interval@(minSlot, maxSlot), inputs) = do
494   let positiveBalances = validateBalances marloweState ||
495     P.traceError "Invalid contract state. There exists an account with non positive balance"
496
497   {- We do not check that a transaction contains exact input payments.
498      We only require an evidence from a party, e.g. a signature for PubKey party,
499      or a spend of a 'party role' token.
500      This gives huge flexibility by allowing parties to provide multiple
501      inputs (either other contracts or P2PKH).
502      Then, we check scriptOutput to be correct.
503   -}
504   let inputsConstraints = validateInputs params inputs
505
506   -- total balance of all accounts in State
507   -- accounts must be positive, and we checked it above
508   let inputBalance = totalBalance (accounts marloweState)
509
510   -- ensure that a contract TxOut has what it suppose to have
511   let balancesOk = inputBalance == scriptInValue
512

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > mkMarloweStateMachineTransition
506 -- total balance of all accounts in State
507 -- accounts must be positive, and we checked it above
508 let inputBalance = totalBalance (accounts marloweState)
509
510 -- ensure that a contract TxOut has what it suppose to have
511 let balancesOk = inputBalance == scriptInValue
512
513 let preconditionsOk = P.traceIfFalse "Preconditions are false" $ positiveBalances && balancesOk
514
515 let txInput = TransactionInput {
516   txInterval = interval,
517   txInputs = inputs }
518
519 let computedResult = computeTransaction txInput marloweState marloweContract
520 case computedResult of
521   TransactionOutput {txOutPayments, txOutState, txOutContract} -> do
522
523     let marloweData = MarloweData {
524       marloweContract = txOutContract,
525       marloweState = txOutState }
526
527     let (outputsConstraints, finalBalance) = case txOutContract of
528       Close -> (payoutConstraints txOutPayments, P.zero)
529     -> let
530       outputsConstraints = payoutConstraints txOutPayments
531       totalIncome = P.foldMap collectDeposits inputs
532       totalPayouts = P.foldMap (\(Payment _ v) -> v) txOutPayments
533       finalBalance = totalIncome P-- totalPayouts
534       in (outputsConstraints, finalBalance)
535     let range = Interval.interval minSlot maxSlot
536     let constraints = inputsConstraints <> outputsConstraints <> mustValidateIn range
537     if preconditionsOk

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > mkMarloweStateMachineTransition
519 let computedResult = computeTransaction txInput marloweState marloweContract
520 case computedResult of
521   TransactionOutput {txOutPayments, txOutState, txOutContract} -> do
522
523     let marloweData = MarloweData {
524       marloweContract = txOutContract,
525       marloweState = txOutState }
526
527     let (outputsConstraints, finalBalance) = case txOutContract of
528       Close -> (payoutConstraints txOutPayments, P.zero)
529     -> let
530       outputsConstraints = payoutConstraints txOutPayments
531       totalIncome = P.foldMap collectDeposits inputs
532       totalPayouts = P.foldMap (\(Payment _ v) -> v) txOutPayments
533       finalBalance = totalIncome P-- totalPayouts
534       in (outputsConstraints, finalBalance)
535     let range = Interval.interval minSlot maxSlot
536     let constraints = inputsConstraints <> outputsConstraints <> mustValidateIn range
537     if preconditionsOk
538     then Just (constraints, SM.State marloweData finalBalance)
539     else Nothing
540     Error _ -> Nothing
541
542 where
543   validateInputs :: MarloweParams -> [Input] -> TxConstraints Void Void
544   validateInputs MarloweParams{rolesCurrency} inputs = let
545     (keys, roles) = P.foldMap validateInputWitness inputs
546     mustSpendSetOfRoleTokens = P.foldMap mustSpendRoleToken (AssocMap.keys roles)
547     in P.foldMap mustBeSignedBy keys P.<> mustSpendSetOfRoleTokens
548   where
549     validateInputWitness :: Input -> ([PubKeyHash], AssocMap.Map TokenName ())
550     validateInputWitness input =

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > mkMarloweStateMachineTransition
537
538   if preconditionsSuk
539   then Just (constraints, SM.State marloweData finalBalance)
540   else Nothing
541   Error _ → Nothing
542
543   where
544   validateInputs :: MarloweParams → [Input] → TxConstraints Void Void
545   validateInputs MarloweParams{rolesCurrency} inputs = let
546     (keys, roles) = P.foldMap validateInputWitness inputs
547     mustSpendSetOfRoleTokens = P.foldMap mustSpendRoleToken (AssocMap.keys roles)
548     in P.foldMap mustBeSignedBy keys P.<> mustSpendSetOfRoleTokens
549
550   where
551   validateInputWitness :: Input → ([PubKeyHash], AssocMap.Map TokenName ())
552   validateInputWitness input =
553     case input of
554       IDeposit _ party _ _ → validatePartyWitness party
555       IChoice (ChoiceId _ party) _ → validatePartyWitness party
556       INotify _ → (P.empty, P.empty)
557
558   where
559   validatePartyWitness (PK pk) = ([pk], P.empty)
560   validatePartyWitness (Role role) = ([], AssocMap.singleton role ())
561
562   mustSpendRoleToken :: TokenName → TxConstraints Void Void
563   mustSpendRoleToken role = mustSpendAtLeast $ Val.singleton rolesCurrency role 1
564
565   collectDeposits :: Input → Val.Value
566   collectDeposits (IDeposit _ _ (Token cur tok) amount) = Val.singleton cur tok amount
567   collectDeposits _ = P.zero
568
569   payoutConstraints :: [Payment] → TxConstraints i0 o0
570   payoutConstraints payments = P.foldMap paymentToTxOut paymentsByParty
571   where
572     paymentsByParty = AssocMap.toList $ P.foldMap paymentByParty payments
573     paymentToTxOut (party, value) = case party of
574       PK pk → mustPayToPubKey pk value
575       Role role → let
576         dataValue = Datum $ PlutusTx.toData role
577         in mustPayToOtherScript (rolePayoutValidatorHash params) dataValue value
578     paymentByParty (Payment party money) = AssocMap.singleton party money
579
580 {-# INLINABLE isFinal #-}
581 isFinal :: MarloweData → Bool
582 isFinal MarloweData{marloweContract=c} = c P.== Close

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > mkMarloweStateMachineTransition
551
552   case input of
553     IDeposit _ party _ _ → validatePartyWitness party
554     IChoice (ChoiceId _ party) _ → validatePartyWitness party
555     INotify _ → (P.empty, P.empty)
556
557   where
558   validatePartyWitness (PK pk) = ([pk], P.empty)
559   validatePartyWitness (Role role) = ([], AssocMap.singleton role ())
560
561   mustSpendRoleToken :: TokenName → TxConstraints Void Void
562   mustSpendRoleToken role = mustSpendAtLeast $ Val.singleton rolesCurrency role 1
563
564   collectDeposits :: Input → Val.Value
565   collectDeposits (IDeposit _ _ (Token cur tok) amount) = Val.singleton cur tok amount
566   collectDeposits _ = P.zero
567
568   payoutConstraints :: [Payment] → TxConstraints i0 o0
569   payoutConstraints payments = P.foldMap paymentToTxOut paymentsByParty
570   where
571     paymentsByParty = AssocMap.toList $ P.foldMap paymentByParty payments
572     paymentToTxOut (party, value) = case party of
573       PK pk → mustPayToPubKey pk value
574       Role role → let
575         dataValue = Datum $ PlutusTx.toData role
576         in mustPayToOtherScript (rolePayoutValidatorHash params) dataValue value
577     paymentByParty (Payment party money) = AssocMap.singleton party money
578
579 {-# INLINABLE isFinal #-}
580 isFinal :: MarloweData → Bool
581 isFinal MarloweData{marloweContract=c} = c P.== Close

```

For off-chain execution, we provide three Marlowe PAB contracts.

- Marlowe Follower Contract
- Marlowe Control Contract
- Marlowe Companion Contract

### 9.3.1 Follower Contract

```

146 marloweFollowContract = do
147   params ← endpoint @"follow"
148   slot ← currentSlot
149   logDebug @String "Getting contract history"
150   follow 0 slot params
151   where
152     I
153     follow ifrom ito params = do
154       let client@StateMachineClient{scInstance} = mkMarloweClient params
155       let inst = validatorInstance scInstance
156       let address = Scripts.scriptAddress inst
157       AddressChangeResponse{acrTxns} ← addressChangeRequest
158         AddressChangeRequest
159         { acreqSlotRangeFrom = ifrom
160         , acreqSlotRangeTo = ito
161         , acreqAddress = address
162         }
163       let go [] = pure InProgress
164       go (tx:rest) = do
165         res ← updateHistoryFromTx client params tx
166         case res of
167           Finished → pure Finished
168           InProgress → go rest
169       res ← go acrTxns
170       case res of
171         Finished → do
172           logDebug @String ("Contract finished " < show params)
173           pure () -- close the contract
174         InProgress →
175           let next = succ ito in
176           follow next next params
177   updateHistoryFromTx StateMachineClient{scInstance, scChooser} params tx = do

```

This is a very simple one - it contains only one endpoint called *follow*. It subscribes to all changes to a Marlowe contract validator address, so that we can store all the inputs that are applied to a Marlowe contract.

It uses the *updateHistoryFromTx* function which, in a nutshell, finds a Marlowe input and constructs a *TransactionInput* data type, and uses *tell* to update the PAB contract state.

If you were connected to a web socket for this contract, you would be notified about transition changes.

The state of the contract is stored in *ContractHistory*, which stores an initial *MarloweParams*, an initial *MarloweData* and a list of all *TransactionInputs* that were applied to this contract. You can always restore the current state by applying a list of inputs to an initial state.

### 9.3.2 Control Contract

The *marlowePlutusContract* is a control contract. It allows you to create an instance of a Marlowe contract, apply inputs to the instance, to auto-execute the contract, if possible, to redeem tokens from payments to roles, and to close the contract.

Let's go through Marlowe contract creation.

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > marloweFollowContract
174   let next = succ ito in
175   follow next next params
176
177   updateHistoryFromTx StateMachineClient{scInstance, scChooser} params tx = do
178   let inst = validatorInstance scInstance
179   let address = Scripts.scriptAddress inst
180   let utxo = outputsMapFromTxForAddress address tx
181   let states = getStates scInstance utxo
182   case findInput inst tx of
183   -- if there's no TxIn for Marlowe contract that means
184   -- it's a contract creation transaction, and there is Marlowe TxOut
185   Nothing → case scChooser states of
186   Left err → throwing _SMContractError err
187   Right (state, _) → do
188     let initialMarloweData = tyTxOutData state
189     logDebug @String ("Contract created " ∘ show initialMarloweData)
190     tell $ Created params initialMarloweData
191     pure InProgress
192   -- There is TxIn with Marlowe contract, hence this is a state transition
193   Just (interval, inputs) → do
194     let txInput = TransactionInput {
195       txInterval = interval,
196       txInputs = inputs }
197     logDebug @String ("Transition " ∘ show txInput)
198     tell $ Transition txInput
199     case states of
200     -- when there is no Marlowe TxOut the contract is closed
201     -- and we can close the follower contract
202     [] → pure Finished
203     -- otherwise we continue following
204     _ → pure InProgress
205

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > ContractHistory > History
112
113 type RoleOwners = AssocMap.Map Val.TokenName PubKeyHash
114
115 type MarloweContractState = [MarloweData]
116
117 data ContractHistory
118   = None
119   | Created MarloweParams MarloweData
120   | Transition TransactionInput
121   | History MarloweParams MarloweData [TransactionInput]
122   deriving (Show, Generic)
123   deriving anyclass (FromJSON, ToJSON)
124
125 instance Semigroup ContractHistory where
126   any ∘ None = any
127   _ ∘ Created params md = History params md []
128   History params md inputs ∘ Transition input = History params md (inputs ∘ [input])
129   cur ∘ _ = cur
130
131 instance Monoid ContractHistory where
132   mempty = None
133
134 data ContractProgress = InProgress | Finished
135   deriving (Show, Eq)
136
137 instance Semigroup ContractProgress where
138   _ ∘ Finished = Finished

```

```

216
217 {- This is a control contract.
218    It allows to create a contract, apply inputs, auto-execute a contract,
219    redeem role payouts, and close.
220 -}
221 marlowePlutusContract :: Contract MarloweContractState MarloweSchema MarloweError ()
222 marlowePlutusContract = do
223   create `select` apply `select` auto `select` redeem `select` close
224   where
225     create = do
226       (owners, contract) ← endpoint @"create"
227       (params, distributeRoleTokens) ← setupMarloweParams owners contract
228       slot ← currentSlot
229       let StateMachineClient{scInstance} = mkMarloweClient params
230       let marloweData = MarloweData {
231         marloweContract = contract,
232         marloweState = emptyState slot }
233       let payValue = adaValueOf 0
234       let StateMachineInstance{validatorInstance} = scInstance
235       let tx = mustPayToTheScript marloweData payValue <> distributeRoleTokens
236       let lookups = Constraints.scriptInstanceLookups validatorInstance
237       utx ← either (throwing _ConstraintResolutionError) pure (Constraints.mkTx lookups tx)
238       submitTxConfirmed utx
239       marlowePlutusContract
240     apply = do
241       (params, slotInterval, inputs) ← endpoint @"apply-inputs"
242       _ ← applyInputs params slotInterval inputs
243       marlowePlutusContract
244     redeem = mapError (review _MarloweError) $ do
245       (MarloweParams{rolesCurrency}, role, pkh) ←
246         endpoint @"redeem"
247       let address = scriptHashAddress (mkRolePayoutValidatorHash rolesCurrency)

```

## Create Endpoint

When you call the *create* endpoint, you provide a contract and a map of roles to public keys. We then setup a *MarloweParams*.

*MarloweParams* is a way to parameterise a Marlowe contract. You can specify your own role payout validator by providing its hash. There is a default one that checks that the role token is spent within the transaction but you can do whatever you like.

When your contract uses roles, we need to know the currency symbol for the role. When the contract uses roles, we need to create role tokens and distribute them to their owners.

In the *setupMarloweParams* function we get the roles that are used within the contract. If we have owners for these roles, we create tokens with role names. By default we create one token per role. We use the *Contract.forgeContract* function to create the tokens and then assign them to the creator. Then, in the same transaction, we distribute the role tokens to their owners.

Next in the control contract, we use the state machine library to create a state machine client and submit the transaction.

## Apply Endpoint

The *apply* endpoint is very simple. We call the *applyInputs* function.

We construct a slot range and we use the *runStep* function which takes a slot range and a list of inputs.



```

Client.hs Semantics.hs X
marlowe > src > Language > Marlowe > Semantics.hs > Language.Marlowe.Semantics > MarloweParams
409
410
411 {-|
412   This data type is a content of a contract's /Datum/
413 -|}
414
415 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 3 authors (You and others)
416 data MarloweData = MarloweData {
417   marloweState :: State,
418   marloweContract :: Contract
419 } deriving stock (Haskell.Show, Generic)
420   deriving anyclass (ToJSON, FromJSON)
421
422 Evgenii Akentev, 24/05/21, 14:15.+03:00 | 2 authors (You and others)
423 data MarloweParams = MarloweParams {
424   rolePayoutValidatorHash :: ValidatorHash,
425   rolesCurrency :: CurrencySymbol
426 }
427   deriving stock (Haskell.Show, Generic, Haskell.Eq, Haskell.Ord)
428   deriving anyclass (FromJSON, ToJSON)
429
430 -- | Empty State for a given minimal 'Slot'
431 emptyState :: Slot -> State
432 emptyState sn = State
433   { accounts = Map.empty
434   , choices = Map.empty
435   , boundValues = Map.empty
436   , minSlot = sn }
437
438 -- | Check if a 'num' is within a list of inclusive bounds.

```

```

Client.hs Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > applyInputs
434
435 checkCase (Case (Choice (ChoiceId _ p) _) cont) | p /= party = check cont
436 checkCase _ = False
437
438 applyInputs :: AsMarloweError e
439   => MarloweParams
440   -> Maybe SlotInterval
441   -> [Input]
442   -> Contract MarloweContractState MarloweSchema e MarloweData
443 applyInputs params slotInterval inputs = mapError (review _MarloweError) $ do
444   slotRange <- case slotInterval of
445     Just si -> pure si
446     Nothing -> do
447       slot <- currentSlot
448       pure (slot, slot + defaultTxValidationRange)
449   let theClient = mkMarloweClient params
450   dat <- SM.runStep theClient (slotRange, inputs)
451   case dat of
452     SM.TransitionFailure e -> do
453       logError e
454       throwing _TransitionError e
455     SM.TransitionSuccess d -> return d
456
457 rolePayoutScript :: CurrencySymbol -> Validator
458 rolePayoutScript symbol = mkValidatorScript ($$(PlutusTx.compile [|| wrapped ||]) `PlutusTx.applyCode` PlutusTx.liftCode symbol)
459   where
460     wrapped s = Scripts.wrapValidator (rolePayoutValidator s)
461
462
463 {-# INLINABLE rolePayoutValidator #-}

```

## Redeem Endpoint

The *redeem* endpoint allows you to get money that has been paid to a role payout script.

```

243 marlowePlutusContract
244 redeem = mapError (review _MarloweError) $ do
245   (MarloweParams[rolesCurrency], role, pkh) ←
246     endpoint @"redeem"
247   let address = scriptHashAddress (mkRolePayoutValidatorHash rolesCurrency)
248   utxos ← utxoAt address
249   let spendPayoutConstraints tx ref TxOutTx{txOutTxOut} = let
250     expectedDatumHash = datumHash (Datum $ PlutusTx.toData role)
251     amount = txOutValue txOutTxOut
252     in case {txOutDatum txOutTxOut of
253       Just datumHash | datumHash == expectedDatumHash →
254         -- we spend the rolePayoutScript address
255         Constraints.mustSpendScriptOutput ref unitRedeemer
256         -- and pay to a token owner
257         ◇ Constraints.mustPayToPubKey pkh amount
258       _ → tx
259
260   let spendPayouts = Map.foldlWithKey spendPayoutConstraints mempty utxos
261   constraints = spendPayouts
262   -- must spend a role token for authorization
263   ◇ Constraints.mustSpendAtLeast (Val.singleton rolesCurrency role 1)
264   -- lookup for payout validator and role payouts
265   validator = rolePayoutScript rolesCurrency
266   lookups = Constraints.otherScript validator
267   ◇ Constraints.unspentOutputs utxos
268   ◇ Constraints.ownPubKeyHash pkh
269   tx ← either (throwing _ConstraintResolutionError) pure (Constraints.mkTx @Void lookups constraints)
270   _ ← submitUnbalancedTx tx
271   marlowePlutusContract
272   auto = do
273     (params, party, untilSlot) ← endpoint @"auto"
274     let theClient = mkMarloweClient params

```

We get the address of the script and then send all the outputs to the token owner.

## Auto Endpoint

The *auto* endpoint is quite interesting and quite complicated. There is a set of contracts that can be executed automatically.

Imagine a contract that contains only deposits and payouts. No participant needs to provide choices or any interactive stuff. There are only scheduled payments. Such a contract can be executed automatically, and the *auto* endpoint allows exactly that.

So, if a contract can be executed automatically, it calls *autoExecuteContract*.

This is a state machine that pays a deposit or waits for other parties to do their part.

### 9.3.3 Companion Contract

The last interesting contract is the Marlowe Companion Contract.

This is a contract that monitors a participant wallet and notifies when a role token arrives.

It listens to transactions that go to your own address and if there is a token and this token is generated by Marlowe contract creation, it tries to find the Marlowe contract and, if it succeeds, it updates the state of the contract. If you are subscribed to the contract's web socket, you will get a notification about a role token, and you'll get a map of *MarloweParams* to *MarloweData*.



```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > marlowePlutusContract
259
260 let spendPayouts = Map.foldlWithKey spendPayoutConstraints mempty utxos
261     constraints = spendPayouts
262     -- must spend a role token for authorization
263     ◇ Constraints.mustSpendAtLeast (Val.singleton rolesCurrency role 1)
264     -- lookup for payout validator and role payouts
265     validator = rolePayoutScript rolesCurrency
266     lookups = Constraints.otherScript validator
267     ◇ Constraints.unspentOutputs utxos
268     ◇ Constraints.ownPubKeyHash pkh
269 tx ← either (throwing _ConstraintResolutionError) pure (Constraints.mkTx @Void lookups constraints)
270 _ ← submitUnbalancedTx tx
271 marlowePlutusContract
272
273 auto = do
274   (params, party, untilSlot) ← endpoint @"auto"
275   let theClient = mkMarloweClient params
276   let continueWith :: MarloweData → Contract MarloweContractState MarloweSchema MarloweError ()
277       continueWith md@MarloweData{marloweContract} =
278         if canAutoExecuteContractForParty party md marloweContract
279         then autoExecuteContract theClient party md
280         else marlowePlutusContract
281
282 maybeState ← SM.getOnChainState theClient
283 case maybeState of
284   Nothing → do
285     wr ← SM.waitForUpdateUntil theClient untilSlot
286     case wr of
287       ContractEnded → do
288         logInfo @String $ "Contract Ended for party " ◇ show party
289         marlowePlutusContract
290       Timeout _ → do
291         logInfo @String $ "Contract Timeout for party " ◇ show party

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > marlowePlutusContract
297
298
299 autoExecuteContract :: StateMachineClient MarloweData MarloweInput
300   → Party
301   → MarloweData
302   → Contract MarloweContractState MarloweSchema MarloweError ()
303 autoExecuteContract theClient party marloweData = do
304   slot ← currentSlot
305   let slotRange = (slot, slot + defaultTxValidationRange)
306   let action = getAction slotRange party marloweData
307   case action of
308     PayDeposit acc p token amount → do
309     logInfo @String $ "PayDeposit " ◇ show amount ◇ " at within slots " ◇ show slotRange
310     let payDeposit = do
311       marloweData ← SM.runStep theClient (slotRange, [IDeposit acc p token amount])
312       case marloweData of
313         SM.TransitionFailure e → throwing _TransitionError e
314         SM.TransitionSuccess d → continueWith d
315
316     catching (_StateMachineError) payDeposit $ \err → do
317       logWarn @String $ "Error " ◇ show err
318       logInfo @String $ "Retry PayDeposit in 2 slots"
319       _ ← awaitSlot (slot + 2)
320       continueWith marloweData
321
322   WaitForTimeout timeout → do
323     logInfo @String $ "WaitForTimeout " ◇ show timeout
324     _ ← awaitSlot timeout
325     continueWith marloweData
326
327   WaitOtherActionUntil timeout → do
328     logInfo @String $ "WaitOtherActionUntil " ◇ show timeout
329     wr ← SM.waitForUpdateUntil theClient timeout
330     case wr of

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > marlowePlutusContract

630
631 {-
632   Contract that monitors a user wallet for receiving a Marlowe role token.
633   When it sees that a Marlowe contract exists on chain with a role currency
634   of a token the user owns it updates its @CompanionState@
635   with contract's @MarloweParams@ and @MarloweData@
636 -}
637 marloweCompanionContract :: Contract CompanionState MarloweCompanionSchema MarloweError ()
638 marloweCompanionContract = contracts
639   where
640     contracts = do
641       pkh ← pubKeyHash <-> ownPubKey
642       let ownAddress = pubKeyHashAddress pkh
643       utxo ← utxoAt ownAddress
644       let txOuts = fmap (txOutTxOut . snd) $ Map.toList utxo
645       forM_ txOuts notifyOnNewContractRoles
646       cont ownAddress
647     cont ownAddress = do
648       txns ← nextTransactionsAt ownAddress
649       let txOuts = txns >= eitherTx (const []) txOutputs
650       forM_ txOuts notifyOnNewContractRoles
651       cont ownAddress
652
653
654 notifyOnNewContractRoles :: TxOut
655   → Contract CompanionState MarloweCompanionSchema MarloweError ()
656 notifyOnNewContractRoles txout = do
657   let curSymbols = filterRoles txout
658   forM_ curSymbols $ \cs → do
659     logInfo @String $ "Processing currency symbol: " <-> show cs
660     contract ← findMarloweContractsOnChainByRoleCurrency cs
661     case contract of

```

```

Client.hs x Semantics.hs
marlowe > src > Language > Marlowe > Client.hs > Language.Marlowe.Client > notifyOnNewContractRoles

641
642   pkh ← pubKeyHash <-> ownPubKey
643   let ownAddress = pubKeyHashAddress pkh
644   utxo ← utxoAt ownAddress
645   let txOuts = fmap (txOutTxOut . snd) $ Map.toList utxo
646   forM_ txOuts notifyOnNewContractRoles
647   cont ownAddress
648   cont ownAddress = do
649     txns ← nextTransactionsAt ownAddress
650     let txOuts = txns >= eitherTx (const []) txOutputs
651     forM_ txOuts notifyOnNewContractRoles
652     cont ownAddress
653
654
655 notifyOnNewContractRoles :: TxOut
656   → Contract CompanionState MarloweCompanionSchema MarloweError ()
657 notifyOnNewContractRoles txout = do
658   let curSymbols = filterRoles txout
659   forM_ curSymbols $ \cs → do
660     logInfo @String $ "Processing currency symbol: " <-> show cs
661     contract ← findMarloweContractsOnChainByRoleCurrency cs
662     case contract of
663       Just (params, md) → tell $ CompanionState (Map.singleton params md)
664       Nothing           → pure ()
665
666
667 filterRoles :: TxOut → [CurrencySymbol]
668 filterRoles TxOut { txOutValue, txOutDatumHash = Nothing } =
669   let curSymbols = filter (≠ adaSymbol) $ AssocMap.keys $ Val.getValue txOutValue
670   in curSymbols
671 filterRoles _ = []

```

```

614 mkMarloweClient :: MarloweParams → SM.StateMachineClient MarloweData MarloweInput
615 mkMarloweClient params = SM.mkStateMachineClient (mkMachineInstance params)
616
617
618 defaultTxValidationRange :: Slot
619 defaultTxValidationRange = 10
620
621
622 newtype CompanionState = CompanionState (Map MarloweParams MarloweData)
623   deriving (Semigroup, Monoid) via (Map MarloweParams MarloweData)
624
625 instance ToJSON CompanionState where
626   toJSON (CompanionState m) = toJSON $ Map.toList m
627
628 instance FromJSON CompanionState where
629   parseJSON v = CompanionState . Map.fromList <$> parseJSON v
630
631 {-
632   Contract that monitors a user wallet for receiving a Marlowe role token.
633   When it sees that a Marlowe contract exists on chain with a role currency
634   of a token the user owns it updates its @CompanionState@
635   with contract's @MarloweParams@ and @MarloweData@
636 -}
637 marloweCompanionContract :: Contract CompanionState MarloweCompanionSchema MarloweError ()
638 marloweCompanionContract = contracts
639   where
640     contracts = do
641       pkh ← pubKeyHash <$> ownPubKey
642       let ownAddress = pubKeyHashAddress pkh

```

## 9.4 Playing in the Playground

Let's play around a little with Marlowe in the playground.

When you go to the playground, you first get presented with three options for you to choose in which language you want to write your Marlowe contracts.

You can do it in Haskell, you can do it in Javascript, or you can do it in Blockly or directly in Marlowe.

Blockly is very nice and you don't need any programming experience to do this.

### 9.4.1 Blockly

Let's start a new project and select *Blockly*.

This is a graphical editor. We can just click and drop a Marlowe contract together.

As an example let's write a contract where there are three parties - Alice, Bob and Charlie.

The idea is that Alice and Bob deposit an amount of Ada into the contract, let's say 10 Ada, and then Charlie decides whether Alice or Bob gets the total amount. Depending on Charlie's decision either Alice gets 20 or Bob gets 20.


There's always the possibility that one of the three doesn't play along; Alice doesn't make her deposit, Bob doesn't make his deposit, or Charlie doesn't make his choice. In this case everybody should just get reimbursed.

When we start with Blockly, there is a contract and it's just a *Close* contract, which in this case doesn't do anything. If there was money in internal accounts it would pay back the money to the owners of the accounts.

We want to do something else, so let's first wait for a deposit by Alice.

Because that's an external action that's triggered by one of the parties, in this case Alice, we need the *When* construct that Simon mentioned.

We can remove the *Close* contract, slide the *When* one into its place.

 MARLOWE PLAYGROUND

Tutorials   Actus Labs

New Project   Open   Open Example

### What is Marlowe?

Marlowe is special-purpose language for financial contracts on Cardano, allowing contracts to be written in the language of finance, rather than using a general-purpose language on the blockchain. Because it is special-purpose, it is easier to read, write and understand Marlowe contracts. It is also safer: some sorts of errors are impossible to write, and we can completely analyse contract behaviour without having to run a contract.

### What is the Marlowe Playground?


In the browser-based Marlowe Playground you can write Marlowe contracts, in a variety of different ways. Once a contract is written, you can analyse its behaviour, e.g. checking whether any payments made by the contract could conceivably fail. You can also step through how a contract will behave, simulating the actions of the participants, and read a comprehensive tutorial on Marlowe and the Playground.

### How does the playground work?

Marlowe contracts can be built in different ways. You can write them as Marlowe text, but also use the Blockly visual programming tool to create contracts by fitting together blocks that represent the different components. Marlowe is embedded in JavaScript and Haskell, and so you can use features from them to help you to build Marlowe contracts more readably and succinctly.

#### Option 1


start with Haskell



```
graph TD; Haskell --> Marlowe; Marlowe <--> Blockly; Marlowe --> Simulator; Blockly --> Simulator;
```

#### Option 2


start with JavaScript



```
graph TD; JavaScript --> Marlowe; Marlowe <--> Blockly; Marlowe --> Simulator; Blockly --> Simulator;
```

#### Option 3

start with Marlowe or Blockly




```
graph TD; Marlowe <--> Blockly; Marlowe --> Simulator; Blockly --> Simulator;
```

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter

 MARLOWE PLAYGROUND

New Project

Tutorials   Actus Labs

New Project   Open   Open Example   Rename   Save   Save As...

View as Marlowe   Send To Simulator

Contracts

Observations

Actions

Values

Payee

Party

Token

Bounds

CONTRACT

Close

Metadata

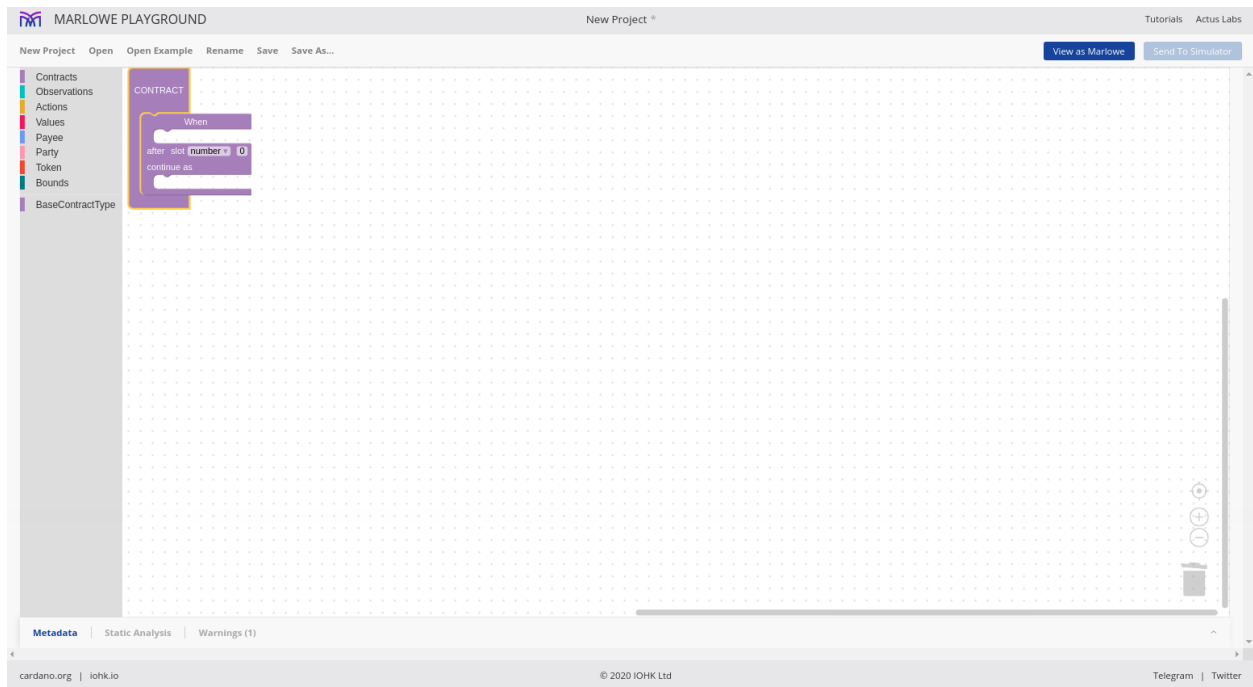
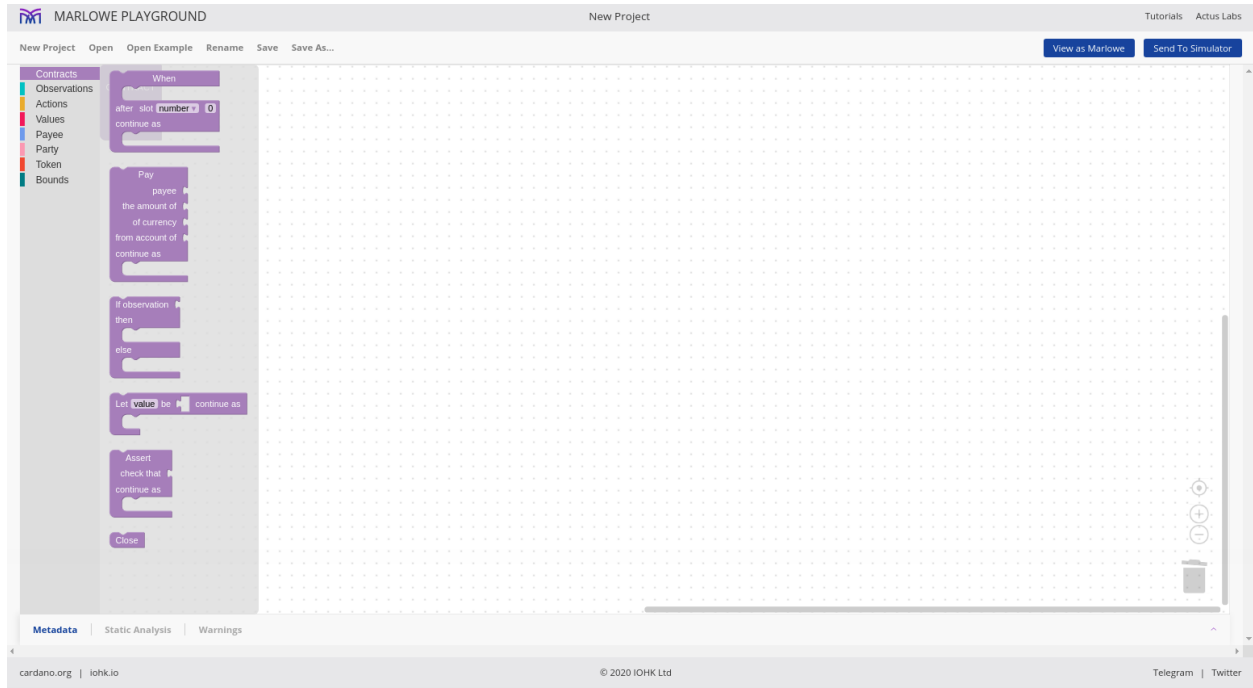
Static Analysis

Warnings

cardano.org | iohk.io

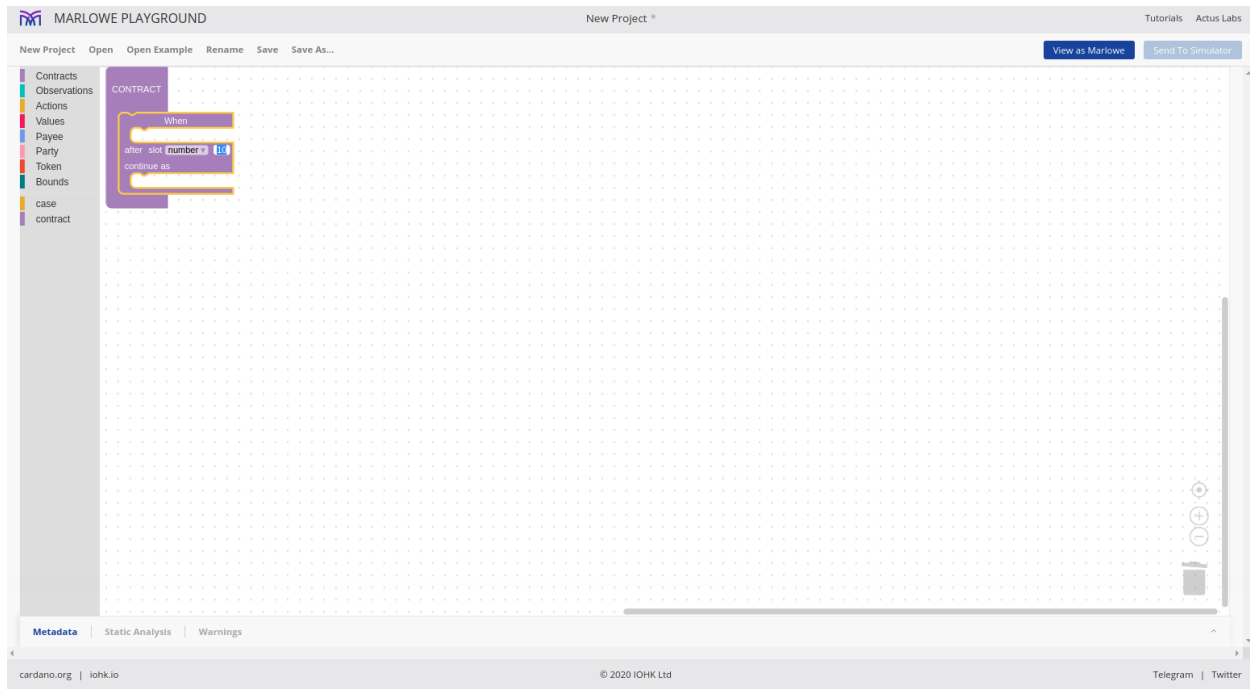
© 2020 IOHK Ltd

Telegram | Twitter



Here we see all the slots where other things need to go. We see some fields that we have to set.

We can set a timeout so let's say this deposit by Alice has to happen by slot 10.



If it doesn't happen, we can say what should happen afterwards, and there is not really a good choice to do anything except close in that case, so in that case nothing will happen.

Here we say what external actions we wait for. Let's say we only wait for one action, namely that Alice makes a deposit.

So we can check for actions and pick the deposit one and slide it in.

We see some slots that we have to fill. First of all, a party who has to make the deposit, and there are two choices - a public key or role.

Let's take role because then we can just say Alice. Normally this would be the name of the role token, so whoever owns that token can incorporate that role.

So Alice makes a deposit. Now the amount. That's a *Value* and let's say we just pick a constant amount of 10 Ada.

The amount is 10, and the fact that it is Ada must be specified in the currency slot.

There's also the option to use tokens than Ada, but let's stick with Ada.

Now there are these internal accounts that also belong to one of the parties, so let's say Alice pays it into her own internal account.

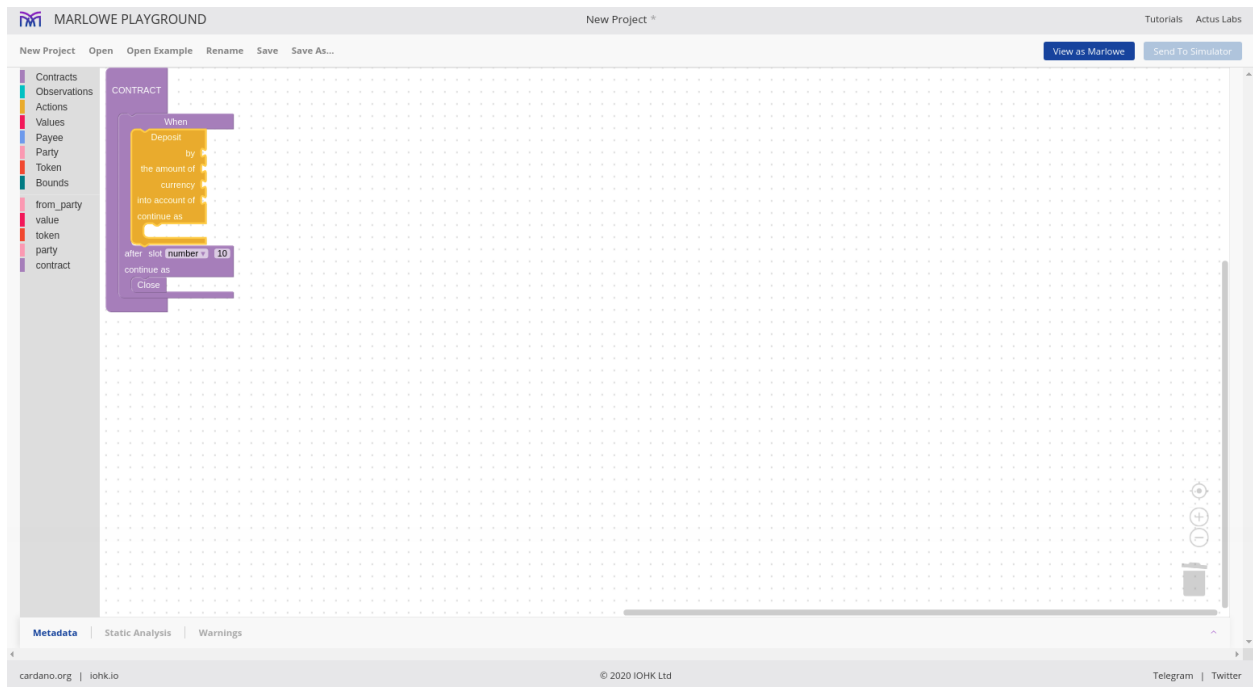
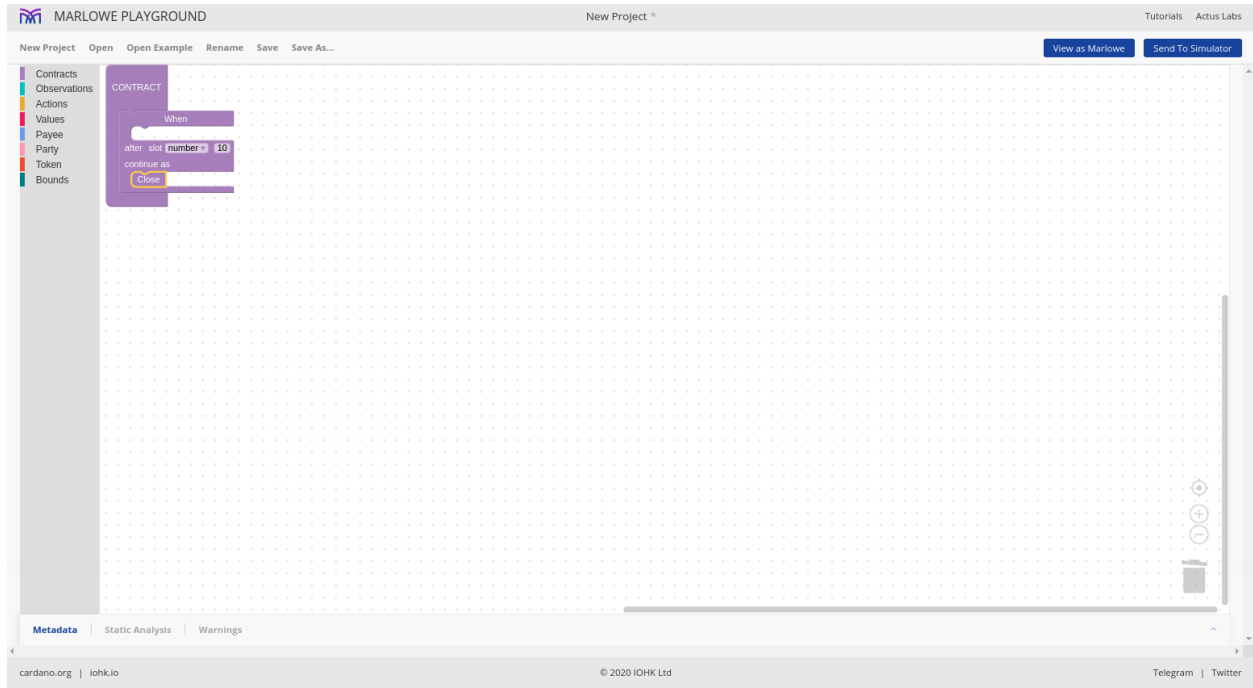
That can be copy/pasted rather than getting it from the *Party* menu again.

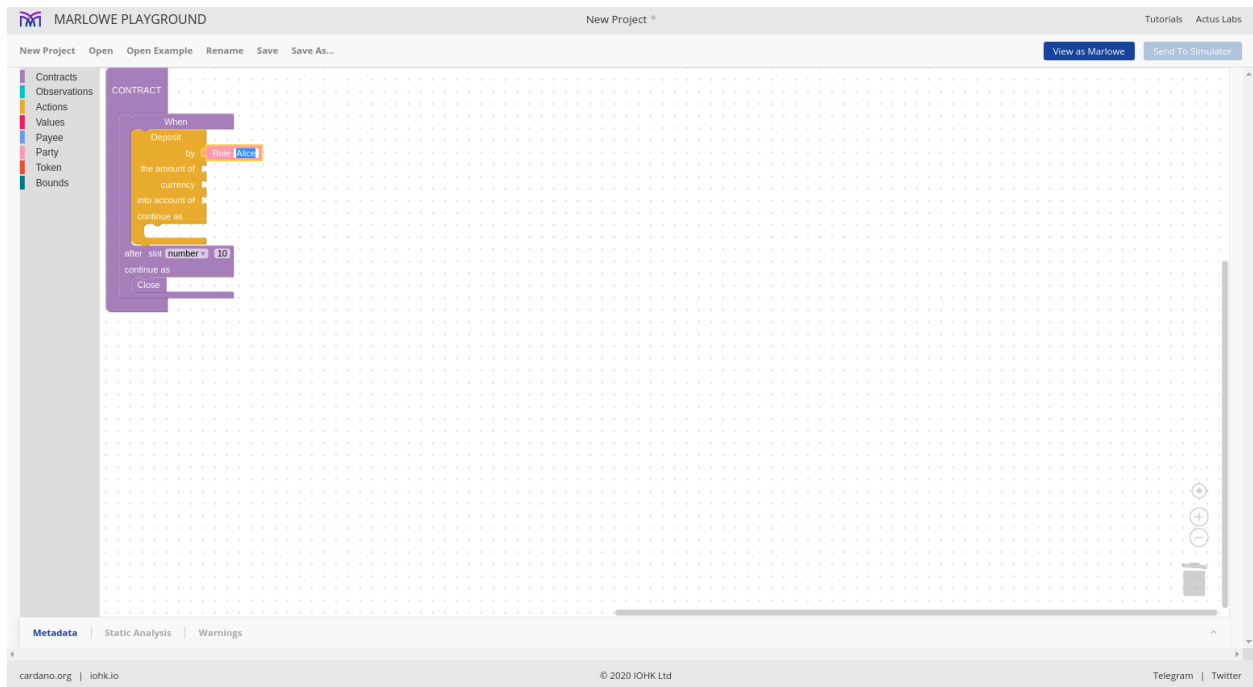
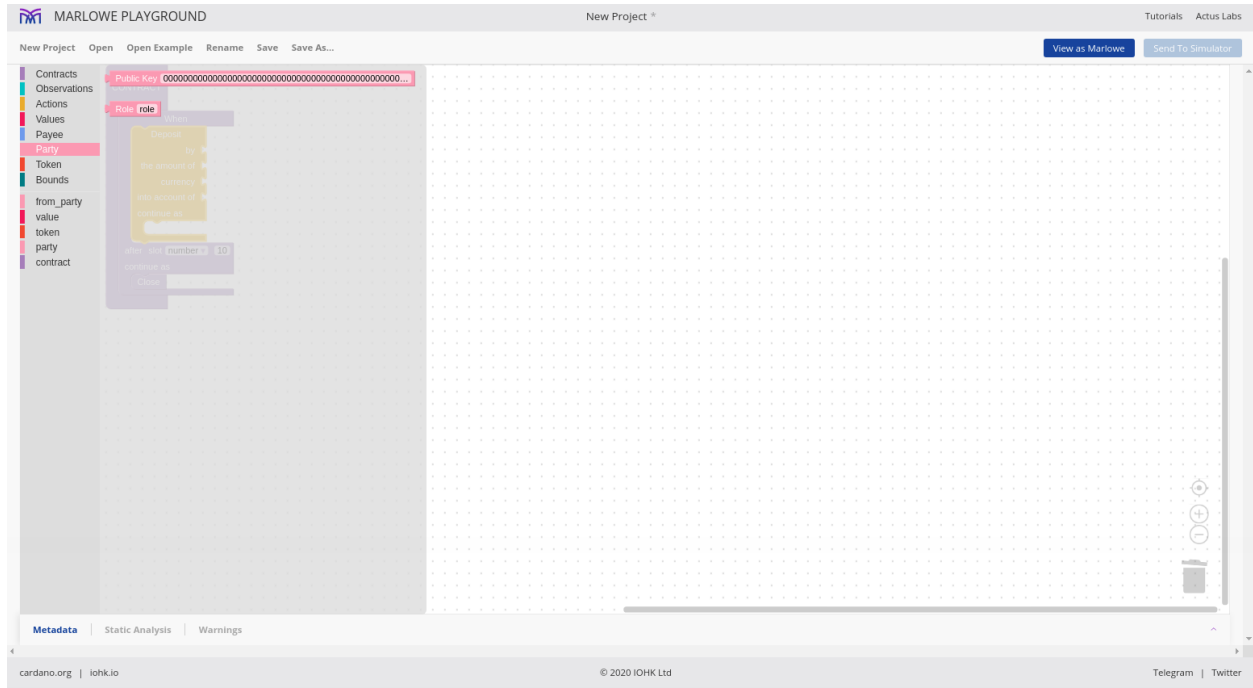
Now we must say what happens next, if Alice makes this deposit. Afterwards we want Bob to make a deposit, so we can start by just copying the whole *When* block.

First of all we change the timeout to 20 so as to give Bob also 10 slots to do something, and then wherever we have Alice, we now put Bob.

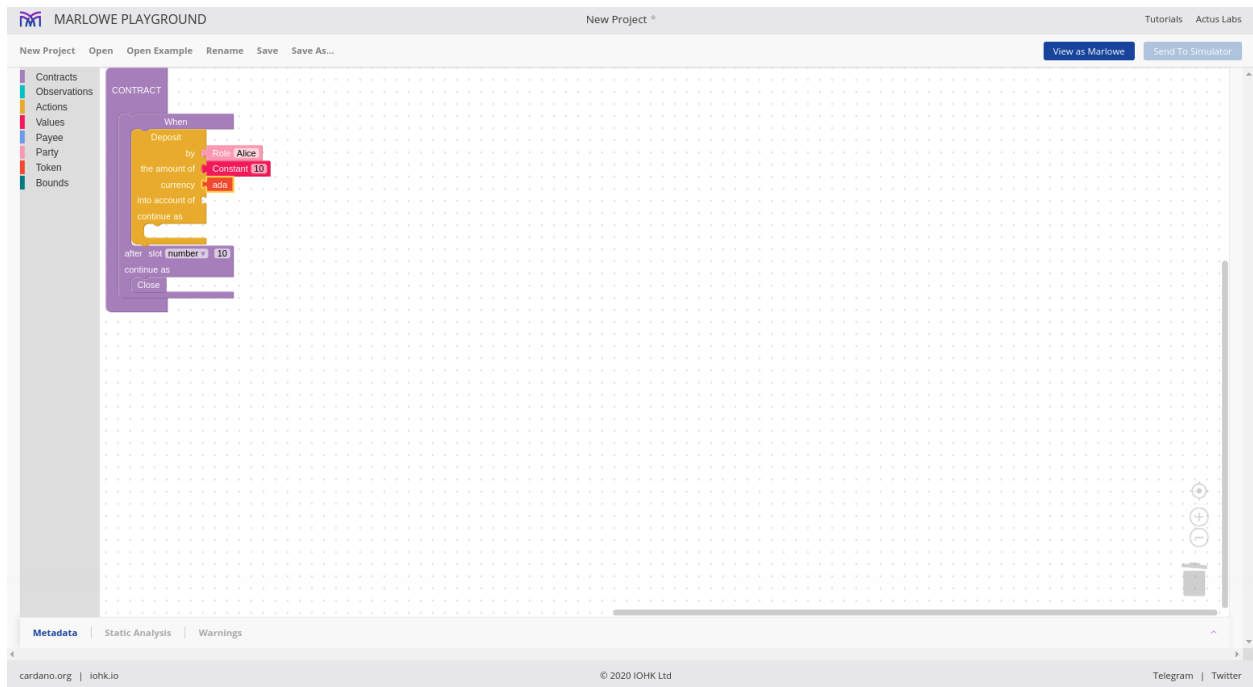
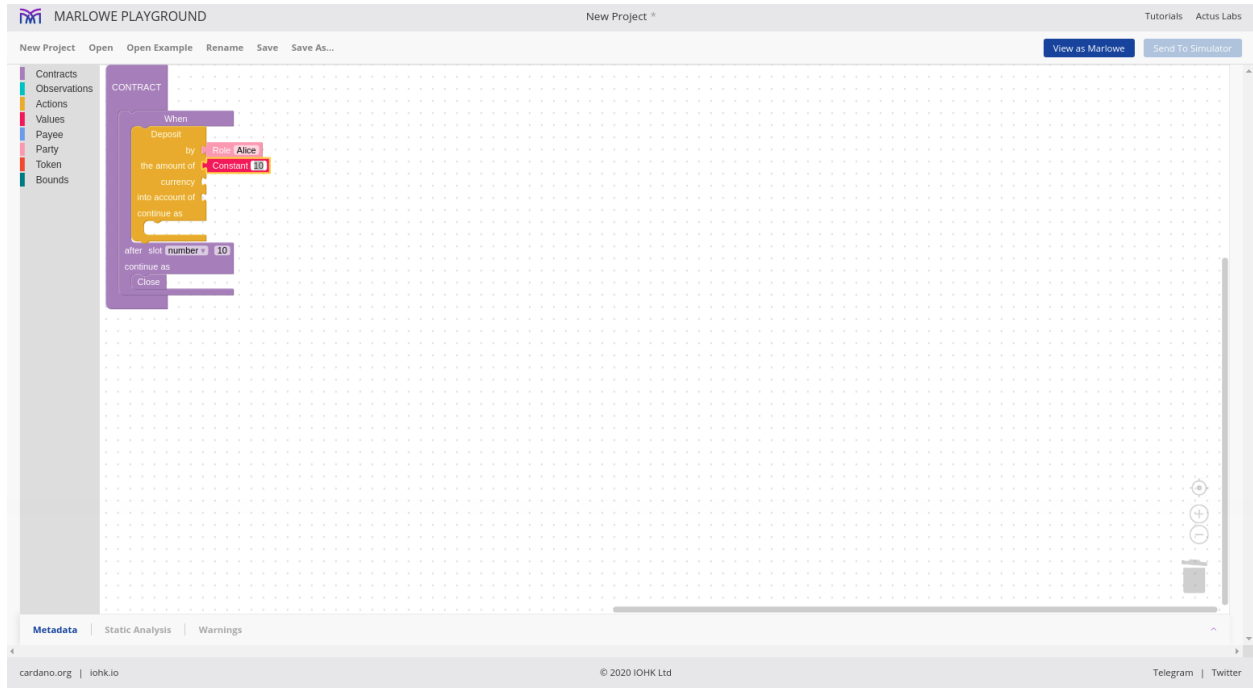
So at this point, if both these actions happen, Alice has deposited 10 into her internal account and Bob has deposited 10 into his external account.

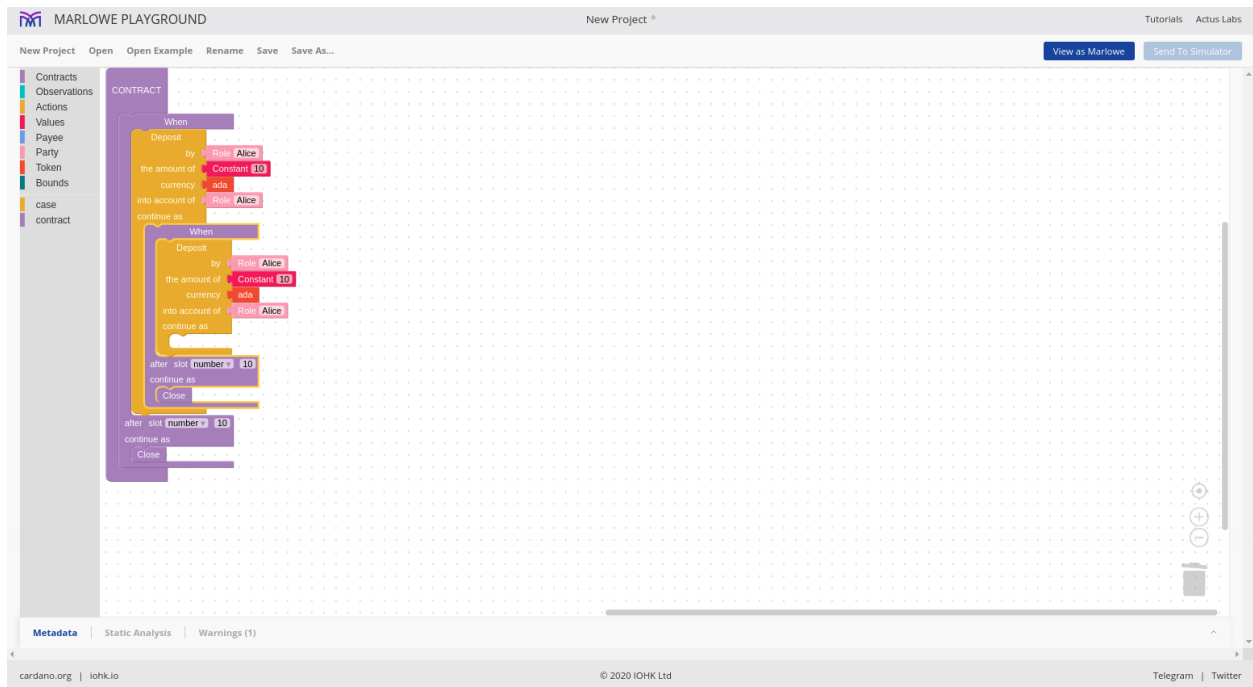
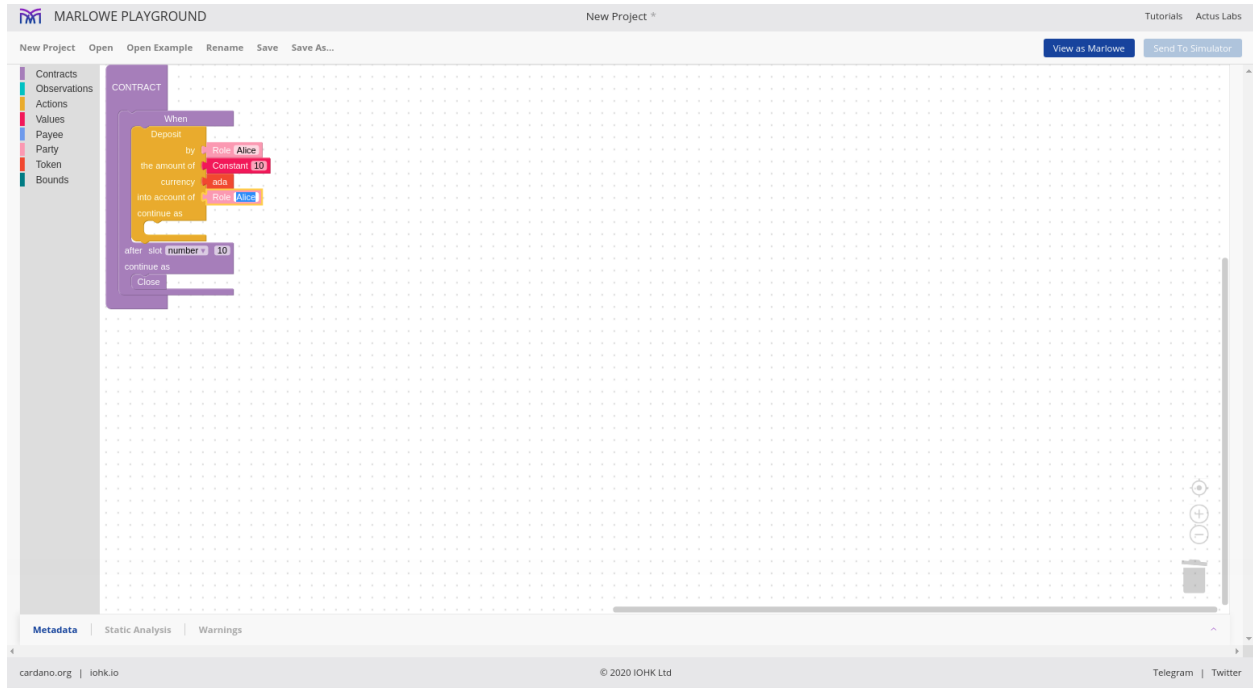
Now we want Charlie to make a choice, and this is again an external action, so again we need a *When*, but this time it's not a deposit so let's delete the deposit. Then let's change the timeout to 30 to give Charlie 10 slots to make his choice.

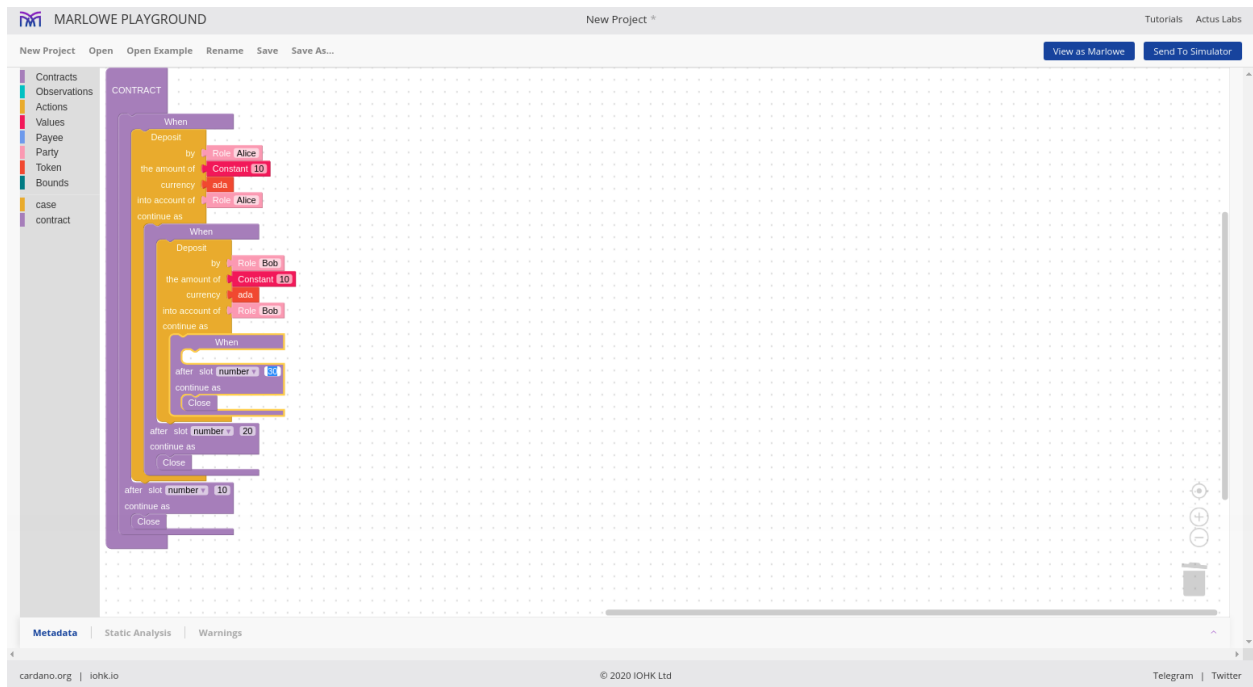
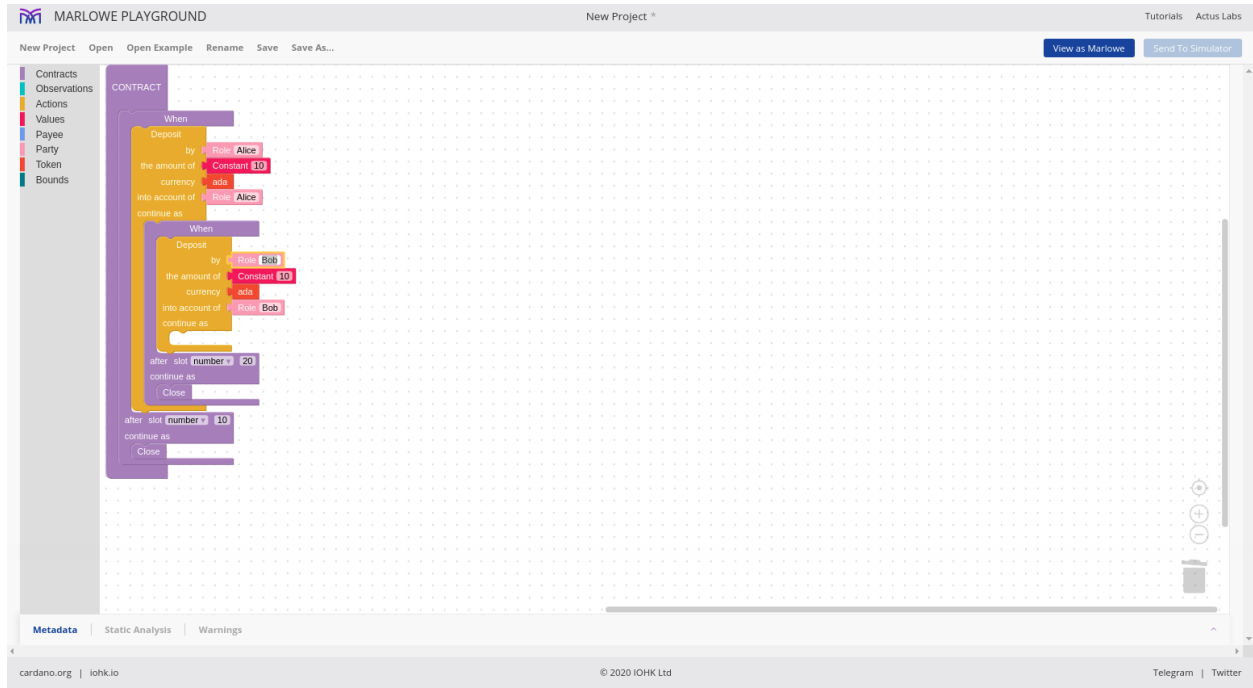




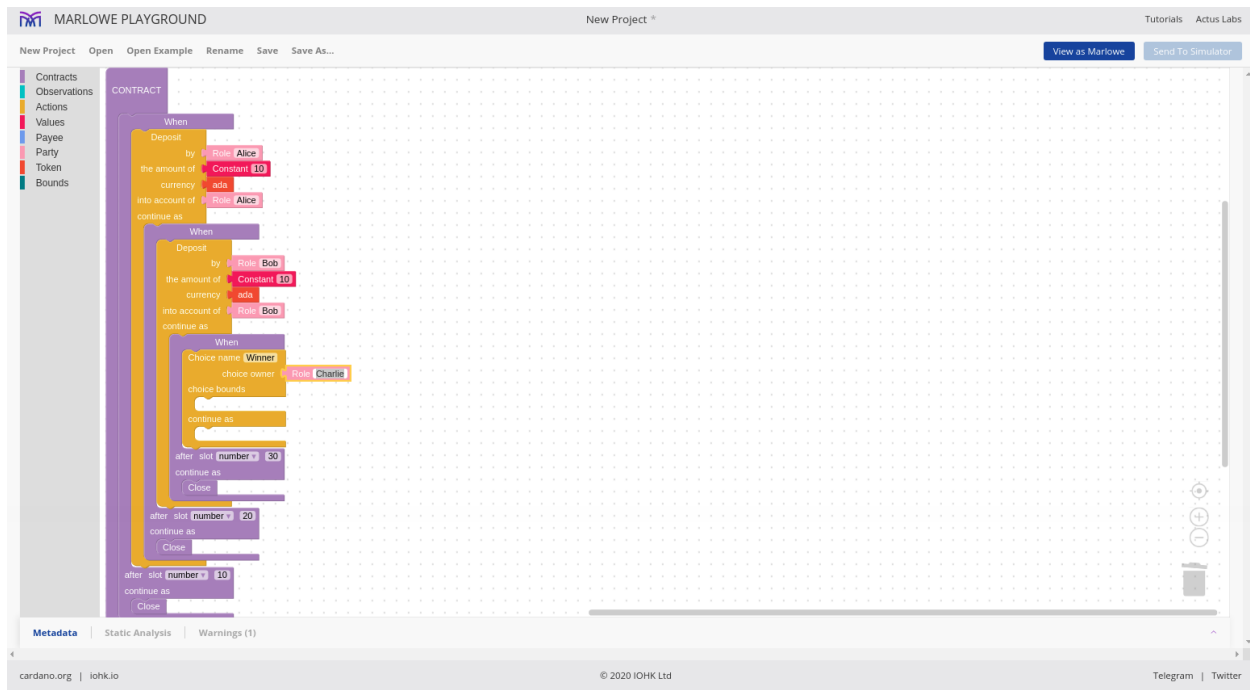








Now we need a different action. Where earlier we had *Deposit*, now we pick the *Choice* action. We can give it a name, let's say *Winner*. We must say who makes the choice, so that's will be Charlie, and now we must specify what values this choice can have.



That's numeric so because Charlie is supposed to choose between Alice and Bobs, which is two choices, I can pick arbitrary values like one and two. One for Alice, two for Bob. That's already the default so that's fine.

This allows Charlie to choose one or two.

Then if and when Charlie makes a choice, we continue, and it now depends on the choice that he has made. If he chose Alice then Alice must get all the money, if he chose Bob then Bob must get all the money.

So we will add an *If* conditional.

Then we will add an observation to check if Alice is the winner. The observation we add is the *value is equal to* observation.

To see if it is Alice, we will use the *Choice by* option to ask if Charlie's *Winner* name is equal to Alice.

In the *then* branch we now take a pay contract. The payee is who gets the money - it can be an internal account or it can be an external party. In this case it doesn't matter, because when we close, all the parties get the money from the internal accounts as well.

So, we'll just pick Alice's internal account.

We will pay 10 Ada.

So who pays? It must be an internal account because this pay contract is something that the contract has control over, so it is not an external action. So, payments are triggered from internal accounts and in this case, it is Bob's account.

So this now says: If Charlie picked 1, then pay from Bob's internal account 10 Ada to Alice's internal account.

After this we can just close. And when we close, all the internal accounts will be paid to the external owners. At this point, Alices internal account will have 20 Ada, and when we close, she will get the 20 Ada paid to her.

And, if Charlie did not choose Alice, then we must pay to Bob. We can copy paste the previous Pay contract for this and make the necessary modifications.

Plutus Pioneer Program - Lecture #9

2,076 views • 3 Jun 2021

LIKE DISLIKE SHARE SAVE

Computer Science Presentations

IOHK | Developers' meetup for Cardano, University College...

MARLOWE PLAYGROUND

New Project \*

View as Marlowe Send To Simulator

Contracts Observations Actions Values Payee Party Token Bounds

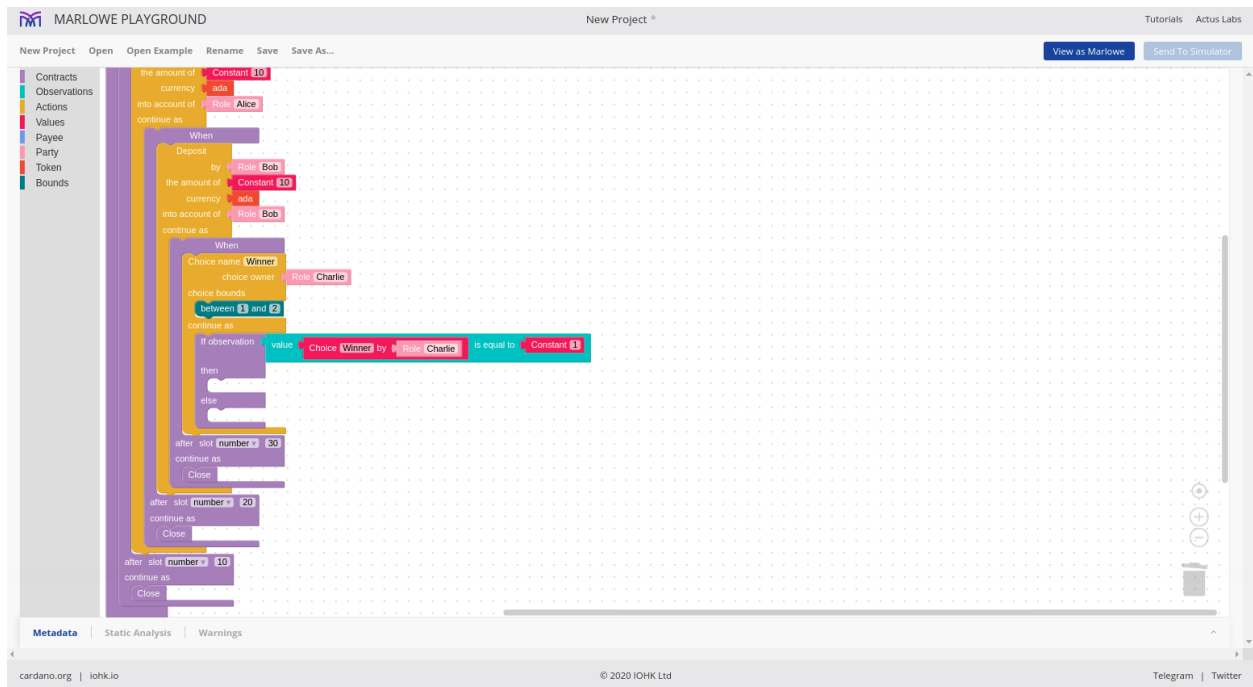
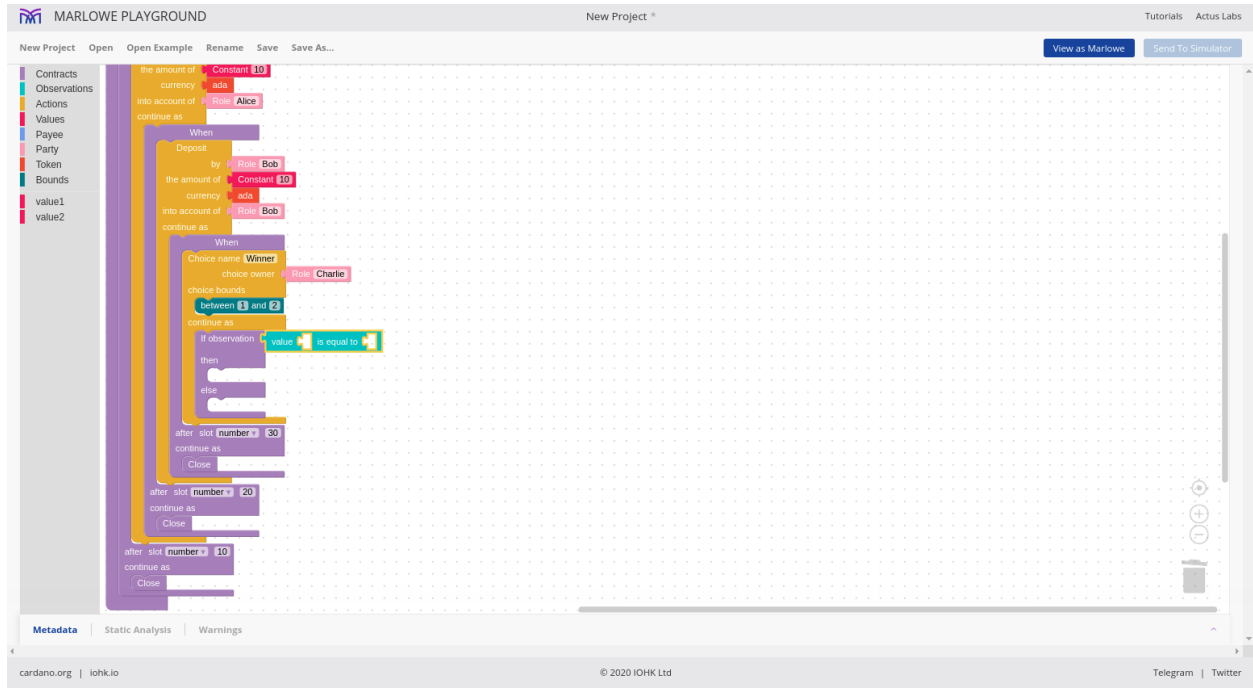
observation contract1 contract2

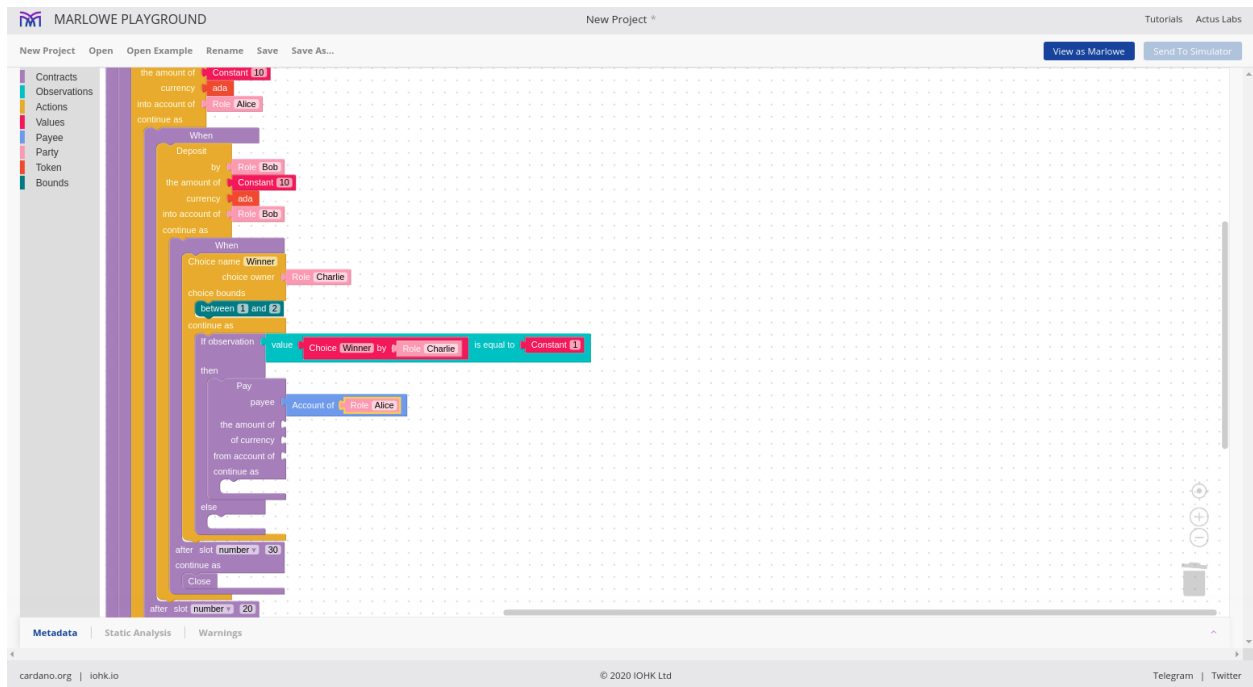
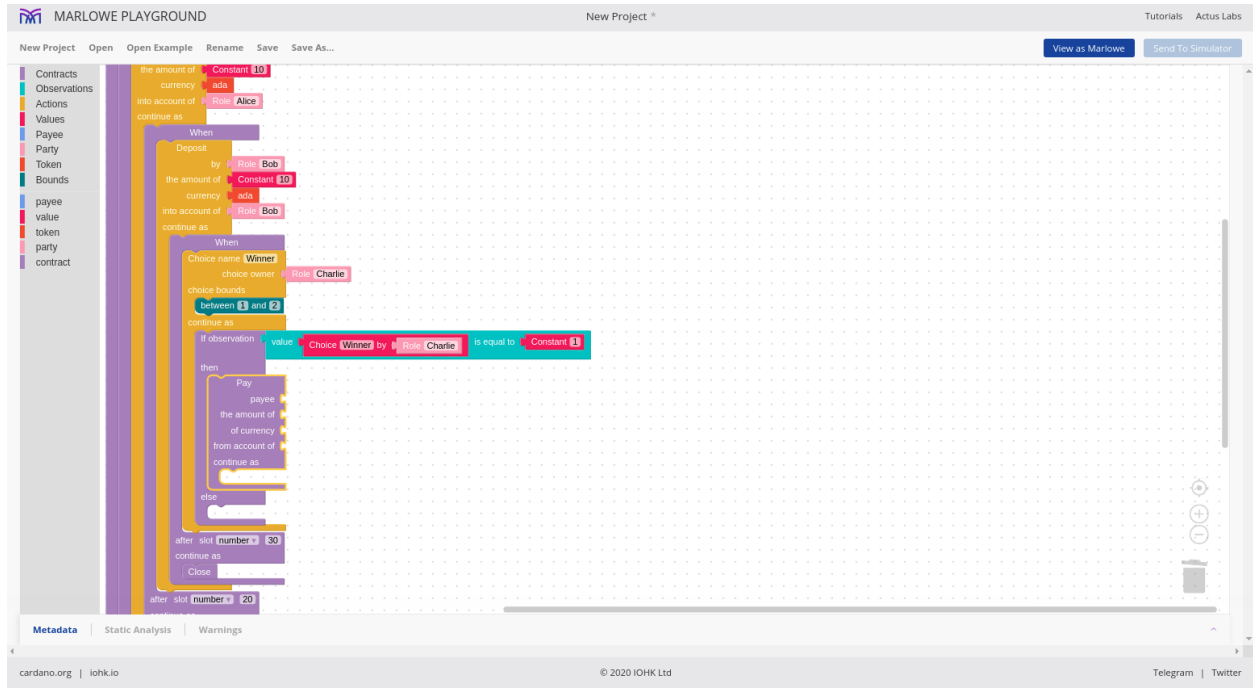
Metadata Static Analysis Warnings

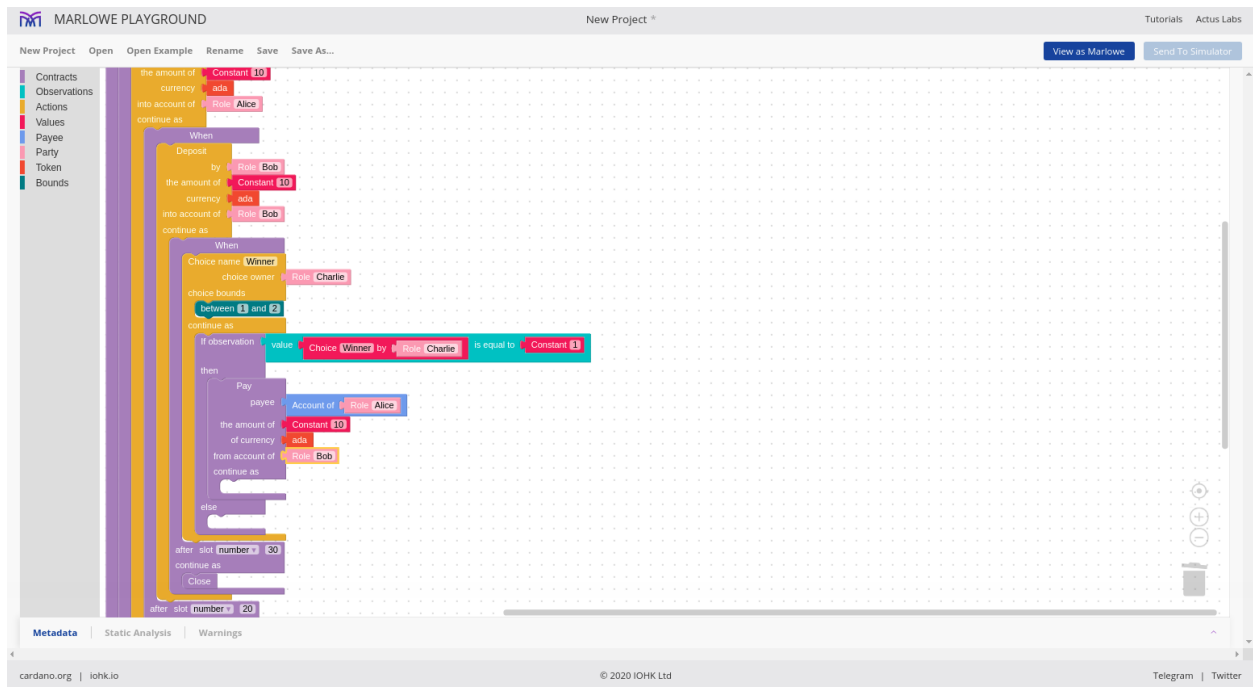
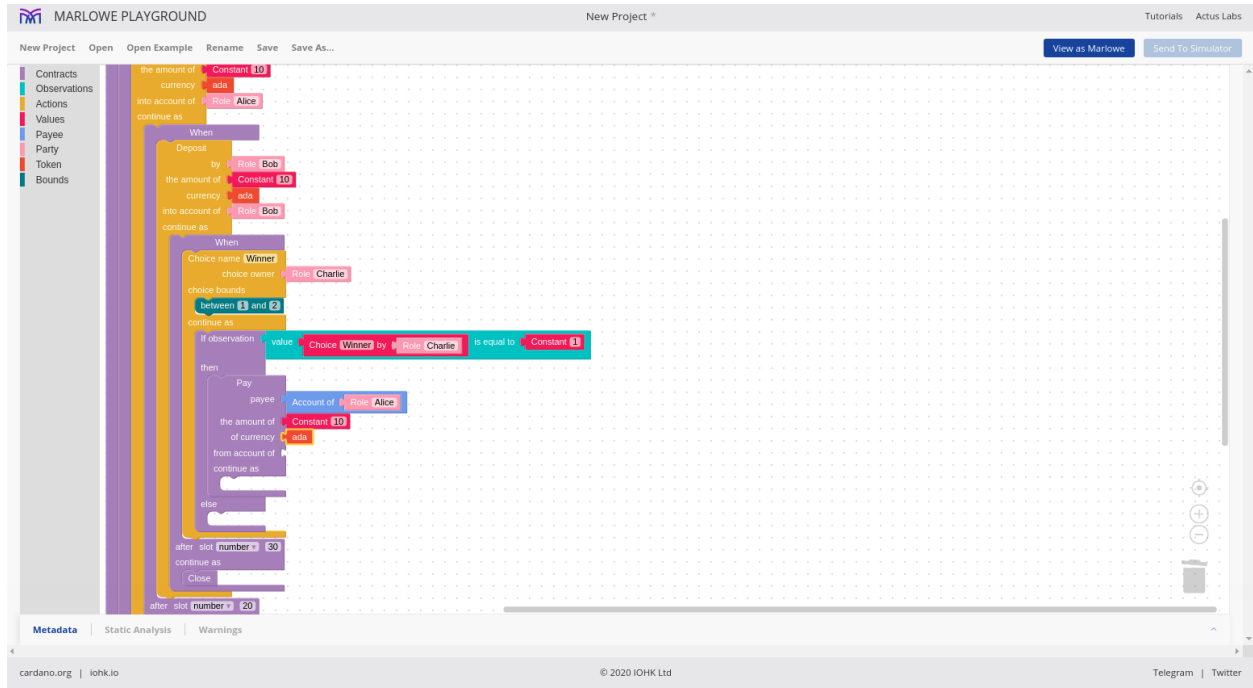
cardano.org | iohk.io

© 2020 IOHK Ltd

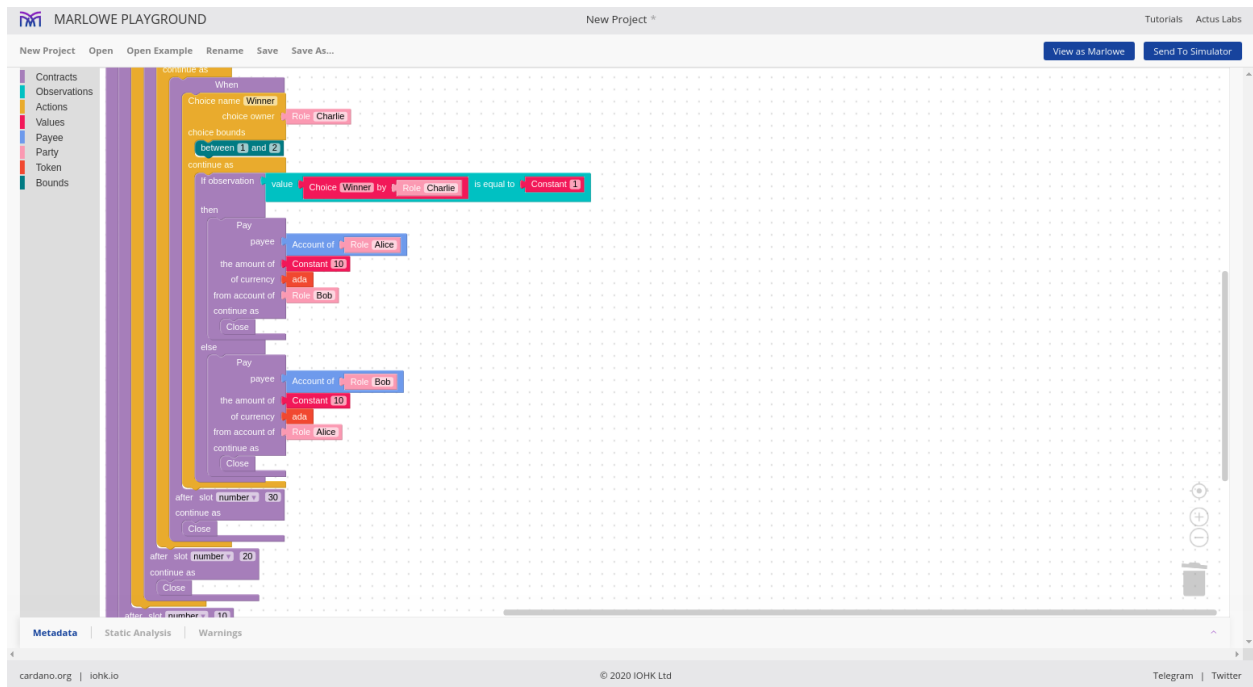
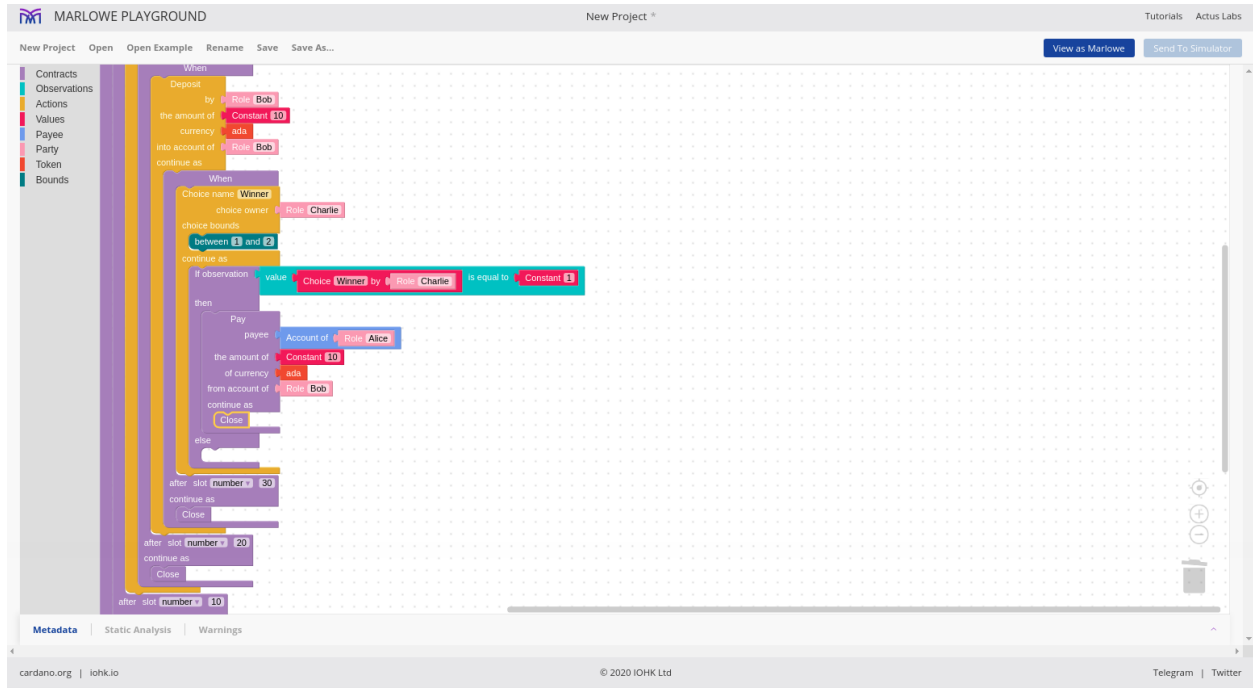
Telegram Twitter





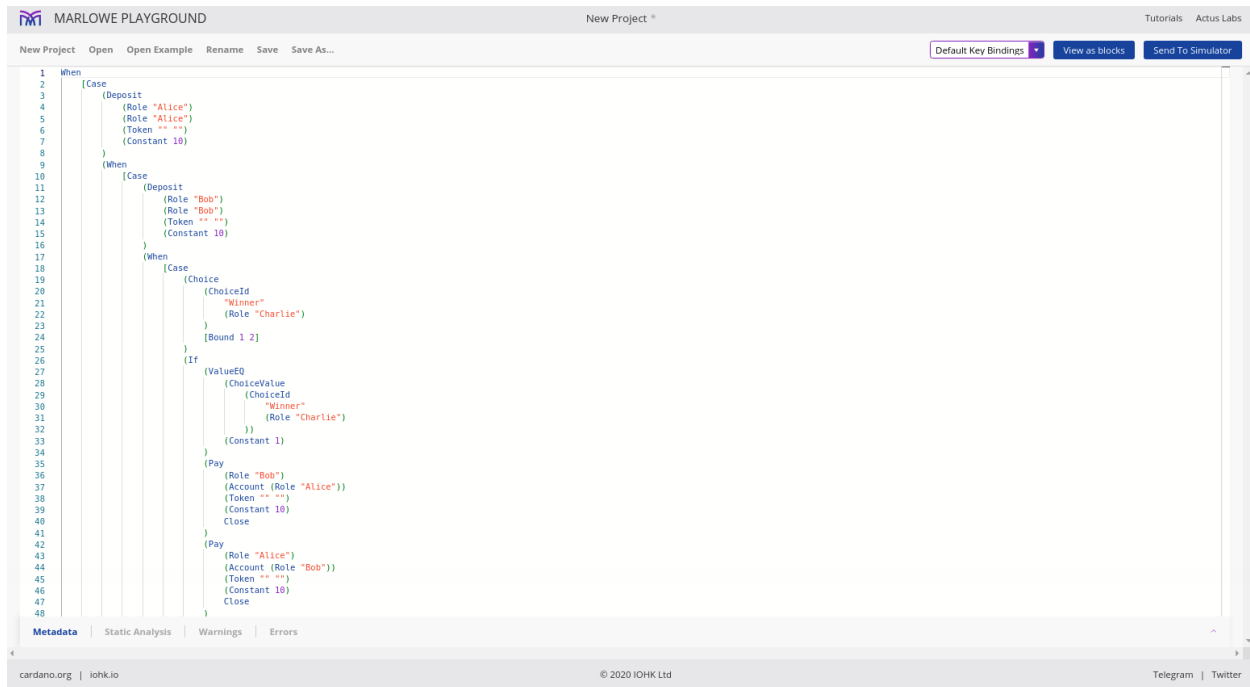






And this should be enough for our contract.

Now we can, for example, look at the pure Marlowe. This is the value of the Marlowe data type called *Contract*.



```
1 when
2   [Case
3     (Deposit
4       (Role "Alice")
5       (Role "Alice")
6       (Token == ""))
7     (Constant 10)
8   )
9   when
10    [Case
11      (Deposit
12        (Role "Bob")
13        (Role "Bob")
14        (Token == ""))
15      (Constant 10)
16    )
17    when
18      [Case
19        (Choice
20          (ChoiceId
21            "Winner"
22            (Role "Charlie"))
23          )
24        (Bound 1 2)
25      )
26      If
27        (ValueEq
28          (ChoiceValue
29            (ChoiceId
30              "Winner"
31              (Role "Charlie"))
32            )
33          (Constant 1)
34        )
35        (Pay
36          (Role "Bob")
37          (Account (Role "Alice")))
38          (Token == "")
39          (Constant 10)
40        )
41        Close
42      )
43      (Pay
44        (Role "Alice")
45        (Account (Role "Bob")))
46        (Token == "")
47        (Constant 10)
48      )
49      Close
50    )
51  ]
```

And we can send it to the simulator.

We can start the simulation.

Now, whenever there is a *When*, we get prompted for which of the available actions to take. In our case we only ever have one available action at each point.

So in the first *When*, either Alice makes her deposit, or the timeout is reached.

If we wait for the timeout it is very boring. The contract is reduced to *Close*, and nothing happened.

If, however, she makes the deposit, then this contract simplifies - it reduces to what happens after she makes the deposit.

And now we can see we are in the second *When*, where we are waiting for Bob's deposit. Again, he can choose not to deposit. If he does that, then we can see the actions in the transaction log that Alice deposited 10 Ada and the contract pays this back to Alice upon close.

It is more interesting though if Bob also makes his deposit.

Now the contract has simplified again. Now we are in the *When* where the only available action is that Charlie chooses a winner.

If Charlie doesn't do anything and the contract times out, Bob and Alice both get their money back.

If Charlie picks Alice (choice 1), then we see that the contract pays 20 Ada to Alice.

If instead he picks choice 2, then the contract pays 20 Ada to Bob.

Let's now reset the contract.

We will copy the Marlowe code to the clipboard, then create a new Haskell project.

In the Haskell editor there is a template.

All this program does is to take a Marlowe contract, and then pretty prints it. This is then used to, for example, run in the simulator.

MARLOWE PLAYGROUND
New Project \*
Tutorials Actus Labs

New Project Open Open Example Rename Save Save As...
Edit source

```

1 When
2   [Case
3     (Deposit
4       (Role "Alice")
5       (Role "Alice")
6       (Token "" ""))
7       (Constant 10)
8     )
9   ]
10  [When
11    [Case
12      (Deposit
13        (Role "Bob")
14        (Role "Bob")
15        (Token "" ""))
16        (Constant 10)
17      )
18    ]
19    [When
20      [Case
21        (Choice
22          (ChoiceId
23            "Winner"
24            (Role "Charlie"))
25          )
26        ]
27        [Bound 1 2]
28      )
29      [If
30        (ValueEq
31          (ChoiceValue
32            (ChoiceId
33              "Winner"
34              (Role "Charlie"))
35            ))
36          (Constant 1)
37        )
38        (Pay
39          (Role "Bob")
40          (Account (Role "Alice")))
41          (Token "" "")
42          (Constant 10)
43          Close
44        )
45      )
46      (Pay
47        (Role "Alice")
48        (Account (Role "Bob")))
49        (Token "" "")
50        (Constant 10)
51        Close
52      )
53    ]
54  ]
55 ]

```


SIMULATION HAS NOT STARTED YET

Initial slot:

Start simulation

Current State
< >

cardano.org | iohk.io
© 2020 IOHK Ltd
Telegram Twitter



Marlowe Playgrounds

[New Project](#)
[Open](#)
[Open Example](#)
[Rename](#)
[Save](#)
[Save As...](#)

New Project

current slot: 0

expiration slot: 30

ACTIONS

Participant **Alice**

Deposit 10 units of ADA into account of Alice as Alice

Other Actions

Move to slot 10

Undo

Reset

TRANSACTION LOG

Action

Slot

1 When

2 [Case

3 [Deposit

4 (Role "Alice")

5 (Role "Alice")

6 (Token "ADA")

7 (Constant 10)

8 ]

9 ]

10 ]

11 [Case

12 [Deposit

13 (Role "Bob")

14 (Role "Bob")

15 (Token "ADA")

16 (Constant 10)

17 ]

18 ]

19 ]

20 ]

21 ]

22 ]

23 ]

24 ]

25 ]

26 ]

27 ]

28 ]

29 ]

30 ]

31 ]

32 ]

33 ]

34 ]

35 ]

36 ]

37 ]

38 ]

39 ]

40 ]

41 ]

42 ]

43 ]

44 ]

45 ]

46 ]

47 ]

48 ]

Current State

1

Close

current slot: 10

expiration slot: Closed

ACTIONS

No valid inputs can be added to the transaction

Undo

Reset

TRANSACTION LOG

Action	Slot
--------	------

Current State

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter

1

When

2

Case

3

(Deposit

4

(Role "Bob")

5

(Role "Bob")

6

(Token == "ADA")

7

(Constant 10)

8

)

9

When

10

Case

11

(ChoiceId

12

"Winner"

13

(Role "Charlie")

14

)

15

[Bound 1 2]

16

)

17

(If

18

(ValueEq

19

(ChoiceValue

20

(ChoiceId

21

"Winner"

22

(Role "Charlie")

23

)

24

(Constant 1)

25

)

26

(Pay

27

(Role "Bob")

28

(Account (Role "Alice"))

29

(Token == "ADA")

30

(Constant 10)

31

Close

32

)

33

(Pay

34

(Role "Alice")

35

(Account (Role "Bob"))

36

(Token == "ADA")

37

(Constant 10)

38

Close

39

)

40

)

41

30 Close

42

30 Close

43

20 Close

44

20 Close

current slot: 0

expiration slot: 30

ACTIONS

Participant **Bob**

Deposit 10 units of ADA into account of Bob as Bob

Other Actions

Move to slot 20

Undo

Reset

TRANSACTION LOG

Action	Slot
Deposit 10 units of ADA into account of Alice as Alice	0

Current State

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter

**MARLOWE PLAYGROUND**

New Project \*

Tutorials Actus Labs

New Project Open Open Example Rename Save Save As...

1 Close

current slot: 20

expiration slot: Closed

ACTIONS

No valid inputs can be added to the transaction

Undo Reset

TRANSACTION LOG

Action	Slot
Deposit 10 units of ADA into account of Alice as Alice	0
The contract pays 10 units of ADA to participant Alice	20

Current State

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter

**MARLOWE PLAYGROUND**

New Project \*

Tutorials Actus Labs

New Project Open Open Example Rename Save Save As...

```

1 when
2   [Case
3     (Choice
4       (ChoiceId
5         "Winner"
6       )
7       (Role "Charlie")
8     )
9     [Bound 1 2]
10    ]
11  (If
12    (ValueEq
13      (ChoiceValue
14        (ChoiceId
15          "Winner"
16        )
17        (Role "Charlie")
18      )
19      (Constant 1)
20    )
21    (Pay
22      (Role "Bob")
23      (Account (Role "Alice"))
24      (Token "ADA")
25      (Constant 10)
26    )
27    (Pay
28      (Role "Alice")
29      (Account (Role "Bob"))
30      (Token "ADA")
31      (Constant 10)
32    )
33  )
34 ] Close

```

current slot: 0

expiration slot: 30

ACTIONS

Participant **Charlie**

Choice "Winner":

Other Actions

Move to slot

Undo Reset

TRANSACTION LOG

Action	Slot
Deposit 10 units of ADA into account of Alice as Alice	0
Deposit 10 units of ADA into account of Bob as Bob	0

Current State

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter

MARLOWE PLAYGROUND

New Project \*

TutorialsActus Labs

New ProjectOpenOpen ExampleRenameSaveSave As...

1Close

current slot: 30expiration slot: Closed

ACTIONS

No valid inputs can be added to the transaction

UndoReset

TRANSACTION LOG

Action	Slot
Deposit 10 units of ADA into account of Alice as Alice	0
Deposit 10 units of ADA into account of Bob as Bob	0
The contract pays 10 units of ADA to participant Bob	30
The contract pays 10 units of ADA to participant Alice	30

Current State

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter

MARLOWE PLAYGROUND

New Project \*

TutorialsActus Labs

New ProjectOpenOpen ExampleRenameSaveSave As...

1Close

current slot: 0expiration slot: Closed

ACTIONS

No valid inputs can be added to the transaction

UndoReset

TRANSACTION LOG

Action	Slot
Deposit 10 units of ADA into account of Alice as Alice	0
Deposit 10 units of ADA into account of Bob as Bob	0
Participant Charlie chooses the value 1 for choice with id "Winner"	0
The contract pays 20 units of ADA to participant Alice	0

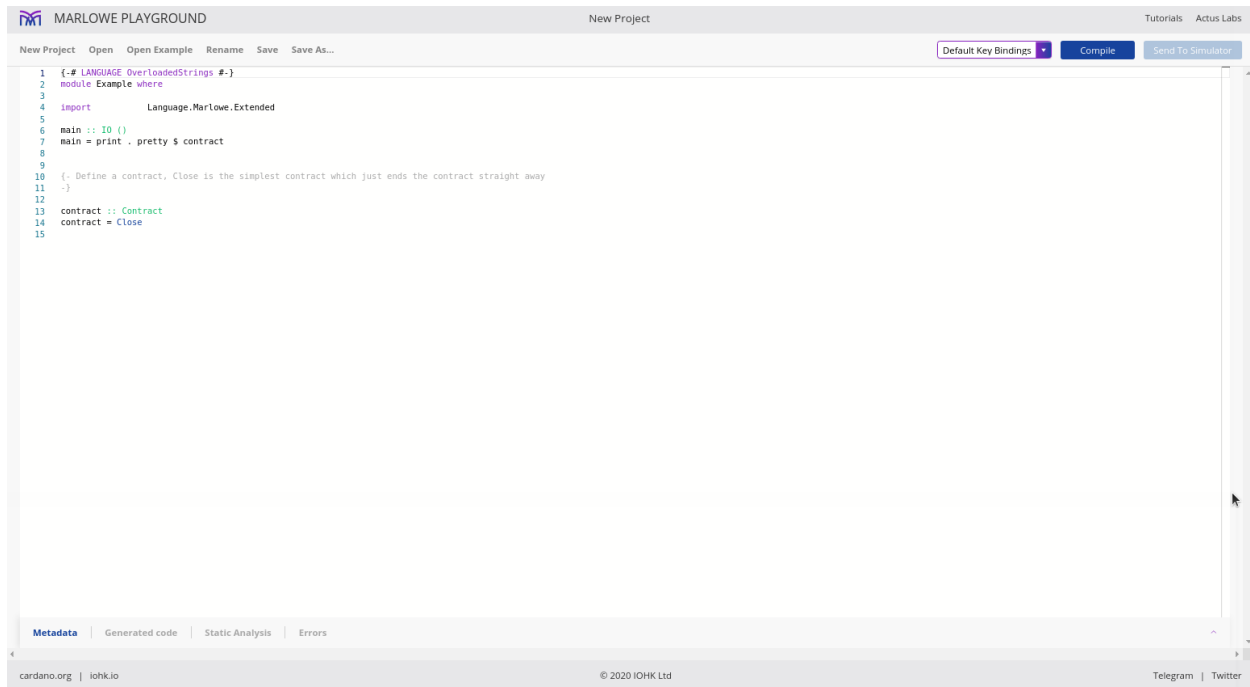
Current State

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter

## 9.4. Playing in the Playground 331



Instead of `Close`, we can paste what we just copied to the clipboard.

We can then compile this, and send it to the simulator and it should behave exactly as before.

There we don't really see the benefit of doing it in Haskell, we could just as well do it in Blockly, although you may find that Blockly is really only useful for learning and writing extremely simple contracts. We have just written a simple contract and already it was starting to get quite unwieldy in the Blockly editor. If you do something more complicated it can start to get very confusing in the editor.

But, we can do other things in this Haskell program. We don't have to literally define a contract. We can use the whole power of Haskell to help us to write the contract.

For example, we can see a lot of repetition because we always have the Alice, Bob and Charlie roles. We could define these separately.

Note that we can use overloaded string literals here because the `fromString` function uses the `Role` constructor for `Party`.

We can also define a constant for the deposit of 10 Ada.

For (Token `"ada"`), we can replace this with the `ada` abbreviation.

We can also simplify Charlie's `ChoiceId`.

Now it is already cleaned up quite a bit.

It's possible to do more sophisticated things. Our contract is slightly asymmetric even though it sounds like a symmetric situation. Alice and Bob are completely symmetric, but in our contract, Alice has to deposit first.

What we could do is to allow Bob to deposit first as well. In the outermost *When* we would have two deposits - one where Alice deposits, and one where Bob deposits.

Let's make a little helper function. It takes two *Partys* - the party that deposits first and the party that deposits second and then it returns a *Case*. We can use this to parameterise Alice and Bob as *x* and *y* in the *Case*. Note that we only need to do this for the deposits, the part where Charlie makes his choice can remain the same, with Alice and Bob continuing to be represented by 1 and 2 respectively.



The screenshot shows the Marlowe Playground interface with a new project. The code defines a contract where Alice deposits 10 tokens, and then Bob and Charlie can interact with it. The contract is defined as follows:

```

1 {-# LANGUAGE OverloadedStrings #-}
2 module Example where
3
4 import Language.Marlowe.Extended
5
6 main :: IO ()
7 main = print . pretty $ contract
8
9
10 {- Define a contract. Close is the simplest contract which just ends the contract straight away
11 -}
12
13 contract :: Contract
14 contract = when
15   [Case
16     (Deposit
17       (Role "Alice")
18       (Role "Alice")
19       (Token -- "")
20       (Constant 10)
21     )
22     when
23       [Case
24         (Deposit
25           (Role "Bob")
26           (Role "Bob")
27           (Token -- "")
28           (Constant 10)
29         )
30         when
31           [Case
32             (Choice
33               (ChoiceId
34                 "Winner"
35                 (Role "Charlie")
36               )
37             )
38             [Bound 1 2]
39           )
40           [If
41             (ValueEq
42               (ChoiceValue
43                 (ChoiceId
44                   "Winner"
45                   (Role "Charlie")
46                 )
47               )
48             )
49             (Constant 1)
50           )
51           [Pav
52             (Constant 1)
53           )
54         ]
55       ]
56     ]
57   ]
58 
```

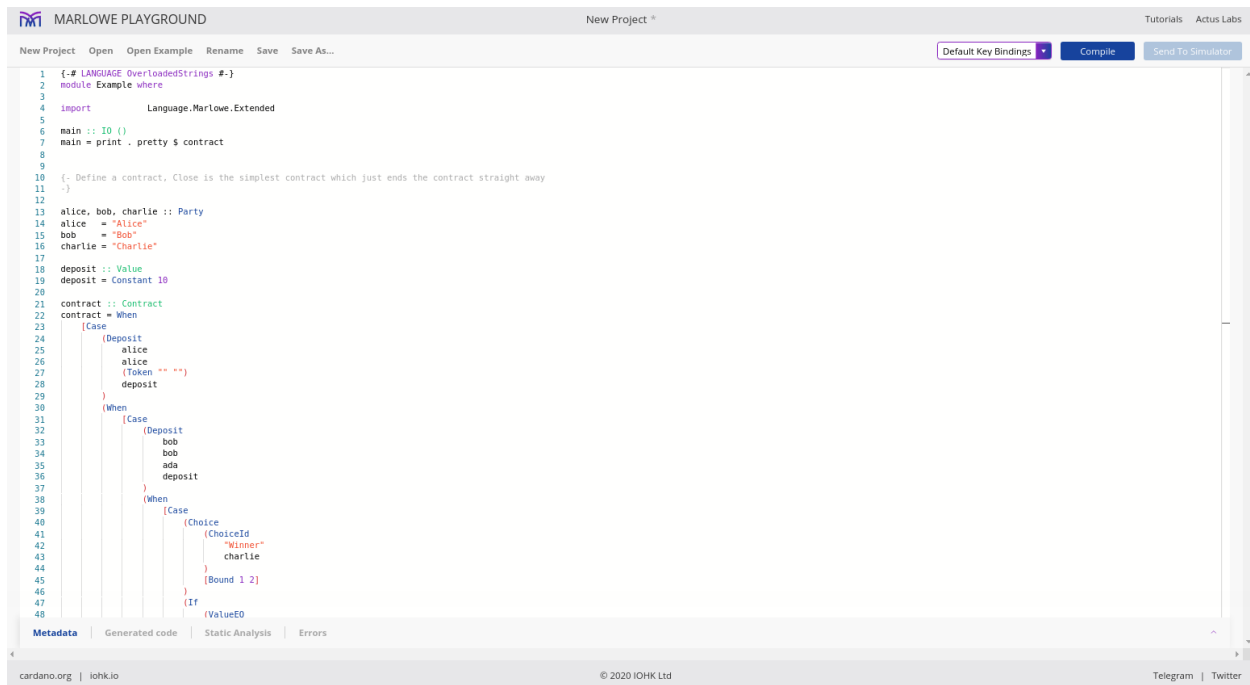
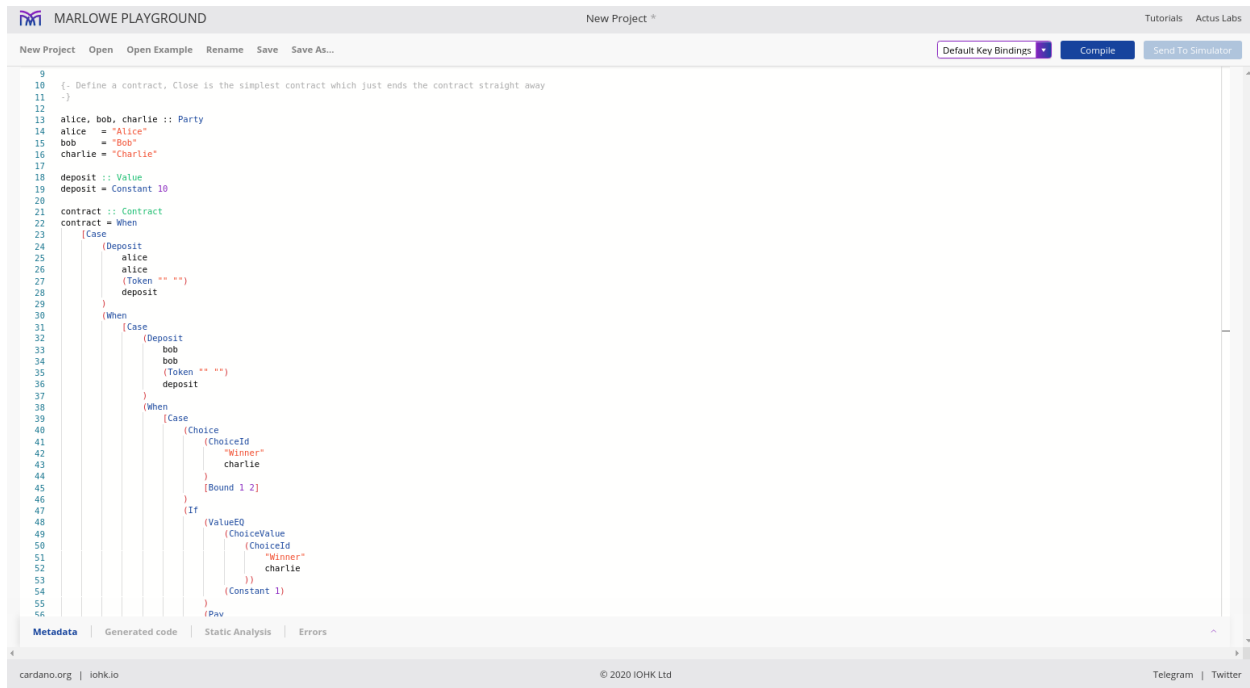
The interface includes a menu bar with 'New Project', 'Open', 'Open Example', 'Rename', 'Save', and 'Save As...'. There are buttons for 'Default Key Bindings', 'Compile', and 'Send To Simulator'. The bottom status bar shows 'cardano.org | lohki.io', '© 2020 IOHK Ltd', and 'Telegram | Twitter'.

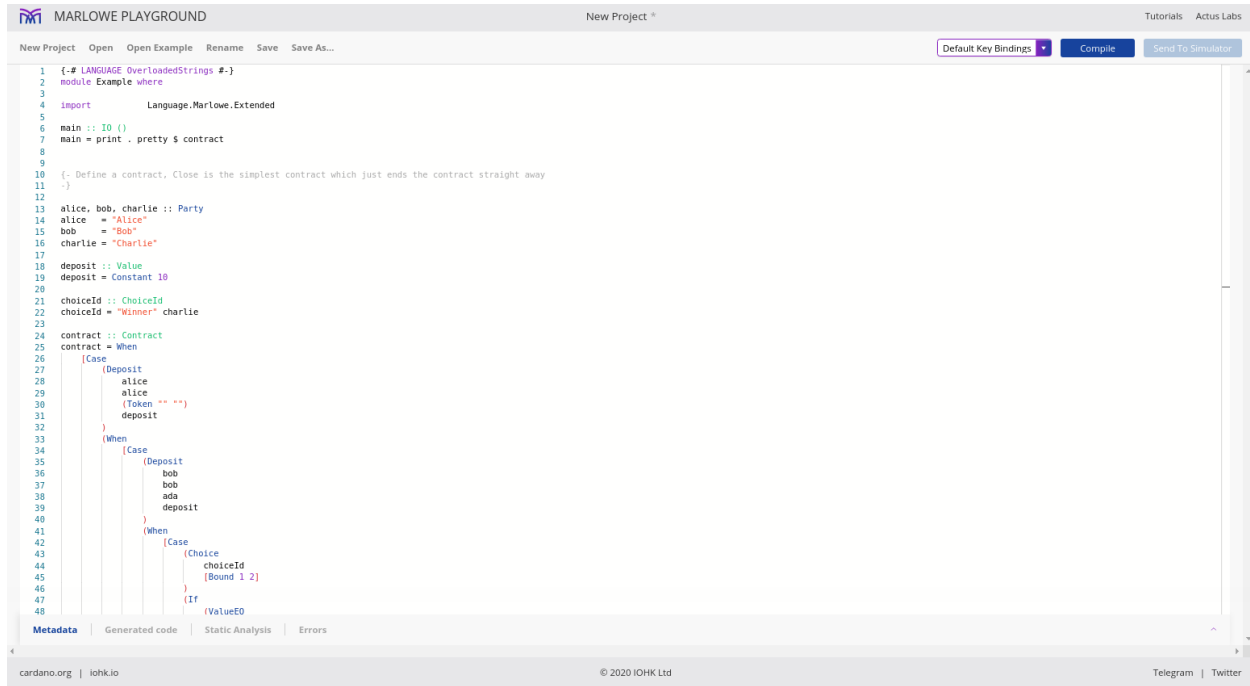
The screenshot shows the Marlowe Playground interface with a new project. The code defines a contract where Alice deposits 10 tokens, and then Bob and Charlie can interact with it. The contract is defined as follows:

```

1 {-# LANGUAGE OverloadedStrings #-}
2 module Example where
3
4 import Language.Marlowe.Extended
5
6 main :: IO ()
7 main = print . pretty $ contract
8
9
10 {- Define a contract. Close is the simplest contract which just ends the contract straight away
11 -}
12
13 alice, bob, charlie :: Party
14 alice = "Alice"
15 bob = "Bob"
16 charlie = "Charlie"
17
18 contract :: Contract
19 contract = when
20   [Case
21     (Deposit
22       alice
23       alice
24       (Token -- "")
25       (Constant 10)
26     )
27     when
28       [Case
29         (Deposit
30           bob
31           bob
32           (Token -- "")
33           (Constant 10)
34         )
35         when
36           [Case
37             (Choice
38               (ChoiceId
39                 "Winner"
40                 charlie
41               )
42             )
43             [Bound 1 2]
44           )
45           [If
46             (ValueEq
47               (ChoiceValue
48                 (ChoiceId
49                   "Winner"
50                 )
51               )
52             )
53             (Constant 1)
54           )
55           [Pav
56             (Constant 1)
57           )
58         ]
59       ]
60     ]
61   ]
62 
```

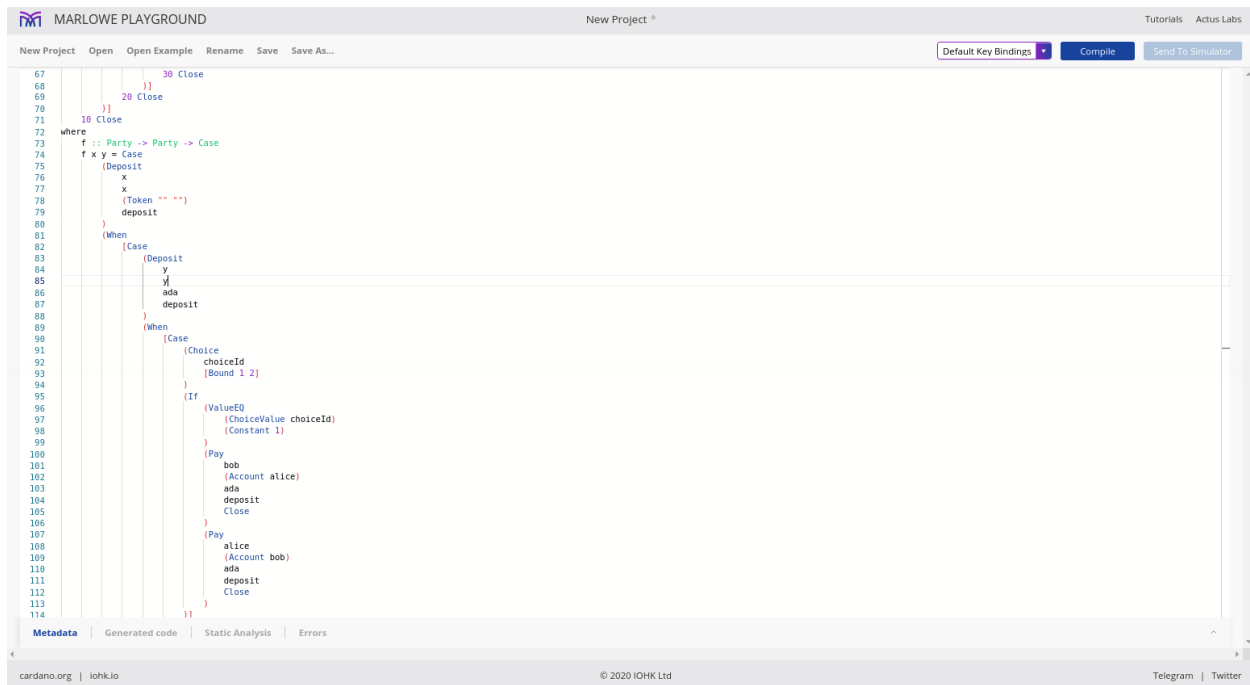
The interface includes a menu bar with 'New Project', 'Open', 'Open Example', 'Rename', 'Save', and 'Save As...'. There are buttons for 'Default Key Bindings', 'Compile', and 'Send To Simulator'. The bottom status bar shows 'cardano.org | lohki.io', '© 2020 IOHK Ltd', and 'Telegram | Twitter'.





The screenshot shows the Marlowe Playground interface with a new project. The code defines a contract where Alice, Bob, and Charlie are parties. Alice has a deposit of 10 tokens. The contract is a choice between Charlie winning (if Alice's deposit is 10) and a choice between Bob and Ada winning (if Alice's deposit is not 10). The contract is named "winner" and is of type "Contract".

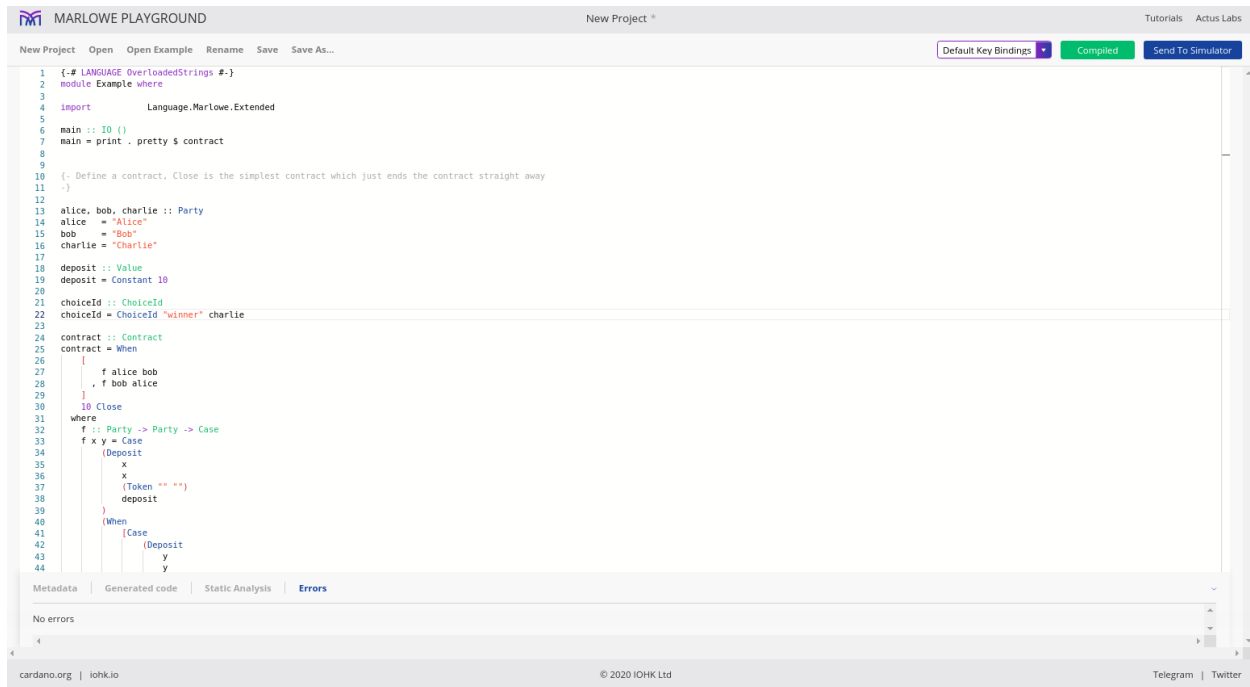
```
1 {-# LANGUAGE OverloadedStrings #-}
2 module Example where
3
4 import Language.Marlowe.Extended
5
6 main :: IO ()
7 main = print . pretty $ contract
8
9
10 {- Define a contract, Close is the simplest contract which just ends the contract straight away
11 -}
12
13 alice, bob, charlie :: Party
14 alice = "Alice"
15 bob = "Bob"
16 charlie = "Charlie"
17
18 deposit :: Value
19 deposit = Constant 10
20
21 choiceId :: ChoiceId
22 choiceId = "winner" charlie
23
24 contract :: Contract
25 contract = When
26   [Case
27     (Deposit
28       alice
29       (Token ** **))
30     deposit
31   )
32   (When
33     [Case
34       (Deposit
35         bob
36         (Token ** **))
37       (Deposit
38         ada
39         (Token ** **))
40     ]
41     (When
42       [Case
43         (Choice
44           choiceId
45           (Bound 1 2))
46       ]
47       (If
48         (ValueEq
49           (ChoiceValue choiceId)
50           (Constant 1))
51         (Pay
52           bob
53           (Account alice)
54           (Account bob)
55           (Account ada)
56           deposit
57           Close)
58         (Pay
59           alice
60           (Account bob)
61           (Account ada)
62           deposit
63           Close)
64       )
65     )
66   ]
67   )
68   )
69   )
70   )
71   )
72   )
73   )
74   )
75   )
76   )
77   )
78   )
79   )
80   )
81   )
82   )
83   )
84   )
85   )
86   )
87   )
88   )
89   )
90   )
91   )
92   )
93   )
94   )
95   )
96   )
97   )
98   )
99   )
100  )
101  )
102  )
103  )
104  )
105  )
106  )
107  )
108  )
109  )
110  )
111  )
112  )
113  )
114  )
```



The screenshot shows the Marlowe Playground interface with a new project. The code defines a contract where Alice, Bob, and Charlie are parties. Alice has a deposit of 10 tokens. The contract is a choice between Charlie winning (if Alice's deposit is 10) and a choice between Bob and Ada winning (if Alice's deposit is not 10). The contract is named "winner" and is of type "Contract".

```
67
68
69
70
71
72 where
73   f :: Party -> Party -> Case
74   f x y = Case
75     (Deposit
76       x
77       (Token ** **))
78     deposit
79   )
80   (When
81     [Case
82       (Deposit
83         y
84         (Token ** **))
85       (Deposit
86         ada
87         (Token ** **))
88     ]
89     (When
90       [Case
91         (Choice
92           choiceId
93           (Bound 1 2))
94       ]
95       (If
96         (ValueEq
97           (ChoiceValue choiceId)
98           (Constant 1))
99         (Pay
100           bob
101           (Account alice)
102           (Account bob)
103           (Account ada)
104           deposit
105           Close)
106         (Pay
107           alice
108           (Account bob)
109           (Account ada)
110           deposit
111           Close)
112       )
113     )
114   ]
115   )
116   )
117   )
118   )
119   )
120   )
121   )
122   )
123   )
124   )
125   )
126   )
127   )
128   )
129   )
130   )
131   )
132   )
133   )
134   )
135   )
136   )
137   )
138   )
139   )
140   )
141   )
142   )
143   )
144   )
145   )
146   )
147   )
148   )
149   )
150   )
151   )
152   )
153   )
154   )
155   )
156   )
157   )
158   )
159   )
160   )
161   )
162   )
163   )
164   )
165   )
166   )
167   )
168   )
169   )
170   )
171   )
172   )
173   )
174   )
175   )
176   )
177   )
178   )
179   )
180   )
181   )
182   )
183   )
184   )
185   )
186   )
187   )
188   )
189   )
190   )
191   )
192   )
193   )
194   )
195   )
196   )
197   )
198   )
199   )
200   )
201   )
202   )
203   )
204   )
205   )
206   )
207   )
208   )
209   )
210   )
211   )
212   )
213   )
214   )
215   )
216   )
217   )
218   )
219   )
220   )
221   )
222   )
223   )
224   )
225   )
226   )
227   )
228   )
229   )
230   )
231   )
232   )
233   )
234   )
235   )
236   )
237   )
238   )
239   )
240   )
241   )
242   )
243   )
244   )
245   )
246   )
247   )
248   )
249   )
250   )
251   )
252   )
253   )
254   )
255   )
256   )
257   )
258   )
259   )
260   )
261   )
262   )
263   )
264   )
265   )
266   )
267   )
268   )
269   )
270   )
271   )
272   )
273   )
274   )
275   )
276   )
277   )
278   )
279   )
280   )
281   )
282   )
283   )
284   )
285   )
286   )
287   )
288   )
289   )
290   )
291   )
292   )
293   )
294   )
295   )
296   )
297   )
298   )
299   )
300   )
301   )
302   )
303   )
304   )
305   )
306   )
307   )
308   )
309   )
310   )
311   )
312   )
313   )
314   )
315   )
316   )
317   )
318   )
319   )
320   )
321   )
322   )
323   )
324   )
325   )
326   )
327   )
328   )
329   )
330   )
331   )
332   )
333   )
334   )
335   )
336   )
337   )
338   )
339   )
340   )
341   )
342   )
343   )
344   )
345   )
346   )
347   )
348   )
349   )
350   )
351   )
352   )
353   )
354   )
355   )
356   )
357   )
358   )
359   )
360   )
361   )
362   )
363   )
364   )
365   )
366   )
367   )
368   )
369   )
370   )
371   )
372   )
373   )
374   )
375   )
376   )
377   )
378   )
379   )
380   )
381   )
382   )
383   )
384   )
385   )
386   )
387   )
388   )
389   )
390   )
391   )
392   )
393   )
394   )
395   )
396   )
397   )
398   )
399   )
400   )
401   )
402   )
403   )
404   )
405   )
406   )
407   )
408   )
409   )
410   )
411   )
412   )
413   )
414   )
415   )
416   )
417   )
418   )
419   )
420   )
421   )
422   )
423   )
424   )
425   )
426   )
427   )
428   )
429   )
430   )
431   )
432   )
433   )
434   )
435   )
436   )
437   )
438   )
439   )
440   )
441   )
442   )
443   )
444   )
445   )
446   )
447   )
448   )
449   )
450   )
451   )
452   )
453   )
454   )
455   )
456   )
457   )
458   )
459   )
460   )
461   )
462   )
463   )
464   )
465   )
466   )
467   )
468   )
469   )
470   )
471   )
472   )
473   )
474   )
475   )
476   )
477   )
478   )
479   )
480   )
481   )
482   )
483   )
484   )
485   )
486   )
487   )
488   )
489   )
490   )
491   )
492   )
493   )
494   )
495   )
496   )
497   )
498   )
499   )
500   )
501   )
502   )
503   )
504   )
505   )
506   )
507   )
508   )
509   )
510   )
511   )
512   )
513   )
514   )
515   )
516   )
517   )
518   )
519   )
520   )
521   )
522   )
523   )
524   )
525   )
526   )
527   )
528   )
529   )
530   )
531   )
532   )
533   )
534   )
535   )
536   )
537   )
538   )
539   )
540   )
541   )
542   )
543   )
544   )
545   )
546   )
547   )
548   )
549   )
550   )
551   )
552   )
553   )
554   )
555   )
556   )
557   )
558   )
559   )
560   )
561   )
562   )
563   )
564   )
565   )
566   )
567   )
568   )
569   )
570   )
571   )
572   )
573   )
574   )
575   )
576   )
577   )
578   )
579   )
580   )
581   )
582   )
583   )
584   )
585   )
586   )
587   )
588   )
589   )
590   )
591   )
592   )
593   )
594   )
595   )
596   )
597   )
598   )
599   )
600   )
601   )
602   )
603   )
604   )
605   )
606   )
607   )
608   )
609   )
610   )
611   )
612   )
613   )
614   )
615   )
616   )
617   )
618   )
619   )
620   )
621   )
622   )
623   )
624   )
625   )
626   )
627   )
628   )
629   )
630   )
631   )
632   )
633   )
634   )
635   )
636   )
637   )
638   )
639   )
640   )
641   )
642   )
643   )
644   )
645   )
646   )
647   )
648   )
649   )
650   )
651   )
652   )
653   )
654   )
655   )
656   )
657   )
658   )
659   )
660   )
661   )
662   )
663   )
664   )
665   )
666   )
667   )
668   )
669   )
670   )
671   )
672   )
673   )
674   )
675   )
676   )
677   )
678   )
679   )
680   )
681   )
682   )
683   )
684   )
685   )
686   )
687   )
688   )
689   )
690   )
691   )
692   )
693   )
694   )
695   )
696   )
697   )
698   )
699   )
700   )
701   )
702   )
703   )
704   )
705   )
706   )
707   )
708   )
709   )
710   )
711   )
712   )
713   )
714   )
715   )
716   )
717   )
718   )
719   )
720   )
721   )
722   )
723   )
724   )
725   )
726   )
727   )
728   )
729   )
730   )
731   )
732   )
733   )
734   )
735   )
736   )
737   )
738   )
739   )
740   )
741   )
742   )
743   )
744   )
745   )
746   )
747   )
748   )
749   )
750   )
751   )
752   )
753   )
754   )
755   )
756   )
757   )
758   )
759   )
760   )
761   )
762   )
763   )
764   )
765   )
766   )
767   )
768   )
769   )
770   )
771   )
772   )
773   )
774   )
775   )
776   )
777   )
778   )
779   )
780   )
781   )
782   )
783   )
784   )
785   )
786   )
787   )
788   )
789   )
790   )
791   )
792   )
793   )
794   )
795   )
796   )
797   )
798   )
799   )
800   )
801   )
802   )
803   )
804   )
805   )
806   )
807   )
808   )
809   )
810   )
811   )
812   )
813   )
814   )
815   )
816   )
817   )
818   )
819   )
820   )
821   )
822   )
823   )
824   )
825   )
826   )
827   )
828   )
829   )
830   )
831   )
832   )
833   )
834   )
835   )
836   )
837   )
838   )
839   )
840   )
841   )
842   )
843   )
844   )
845   )
846   )
847   )
848   )
849   )
850   )
851   )
852   )
853   )
854   )
855   )
856   )
857   )
858   )
859   )
860   )
861   )
862   )
863   )
864   )
865   )
```

Now we can replace the originally-pasted code with our helper function, and we can create the symmetric case where Bob deposits first as an option to the outermost *When*.



```
1 {-# LANGUAGE OverloadedStrings #-}
2 module Example where
3
4 import Language.Marlowe.Extended
5
6 main :: IO ()
7 main = print . pretty $ contract
8
9
10 {- Define a contract, Close is the simplest contract which just ends the contract straight away
11 -}
12
13 alice, bob, charlie :: Party
14 alice = "Alice"
15 bob = "Bob"
16 charlie = "Charlie"
17
18 deposit :: Value
19 deposit = Constant 10
20
21 choiceId :: ChoiceId
22 choiceId = ChoiceId "winner" charlie
23
24 contract :: Contract
25 contract = When
26   [ f alice bob
27   , f bob alice
28   ]
29   10 Close
30   where
31     f :: Party -> Party -> Case
32     f x y = Case
33       (Deposit
34         x
35         x
36         (Taken ** **))
37         deposit
38       )
39       When
40         (Case
41           (Deposit
42             y
43             y
44             )
45           )
46         )
```

This should compile and we can now send to the simulator.

Now we have two possible actions that can happen in the first step. Alice can deposit 10, or Bob can deposit 10.

If Bob starts...

Then now it is Alice's turn.

So, basically, to use the Haskell editor, we write a program that produces something of type *Contract* and you can use all the features of Haskell like local functions or whatever to make your life easier.


When using Blockly, if we had wanted to give Bob the option of being the first to deposit, we would have had no choice but to have copy and pasted the whole *When* construct.

Of course, there are other options when using Haskell. We could also parameterise the contract, for example, we could pass in the deposit value as an argument.

We could also parameterise the parties and even generalise it so that the number of parties could be variable. This would be very inconvenient if we were to have to do this by hand using Blockly, but in Haskell it is quite straightforward.

What is noteworthy here is that Marlowe, in contrast to Plutus, is very simply Haskell. The Marlowe team made a point of using only basic Haskell features. You don't need lenses, you don't need Template Haskell, you don't even need monads or type-level programming.

Marlowe is not always appropriate because it is specifically for financial contracts, but if it is appropriate then it is a very nice option due to all the safety assurances that Simon mentioned and because it is much simpler and easy to get right than Plutus.



MARLOWE PLAYGROUND

New Project +

Tutorials

Actus Labs

New Project

Open

Open Example

Rename

Save

Save As...

```

1 When
2   [Case
3     (Deposit
4       (Role "Alice")
5       (Role "Alice")
6       (Token "" ""))
7       (Constant 10)
8     )
9   ]
10  [When
11    [Case
12      (Deposit
13        (Role "Bob")
14        (Role "Bob")
15        (Token "" ""))
16        (Constant 10)
17      )
18      [When
19        [Case
20          (ChoiceId
21            "winner"
22            (Role "Charlie")
23          )
24          [Bound 1 2]
25        ]
26      ]
27      [If
28        (ValueEq
29          (ChoiceValue
30            (ChoiceId
31              "winner"
32              (Role "Charlie")
33            )
34            (Constant 1)
35          )
36          (Pay
37            (Role "Bob")
38            (Account (Role "Alice"))
39            (Token "" ""))
40            (Constant 10)
41          )
42          (Pay
43            (Role "Alice")
44            (Account (Role "Bob"))
45            (Token "" ""))
46            (Constant 10)
47          )
48        ]
49      ]
50    ]
51  ]

```

Current State

current slot: 0

expiration slot: 30

ACTIONS

Participant **Alice**

Deposit 10 units of ADA into account of Alice as Alice

+

Participant **Bob**

Deposit 10 units of ADA into account of Bob as Bob

+

Other Actions

Move to slot 10

+

Undo

Reset

TRANSACTION LOG


Action

Slot

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter



MARLOWE PROJECT

New Project +

Tutorials

Actus Labs

New Project

Open

Open Example

Rename

Save

Save As...

```

1 When
2   [Case
3     (Deposit
4       (Role "Alice")
5       (Role "Alice")
6       (Token "" ""))
7     (Constant 10)
8   )
9   [When
10    [Case
11      (Deposit
12        (Role "Bob")
13        (Role "Bob")
14        (Token "" ""))
15      (Constant 10)
16    ]
17    [When
18      [Case
19        (Choice
20          (ChoiceId
21            "winner"
22            (Role "Charlie"))
23          )
24        [Bound 1 2]
25      ]
26      [If
27        (ValueEq
28          (ChoiceValue
29            (ChoiceId
30              "winner"
31              (Role "Charlie"))
32            )
33          (Constant 1)
34        )
35        [Pay
36          (Role "Bob")
37          (Account (Role "Alice"))
38          (Token "" "")
39          (Constant 10)
40          Close
41        ]
42        [Pay
43          (Role "Alice")
44          (Account (Role "Bob"))
45          (Token "" "")
46          (Constant 10)
47          Close
48        ]
49      ]
50    ]
51  ]
52 ]

```

Current State

current slot: 0

expiration slot: 30

ACTIONS

Participant **Alice**

Deposit 10 units of ADA into account of Alice as Alice

+

Participant **Bob**

Deposit 10 units of ADA into account of Bob as Bob

+

Other Actions

Move to slot 10

+

Undo

Reset

TRANSACTION LOG

Action

Slot

cardano.org | iohk.io

© 2020 IOHK Ltd

Telegram | Twitter



## WEEK 10 - UNISWAP

---

**Note:** These is a written version of [Lecture #10](#).

In this lecture we look at an implementation of Uniswap in Plutus.

This is the last lecture in the Plutus Pioneer Program. However, there will be a special lecture once it is possible to deploy contracts to the testnet.

---

In this lecture we won't be introducing any new topics or concepts. Instead we will do an end-to-end walk through of a demo that Lars wrote some months ago that clones the very popular Uniswap contract from Ethereum.

The one new thing we will look at following several requests is how you can query the endpoints created by the PAB with Curl commands just from the console.

### 10.1 What is Uniswap

So for those of you who haven't heard of Uniswap, what is Uniswap?

Uniswap is a so-called DeFi, or decentralized finance application, that allows swapping of tokens without any central authority. In the case of Ethereum it's ERC20 tokens.

So you don't need a centralized exchange, the traditional way to exchange tokens or other crypto assets. Instead everything is governed by smart contracts and works fully automatically on the blockchain.

Another interesting feature of Uniswap is that it doesn't discover prices the usual way with the so-called order book, but uses a different automatic price discovery system. The idea is that people can create so-called liquidity pools.

If they want other users to be able to swap two different tokens, then somebody can create a liquidity pool and put a certain amount of those two tokens in this liquidity pool, and in return the creator of the pool will receive so-called liquidity tokens that are specific to this one pool.

Other users can use that pool to swap. They take some amount of one of the tokens out in exchange for putting an amount of the other token back in.

Additionally, people can also add liquidity to the pool and receive liquidity tokens, or they can also burn liquidity tokens in exchange for tokens from the pool.

And all these features are also implemented in the version of Uniswap that works on Cardano that we're going to look at now.

So let's look at the various operations that are available in turn.

It all starts by somebody setting up the whole system. So some organization or entity that wants to offer this Uniswap service.



It starts with a transaction that creates a UTxO at this script address, here we call that *factory* for Uniswap factory. It contains an NFT that identifies the factory, the same trick that we have used a couple of times before, and as datum, it will contain the list of all liquidity pools.

So in the beginning, when the factory is just being created, that list will be empty.

Now let's assume that one user, Alice wants to create a liquidity pool for tokens A and B. A pool that allows others to swap A against B or B against A.

She has to provide some initial liquidity for the pool. So she needs some amount of token A and some amount of token B, let's say she has 1,000A and 2000B.

It's important to note here that the ratio between A and B reflects Alice's belief in the relative value of the tokens. So if she wants to set up a pool with 1000A and 2000B, then she believes that one A has the same value as two Bs.

In order to create the liquidity pool, she will create a transaction with two inputs and three outputs.

The two inputs will be the liquidity she wants to provide; the 1000A and 2000B and the Uniswap factory invoked with the create redeemer. The three outputs will be the newly-created pool.

We call it *Pool AB* here to indicate that it contains tokens AB, which will contain the liquidity that Alice provided; the 1000A and the 2000B and a freshly-minted token that identifies this pool, an NFT, called *AB NFT* here.

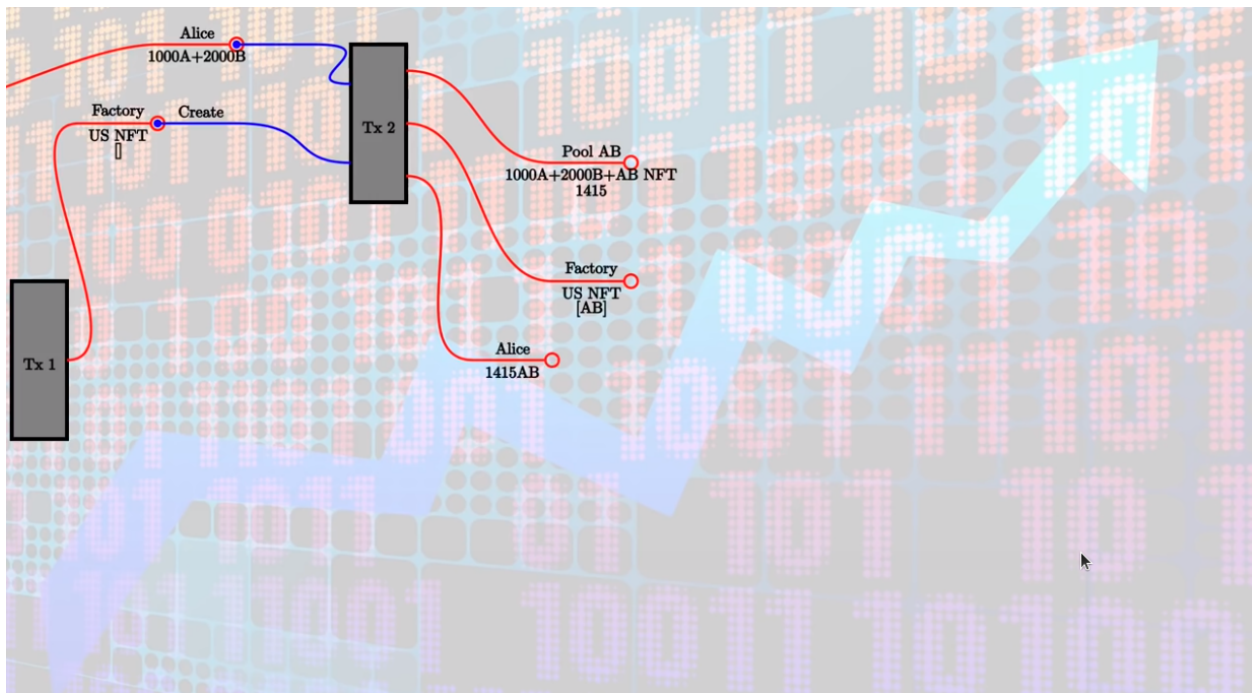
The datum of the pool, the 1415, will be the amount of liquidity tokens that Alice receives in return for setting up this pool and providing the liquidity. #

If you wonder about the number, that is the square root of the product of 1000 and 2000, so that's how the initial amount of liquidity tokens is calculated. It doesn't really matter, you could scale it arbitrarily, but that's the way Uniswap does it.

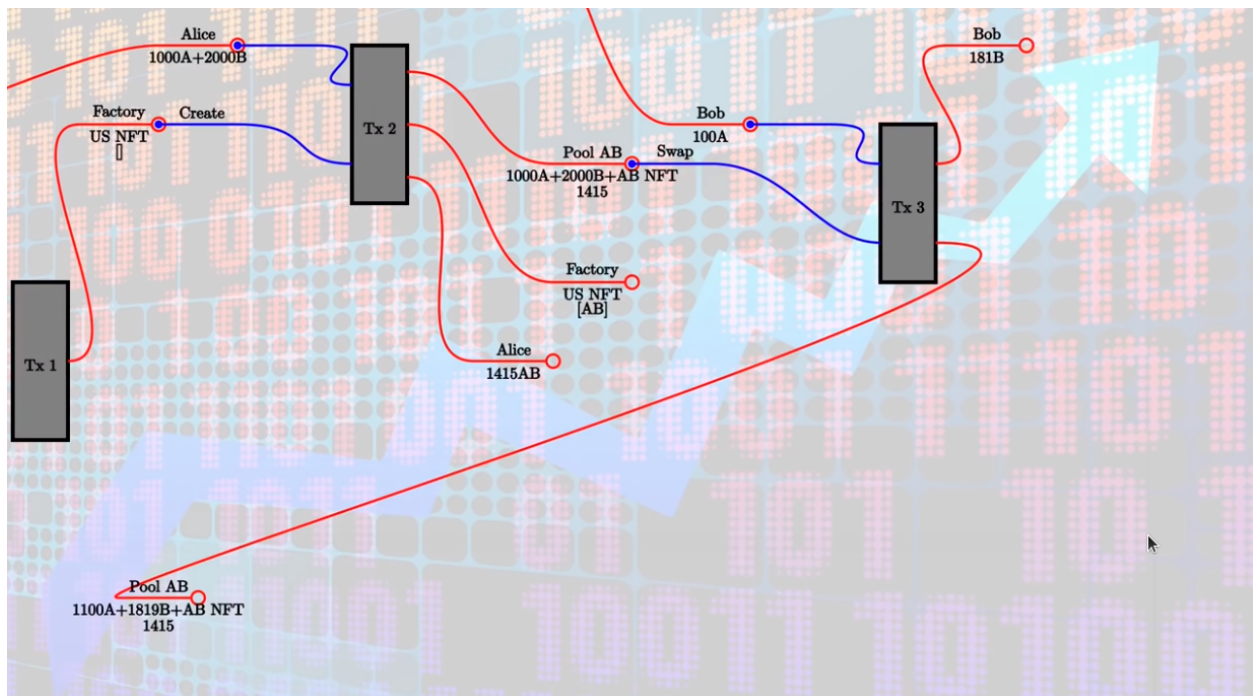
The second output is the Uniswap factory again, with the same NFT as before that identifies it. And now the datum has been updated. So in this list that was empty before, the list of all liquidity pools, there is now an entry for the newly-created AB pool.

Finally, there's a third output for Alice, where she receives the freshly-minted liquidity tokens, called *AB* here to indicate that they belong to the pool AB.





Now that the liquidity pool has been set up, other users can use it to swap.



So let's assume that Bob wants to swap 100A against B. What will Bob do?

He will create a transaction that has two inputs and two outputs. The two inputs are the 100A he wants to swap, and the pool with the swap redeemer. The outputs are the Bs he gets in return.

In this example, that would be 181B and the updated pool. So the pool now has the additional 100A that Bob provided. So now it's 1,100A, and it has 181B fewer than before.

It still, of course, has the NFT that identifies the pool and the datum hasn't changed because the amount of liquidity tokens that have been minted hasn't changed.

Now, of course, the question is, where does this 181 come from? This is this ingenious idea, how price discovery works in Uniswap.

So the rule is roughly that the product of the amounts of the two tokens must never decrease. Initially we have 1000 As and 2000 Bs and the product is 2 million.

If you do the calculation, then you will see that after the swap  $1100 \times 1819$  will be slightly larger than 2 million.

If you think about it or try a couple of examples by yourself, then you will see that in principle, you will always pay this ratio of the As and Bs in the pool, at least if you swap small amounts.

So originally the ratio from A to B was 1:2, 1000:2000. 100 is relatively small in comparison to the 1000 liquidity, so Bob should roughly get 200B, but he does get less and there are two reasons for that.

One is that the amount of tokens in the liquidity pool is never allowed to go to zero. And the more of one sort you take out, the more expensive it gets - the less you get in return. So 100 depletes the pool a bit of As, so Bob doesn't get the full factor 2 out, he gets a little bit less out. That's exactly how this product formula works.

This also makes it ingenious, because it automatically accounts for supply and demand. If the next person also wants to swap 100A, they would get even less out.

The idea is if a lot of people want to put A in and want to get B in return, that means the demand for B is high. And that means the price of B in relation to A should rise. And that is exactly what's happening.

So the more people do a swap in this direction, put A in and get B out, the less of the gap because the price of B rises. If there were swaps in the other direction, you would have the opposite effect.

If there's an equal amount of swaps from A to B and B to A, then this ratio between the two amounts would stay roughly the same.

There's an additional reason why Bob doesn't get the full 200 that he might expect, and that is fees.

We want to incentivize Alice to set up the pool in the first place. She won't just do that for fun, she wants to profit from it, so she wants to earn on swaps that people make.

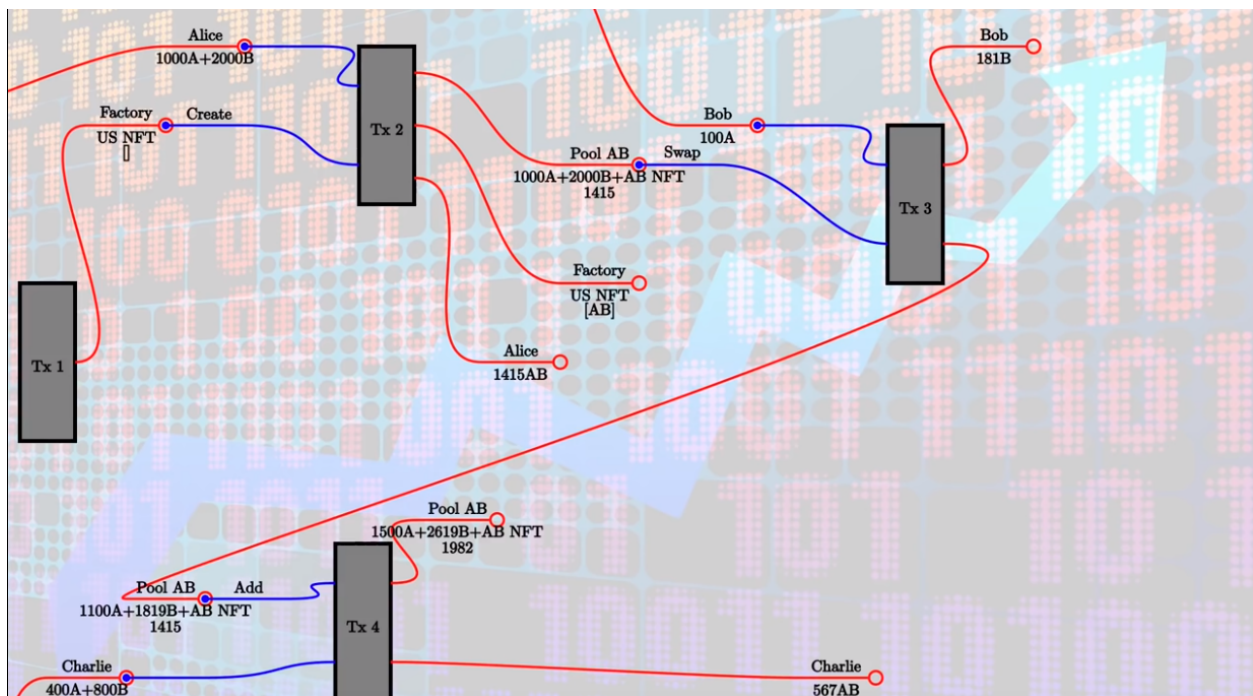
The original product formula is modified a bit to insist that the product doesn't only not decrease, but that it increases by a certain amount, a certain percentage, depending on how much people swap. That's 3% in this example of the 100A that Bob swaps, and it would be the same if you swap B instead.

This is basically added on top of this product, so anytime somebody swaps, not only does the product not decrease, it actually increases. And the more people swap, the more it increases.

The idea is that if Alice now would close the pool by burning her liquidity tokens, she gets all the remaining tokens in the pool and the product would be higher than what she originally put in.

So that's her incentive to set up the pool in the first place.

The next operation we look at is the add operation where somebody supplies the pool with additional liquidity.



So let's say that Charlie also believes that the ratio from A to B should be 1:2 and he wants to contribute 400A and 800B.

He could also have tokens in a different ratio; the ratio reflects his belief in the true relative value of the tokens.

So Charlie wants to add 400 As and 800 Bs, and he creates a transaction with two inputs and two outputs. The inputs are the pool and his contribution, his additional liquidity, and the outputs are the updated pool where now his As and Bs have been added to the pool tokens. Note that now the datum has changed.

So we had 1415 liquidity tokens before, and now we have 1982, and the difference, the 567, go to Charlie. So that's the second output of this transaction, and that's the reward to Charlie for providing this liquidity.

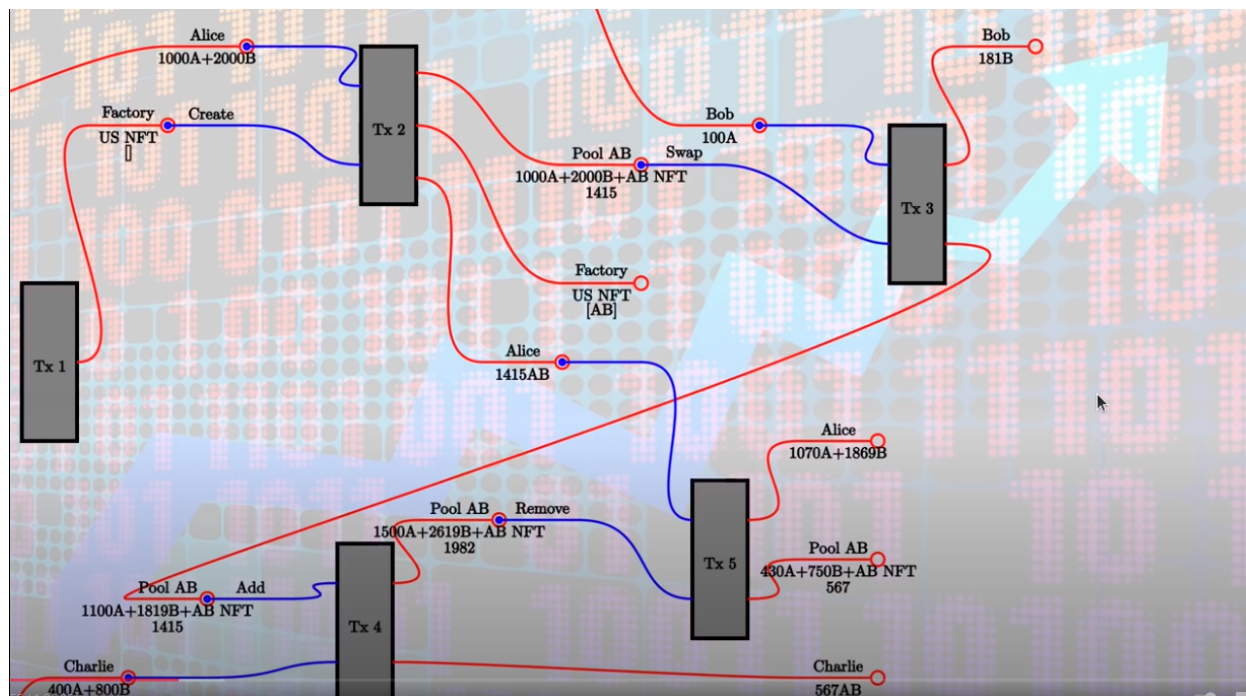


And there the formula is a bit complicated, but in principle, it also works with the product. So you check how much the product was before and after the tokens have been added and you take into account the number that have already been minted. That also ensures that now basically Alice profits from the fees that Bob paid with the swap and Charlie doesn't.

The specific formula doesn't matter. The idea is just that it's fair.

So people should receive liquidity tokens proportional to their contribution, but, if they only add liquidity after a couple of swaps have already happened, then they shouldn't profit from the fees that have accumulated in the meantime.

The next operation we look at is called *remove* and it allows owners of liquidity tokens for a pool to burn some of them.



So in this example, let's assume that Alice wants to burn all her liquidity tokens. She could also keep some, she doesn't have to burn all, but in this example, she wants to burn all her 1415 liquidity tokens.

So for that, she creates another transaction with two inputs and two outputs, the inputs are the liquidity token she wants to burn and, of course, the pool again with the *remove* redeemer.

The outputs are the tokens from the pool that she receives in return, so in this case, she would get 1078A and 1869B. The second output is the updated pool.

So the 1078A and 1869B have been removed from the pool and the datum has been updated, so the 1415 liquidity tokens that Alice burnt are now subtracted from the 1982 we had before. We see that 567 are remaining which are exactly those that Charlie owns.

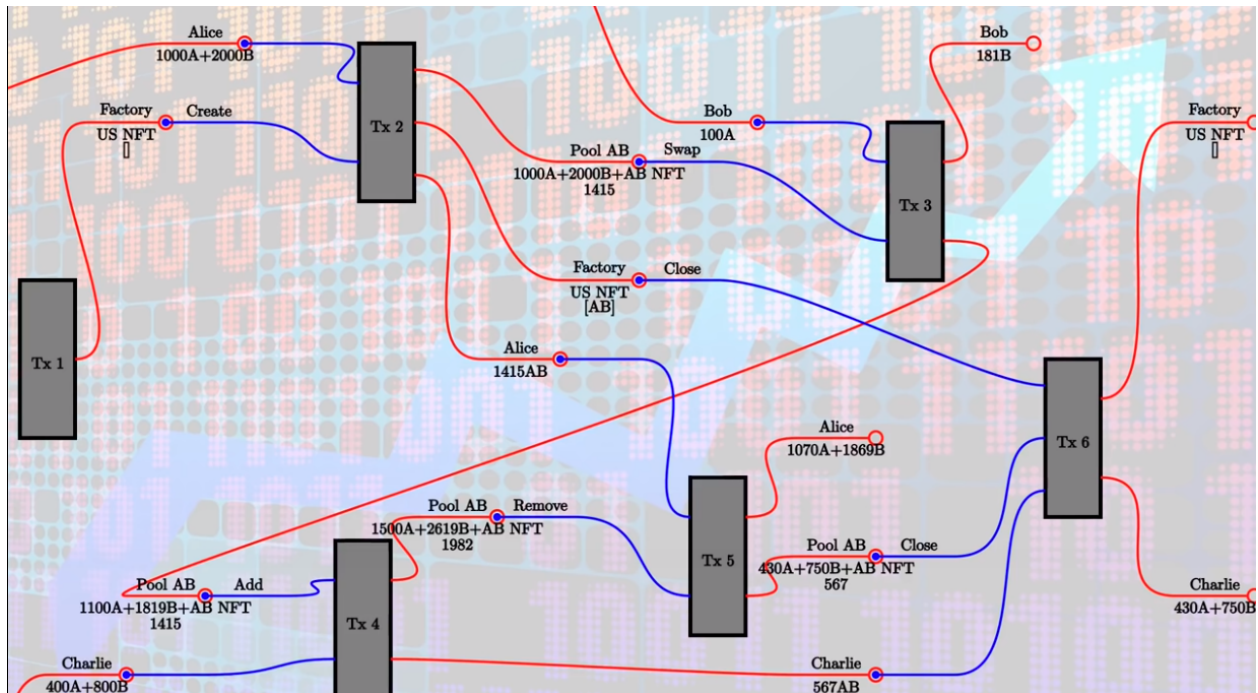
The formula for how many tokens Alice gets for burning liquidity tokens is again somewhat complicated, but it's basically just proportional.

So we know how many liquidity tokens there are in total, 1982, from the datum. And she basically just gets  $1415:1982$  of the pool. And she gets the tokens in the ratio that they are in now.

So the  $1072:1869$  should be the same ratio as the  $1500:2619$  which means that by burning, you don't change the ratio of the pool.

The last operation is *close* and it is for completely closing a pool and removing it.

This can only happen when the last remaining liquidity tokens are burnt.



So in our example, Charlie holds all the remaining 567 liquidity tokens and therefore he can close down the pool.

In order to do that, he creates a transaction with three inputs. One is the factory. Note that we only involve the factory when we create the pool and now when we close it again, which also means that the contention on the factory is not very high.

So the factory only gets involved when new pools are created and when pools are closed down, but once they exist and as long as they are not closed, the operations are independent of the factory.

We just need the factory when we want to update the list of existing pools, and by the way, this list is used to ensure that there won't be duplicate pools. So the create operation that we looked at in the beginning will fail if somebody tries to create a pool that already exists for a pair of tokens that already exist.

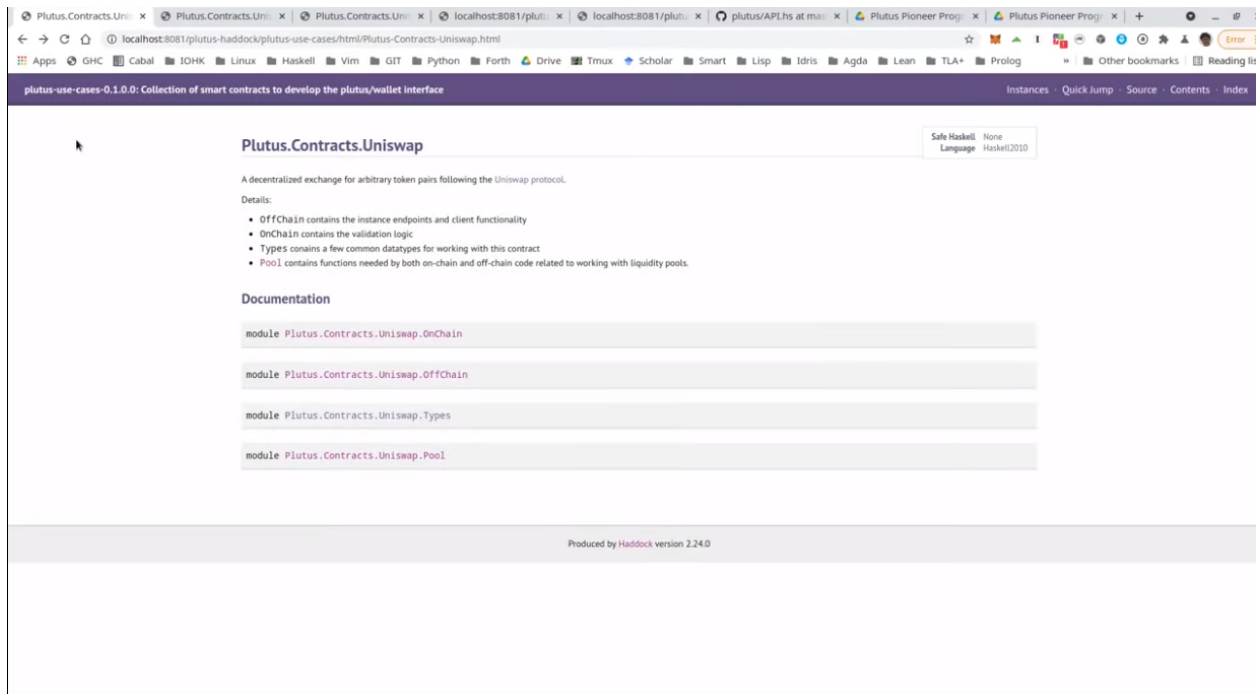
Okay, so let's go back to the *close* operation.

So the first input is the factory with the *close* redeemer, second the input is the pool that we want to close. And third input is all the remaining liquidity tokens.

We get two outputs, one is the updated factory. In this case we only had one pool, so the list only contains this one pool, and this is now removed from the list. The second output contains of all the remaining tokens, all the tokens that are still in the pool when it gets closed down.

So the remaining liquidity tokens are burnt and Charlie gets all the remaining tokens from the pool.

## 10.2 Uniswap in Plutus



Code for Uniswap is actually part of the Plutus repository and it is in the `plutus-usecases` library, split into four modules that are imported by the `Plutus.Contracts.Uniswap` module - `OnChain`, `OffChain`, `Types` and `Pool`.

So as the names suggest, `OnChain` contains the on-chain validation, `OffChain` contains the off-chain contracts, `Types` contains common types, and `Pool` contains the business logic, the calculations, how many liquidity tokens the creator of a pool gets, how many tokens you get when you add liquidity to a pool, how many tokens you get back when you burn liquidity tokens and under which conditions a swap is valid.

We won't go through all of that in too much detail. It contains nothing we haven't talked about before, but let's at least have a brief look.

So let's look at the `Types` module first.

`U` represents the Uniswap coin, the one that identifies the factory.

`A` and `B` are used for pool operations where we have these two sorts of tokens inside the pool.

`PoolState` is the token that identifies a pool, actually in the diagram earlier I said it's an NFT. By definition, an NFT is something that only exists once. Actually here in the implementation for each pool, an identical coin is created that identifies that pool. So it's not strictly speaking an NFT.

All the liquidity pools have one coin of that sort.

`Liquidity` is used for the liquidity tokens that the liquidity providers get.

And all these types are then used in the coin `A` type. So `A` is a type parameter, that's a so-called *phantom* type. So that means it has no representation at run time. It's just used to not mix up the various coins to make it easier to see what goes where, so in the datum, a coin is simply an asset class that we have seen before. Recall that `AssetClass` is a combination of currency symbol and token name.

Then `amount` is just a wrapper around integer that also contains such a phantom type parameter, so that we don't confuse amounts for token `A` and token `B`, for example.

The screenshot shows a web browser displaying the documentation for the Plutus Contracts Uniswap Types. The page title is "Documentation". It features two main sections: "data U" and "data A".

**data U** # Source

Uniswap coin token

Constructors

**U**

Instances

- `> IsData U` # Source
- `> Lift DefaultUni U` # Source
- `> Typeable DefaultUni U` # Source

**data A** # Source

A-side coin token

Constructors

**A**

Instances

The screenshot shows the same documentation page, but with the "data A" section expanded. It shows the "Constructors" and "Instances" for "A".

**data A** # Source

A-side coin token

Constructors

**A**

Instances

- `> IsData A` # Source
- `> Lift DefaultUni A` # Source
- `> Typeable DefaultUni A` # Source

**data B** # Source

B-side coin token

Constructors

**B**

Instances

- `> IsData B` # Source

The screenshot shows a web browser displaying the Plutus Pioneer Program documentation. The page is titled "Plutus-Contracts-Uniswap-Types.html". The main content area shows the definition of the `PoolState` data type. It includes a "Synopsis" sidebar on the right. The `PoolState` data type is defined as a coin token. It has a constructor `PoolState`. Under the "Instances" section, there are three instances listed: `IsData PoolState`, `Lift DefaultUni PoolState`, and `Typeable DefaultUni PoolState`. Below the `PoolState` section, the `Liquidity` data type is also shown, including its constructor and instances.

The screenshot shows the same web browser displaying the Plutus Pioneer Program documentation. The main content area now shows the definition of the `Liquidity` data type. It includes a "Synopsis" sidebar on the right. The `Liquidity` data type is defined as a coin token. It has a constructor `Liquidity`. Under the "Instances" section, there are three instances listed: `IsData Liquidity`, `Lift DefaultUni Liquidity`, and `Typeable DefaultUni Liquidity`. Below the `Liquidity` section, the `Coin` newtype is shown, defined as `newtype Coin a`. It includes a description: "A single `AssetClass`. Because we use three coins, we use a phantom type to track which one is which." and a constructor `Coin`.



Plutus.Contracts.Uniwap.Types

localhost:8081/plutus-haddock/plutus-use-cases/html/Plutus-Contracts-Uniswap-Types.html

Apps GHC Cabal IOHK Linux Haskell Vim GIT Python Forth Drive Tmux Scholar Smart Lisp Idris Agda Lean TLA+ Prolog Other bookmarks Reading list

Typeable DefaultUni Liquidity # Source

newtype Coin a # Source

A single `AssetClass`. Because we use three coins, we use a phantom type to track which one is which.

Constructors

Coin

unCoin :: AssetClass

Instances

- Lift DefaultUni (Coin a) # Source
- Eq (Coin a) # Source
- Ord (Coin a) # Source
- Show (Coin a) # Source
- Generic (Coin a) # Source
- ToJSON (Coin a) # Source
- FromJSON (Coin a) # Source
- ToSchema (Coin a) # Source

Plutus.Contracts.Uniwap.Types

localhost:8081/plutus-haddock/plutus-use-cases/html/Plutus-Contracts-Uniswap-Types.html

Apps GHC Cabal IOHK Linux Haskell Vim GIT Python Forth Drive Tmux Scholar Smart Lisp Idris Agda Lean TLA+ Prolog Other bookmarks Reading list

IsData a => IsData (Coin a) # Source

Eq (Coin a) # Source

Typeable DefaultUni Coin # Source

type Rep (Coin a) # Source

newtype Amount a # Source

Likewise for `Integer`; the corresponding amount we have of the particular Coin.

Constructors

Amount

unAmount :: Integer

Instances

- Lift DefaultUni (Amount a) # Source
- Eq (Amount a) # Source
- Num (Amount a) # Source
- Ord (Amount a) # Source
- Show (Amount a) # Source

```

valueOf :: Coin a -> Amount a -> Value # Source
unitValue :: Coin a -> Value # Source
isUnity :: Value -> Coin a -> Bool # Source
amountOf :: Value -> Coin a -> Amount a # Source
mkCoin :: CurrencySymbol -> TokenName -> Coin a # Source
newtype Uniswap # Source

Constructors
  Uniswap
    usCoin :: Coin U

Instances
  > Eq Uniswap # Source
  > Ord Uniswap # Source
  
```

Then we have some helper functions, for example *valueOf* for constructing a *Value* from *Coin* and *Amount*. Here, for example, we see the use of this phantom type.

That's actually a common trick in Haskell because now if you have, for example, pool operations that have two different coins and two different amounts for the different coins. And if the one is tagged with this type capital A and the other with capital B, then normally one could easily confuse them and somehow do operations with the one coin, with the amount for the other, and then make a mistake.

And here the type system enforces that we don't do that. So we can only use this value of function, for example, if we a coin and an amount with the same tag type tag.

So as I said, that's a common trick in Haskell, some lightweight type level programming that doesn't need any fancy GHC extensions.

The *unitValue* function creates one amount of the given coin and *isUnity* checks whether this coin is contained in the value exactly once,

Then *amountOf* checks how often the coin is contained in the value, and finally *mkCoin* turns a currency symbol into a token name, into a coin.

Then we have the Uniswap type which identifies the instance of the Uniswap system we are running. So of course, nobody can stop anybody from setting up a competing Uniswap system with the competing factory, but the value of this type identifies a specific system.

And all the operations that are specific to pool will be parameterized by a value of this type, but it's just a wrapper around the coin U. And that is just the NFT that identifies the factory.

Then we have a type for liquidity pools, and that is basically just two coins, the two coins in there.

However, there is one slight complication, only the two types of tokens inside the pool matter, not the order, there is no first or second token.

And in order to achieve that, the *Eq* instance has a special implementation. If we want to compare two liquidity pools, we don't just compare the first field with the first field of the other, and the second with the second, but we also try the other way round.

The screenshot shows the Plutus IDE interface with the `newtype Uniswap` definition. The `Constructors` section shows `Uniswap` with the constructor `usCoin :: Coin U`. The `Instances` section lists several instances for `Uniswap`, including `Eq`, `Ord`, `Show`, `Generic`, `ToJSON`, `FromJSON`, `ToSchema`, `IsData`, `Lift DefaultUni`, and `Typeable DefaultUni`.

```

newtype Uniswap
  Constructors
    Uniswap
      usCoin :: Coin U
  Instances
    > Eq Uniswap
    > Ord Uniswap
    > Show Uniswap
    > Generic Uniswap
    > ToJSON Uniswap
    > FromJSON Uniswap
    > ToSchema Uniswap
    > IsData Uniswap
    > Lift DefaultUni Uniswap
    > Typeable DefaultUni Uniswap
  
```

The screenshot shows the Plutus IDE interface with the definition of the `LiquidityPool` data type. The `Constructors` section shows `LiquidityPool` with constructors `lpCoinA :: Coin A` and `lpCoinB :: Coin B`. The `Instances` section lists several instances for `LiquidityPool`, including `Show`, `Generic`, `ToJSON`, `FromJSON`, `ToSchema`, `IsData`, and `Eq`.

```

data LiquidityPool
  Constructors
    LiquidityPool
      lpCoinA :: Coin A
      lpCoinB :: Coin B
  Instances
    > Show LiquidityPool
    > Generic LiquidityPool
    > ToJSON LiquidityPool
    > FromJSON LiquidityPool
    > ToSchema LiquidityPool
    > IsData LiquidityPool
    > Eq LiquidityPool
  
```

```

newtype Uniswap = Uniswap
{ usCoin :: Coin U
}
deriving stock (Haskell.Show, Generic)
deriving anyclass (ToJSON, FromJSON, ToSchema)
deriving newtype (Haskell.Eq, Haskell.Ord)
PlutusTx.makeIsDataIndexed 'Uniswap [( 'Uniswap, 0)]
PlutusTx.makeLift 'Uniswap

data LiquidityPool = LiquidityPool
{ lpCoinA :: Coin A
, lpCoinB :: Coin B
}
deriving (Haskell.Show, Generic, ToJSON, FromJSON, ToSchema)
PlutusTx.makeIsDataIndexed 'LiquidityPool [( 'LiquidityPool, 0)]
PlutusTx.makeLift 'LiquidityPool

instance Eq LiquidityPool where
{-# INLINABLE (==) #-}
x == y = (lpCoinA x == lpCoinA y && lpCoinB x == lpCoinB y) ||
-- Make sure the underlying coins aren't equal.
(unCoin (lpCoinA x) == unCoin (lpCoinB y) && unCoin (lpCoinB x) == unCoin (lpCoinA y))

data UniswapAction = Create LiquidityPool | Close | Swap | Remove | Add
deriving Haskell.Show
PlutusTx.makeIsDataIndexed 'UniswapAction [ ( 'Create, 0)
, ( 'Close, 1)
, ( 'Swap, 2)
, ( 'Remove, 3)
, ( 'Add, 4)
]
PlutusTx.makeLift 'UniswapAction

data UniswapDatum =
  Factory [LiquidityPool]
  | Pool LiquidityPool (Amount Liquidity)
deriving stock (Haskell.Show)
PlutusTx.makeIsDataIndexed 'UniswapDatum [ ( 'Factory, 0)
, ( 'Pool, 1)
]
PlutusTx.makeLift 'UniswapDatum

```

So liquidity pool tokens AB would be the same as liquidity pool with tokens BA. So that's the only slight complication here.

Then we define the actions, that's basically the redeemers. So *Create* with argument *LiquidityPool* is for creating a new liquidity pool, *Close* is for closing one, *Swap* is for swapping, *Remove* is for removing liquidity and *Add* is for adding liquidity.

Note that in the diagrams we saw earlier for simplicity, the redeemer was called simply *Create*. So I didn't mention this argument of type liquidity pool.

The datum is a bit more complex than we have seen before. It's not just a simple integer or similarly simple type, it's the type *UniswapDatum*.

There are two constructors, one for the factory and one for each pool. The factory will use the *Factory* constructor and the pool will use the *Pool* constructor.

And we saw before, the datum contains a list of all liquidity pools that currently exist. And the datum for *Pool* contains the *LiquidityPool* that we didn't see in the diagram. It also contains something we did see in the diagram, the amount of liquidity that has been minted for this pool. Remember that gets updated when somebody adds liquidity or removes liquidity.

Next let's look at the *Pool* module, which as I explained before, contains the business logic, the calculations.

So we have *calculateInitialLiquidity*. It gets the initial amount of token A and B that are put into the pool and returns the liquidity tokens that are returned in exchange for those.

Then *calculateAdditionalLiquidity* for the case that the pool already exists and somebody provides additional liquidity. So the first two arguments are the amount of token already in there. Then the third one is the liquidity tokens that have already been minted for the pool. And the next two arguments are how many As and Bs are added to the pool. The result is how many liquidity tokens will be minted in exchange for this additional amount.

The *calculateRemoval* function is for the opposite case. So given how many tokens are in the pool, how many liquidity tokens have been minted, how many liquidity tokens should be removed? It gives how many of tokens A and B remain in the pool.

Now *checkSwap* is arguably the central function of the whole Uniswap system. It calculates a swap.

The screenshot shows a web browser window displaying the Haddock documentation for the Uniswap module. The browser's address bar shows the URL: `localhost:8081/plutus-haddock/plutus-use-cases/html/Plutus-Contracts-Uniswap-Types.html`. The page content is organized into sections for each data type.

**data UniswapAction** # Source

**Constructors**

- Create LiquidityPool
- Close
- Swap
- Remove
- Add

**Instances**

- > Show UniswapAction # Source
- > IsData UniswapAction # Source
- > Lift DefaultUni UniswapAction # Source
- > Typeable DefaultUni UniswapAction # Source

**data UniswapDatum** # Source

**Constructors**

- Factory [LiquidityPool]
- Pool LiquidityPool (Amount Liquidity)

**Instances**

- > Show UniswapDatum # Source
- > IsData UniswapDatum # Source
- > Lift DefaultUni UniswapDatum # Source
- > Typeable DefaultUni UniswapDatum # Source

The bottom of the browser window shows the footer: "Produced by Haddock version 2.24.0".

```

-- | The initial liquidity is 'ceil( sqrt(x*y) )' where 'x' is the amount of
-- 'Coin A' and y the amount of 'Coin B'. See Eq. 13 of the Uniswap v2 paper.
calculateInitialLiquidity :: Amount A -> Amount B -> Amount Liquidity
calculateInitialLiquidity outA outB = Amount $ case isqrt (unAmount outA * unAmount outB) of
  Exactly 1
    | 1 > 0 -> 1
  Approximately 1
    | 1 > 0 -> 1 + 1
  _
    -> traceError "insufficient liquidity"

{-# INLINABLE calculateAdditionalLiquidity #-}
calculateAdditionalLiquidity :: Amount A -> Amount B -> Amount Liquidity -> Amount A -> Amount B -> Amount Liquidity
calculateAdditionalLiquidity oldA' oldB' liquidity delA' delB' =
  case rsqrt ratio of
    Imaginary    -> traceError "insufficient liquidity"
    Exactly x    -> Amount x - liquidity
    Approximately x -> Amount x - liquidity
  where
    ratio = (unAmount (liquidity * liquidity * newProd)) % unAmount oldProd

-- Unwrap, as we're combining terms
oldA = unAmount oldA'
oldB = unAmount oldB'
delA = unAmount delA'
delB = unAmount delB'

oldProd, newProd :: Amount Liquidity
oldProd = Amount $ oldA * oldB
newProd = Amount $ (oldA + delA) * (oldB + delB)

{-# INLINABLE calculateRemoval #-}
-- | See Definition 3 of <https://github.com/runtimeverification/verified-smart-contracts/blob/c40c98d6ae35148b76742aaaa29e6aa405b2f93/uniswap/x-y-k.pdf>.
calculateRemoval :: Amount A -> Amount B -> Amount Liquidity -> (Amount A, Amount B)
calculateRemoval inA inB liquidity' diff' = (f inA, f inB)
  where
    f :: Amount a -> Amount a
    f = Amount . g . unAmount

    diff = unAmount diff'
    liquidity = unAmount liquidity'

    g :: Integer -> Integer
    g x = x - divide (x * diff) liquidity

```

```

{-# INLINABLE calculateRemoval #-}
-- | See Definition 3 of <https://github.com/runtimeverification/verified-smart-contracts/blob/c40c98d6ae35148b76742aaaa29e6aa405b2f93/uniswap/x-y-k.pdf>.
calculateRemoval :: Amount A -> Amount B -> Amount Liquidity -> (Amount A, Amount B)
calculateRemoval inA inB liquidity' diff' = (f inA, f inB)
  where
    f :: Amount a -> Amount a
    f = Amount . g . unAmount

    diff = unAmount diff'
    liquidity = unAmount liquidity'

    g :: Integer -> Integer
    g x = x - divide (x * diff) liquidity

{-# INLINABLE checkSwap #-}
-- | A swap is valid if the fee is computed correctly, and we're swapping some
-- positive amount of A for B. See: <https://uniswap.org/whitepaper.pdf> Eq (11) (Page 7.)
checkSwap :: Amount A -> Amount B -> Amount A -> Amount B -> Bool
checkSwap oldA' oldB' newA' newB' =
  traceIfFalse "expected positive oldA" (oldA > 0) &&
  traceIfFalse "expected positive oldB" (oldB > 0) &&
  traceIfFalse "expected positive-newA" (newA > 0) &&
  traceIfFalse "expected positive-newB" (newB > 0) &&
  traceIfFalse "expected product to increase"
    (((newA * feeDen) - (inA * feeNum)) * ((newB * feeDen) - (inB * feeNum)))
    >= (feeDen * feeDen * oldA * oldB)
  where
    -- Unwrap; because we are mixing terms.
    oldA = unAmount oldA'
    oldB = unAmount oldB'
    newA = unAmount newA'
    newB = unAmount newB'

    inA = max 0 $ newA - oldA
    inB = max 0 $ newB - oldB
    -- The uniswap fee is 0.3%; here it is multiplied by 1000, so that the
    -- on-chain code deals only in integers.
    -- See: <https://uniswap.org/whitepaper.pdf> Eq (11) (Page 7.)
    feeNum, feeDen :: Integer
    feeNum = 3
    feeDen = 1000

```



This is how many As and Bs are originally in the pool and this says how many As and Bs there are after the swap in the pool. And it just returns whether that's okay or not.

So in principle, it just checks that the product of the last two arguments is larger than the product of the first two.

And we noted before, it's a bit more complicated because the fee is taken into account. So in this case, it's 0.3% and you can see this is taking into account here.

```

-- | Calculate the additional liquidity required for a swap.
-- | This function takes the current state of the pool and the swap details,
-- | and returns the additional liquidity required.
-- | The function is defined as follows:
-- |   calculateAdditionalLiquidity :: Integer -> Integer -> Integer -> Integer -> Integer
-- |   calculateAdditionalLiquidity oldA oldB newA newB =
-- |     traceIfFalse "expected positive oldA" (oldA > 0) &&
-- |     traceIfFalse "expected positive oldB" (oldB > 0) &&
-- |     traceIfFalse "expected positive newA" (newA > 0) &&
-- |     traceIfFalse "expected positive newB" (newB > 0) &&
-- |     traceIfFalse "expected product to increase"
-- |       (((newA * feeDen) - (inA * feeNum)) * ((newB * feeDen) - (inB * feeNum)))
-- |       >= (feeDen * feeDen * oldA * oldB)
-- |   where
-- |     -- Unwrap; because we are mixing terms.
-- |     oldA = unAmount oldA'
-- |     oldB = unAmount oldB'
-- |     newA = unAmount newA'
-- |     newB = unAmount newB'
-- |
-- |     inA = max 0 $ newA - oldA
-- |     inB = max 0 $ newB - oldB
-- |     -- The uniswap fee is 0.3%; here it is multiplied by 1000, so that the
-- |     -- on-chain code deals only in integers.
-- |     -- See: <https://uniswap.org/whitepaper.pdf> Eq (11) (Page 7.)
-- |     feeNum = 3
-- |     feeDen = 1000
-- |
-- |   {-# INLINABLE lpTicker #-}
-- |   -- | Generate a unique token name for this particular pool; based on the
-- |   -- | tokens it exchanges. This should be such that looking for a pool exchanging
-- |   -- | any two tokens always yields a unique name.
-- |   lpTicker :: LiquidityPool -> TokenName
-- |   lpTicker LiquidityPool{..} = TokenName hash
-- |   where
-- |     cA@(csA, tokA) = unAssetClass (unCoin lpCoinA)
-- |     cB@(csB, tokB) = unAssetClass (unCoin lpCoinB)
-- |     ((x1, y1), (x2, y2)) =
-- |       | cA < cB = ((csA, tokA), (csB, tokB))
-- |       | otherwise = ((csB, tokB), (csA, tokA))
-- |
-- |     h1 = sha2_256 $ unTokenName y1
-- |     h2 = sha2_256 $ unTokenName y2
-- |     h3 = sha2_256 $ unCurrencySymbol x1
-- |     h4 = sha2_256 $ unCurrencySymbol x2
-- |     hash = sha2_256 $ h1 <> h2 <> h3 <> h4

```

It also makes sure that none of the amounts ever drops to zero. So it's not allowed to remove all coins of one sort or of both from a pool. That also makes sense because of this product, if one of the factors was zero, then of course it couldn't be larger than it was before.

Finally, there's this *lpTicker* function. It's just a helper function that given a liquidity pool, computes a token name for the liquidity token. The idea here is that this token name should only depend on the liquidity pool and should be unique. So each pair of tokens should result in a unique token name. In principle it just takes the currency symbols and the token names of the two tokens or coins, concatenates all of them and hashes that, and then uses the hash of the concatenation to get something unique.

A slight complication here is that again we must make sure that the order of coins in the pool doesn't matter.

Now let's look at the on-chain part. Only two functions are exported.

First *mkUniswapValidator*, to make the validator for the Uniswap, both factory and pools, because they share the same script address. They are just distinguished by the datum and by the coins that identify them.

Then *validateLiquidityForging* which is the monetary policy script for the liquidity tokens, but there is a lot of code in this module and we don't want to go through it in detail, let's rather look at the structure.

So this is the *mkUniswapValidator* function. This function contains all the cases for factories and pools and the various redeemers.

And we have the function *validateLiquidityForging*, which is the monetary policy for liquidity tokens. The idea here is that it doesn't contain any logic and simply delegates the logic to the Uniswap validator. The way it does that is it checks the inputs of the forging transaction and checks that it either contains a factory or contains a pool, because if it does, then we know that this validator will run and then the validator can check that the forging is okay.

Plutus.Contracts.Uniswap.OnChain

Safe Haskell: None  
Language: Haskell2010

Documentation

```
mkUniswapValidator :: Uniswap -> Coin PoolState -> UniswapDatum -> UniswapAction ->
ScriptContext -> Bool # Source

validateLiquidityForging :: Uniswap -> TokenName -> ScriptContext -> Bool # Source
```

Produced by Haddock version 2.24.0

```
lc :: Coin Liquidity
lc = let AssetClass (cs, _) = unCoin c in mkCoin cs $ lpTicker lp

{-# INLINABLE findPoolDatum #-}
findPoolDatum :: TxInfo -> DatumHash -> (LiquidityPool, Amount Liquidity)
findPoolDatum info h = case findDatum h info of
  Just (Datum d) -> case PlutusTx.fromData d of
    Just (Pool lp a) -> (lp, a)
    _ -> traceError "error decoding data"
  _ -> traceError "pool input datum not found"

{-# INLINABLE mkUniswapValidator #-}
mkUniswapValidator :: Uniswap
-> Coin PoolState
-> UniswapDatum
-> UniswapAction
-> ScriptContext
-> Bool

mkUniswapValidator us c (Factory lps) (Create lp) ctx = validateCreate us c lps lp ctx
mkUniswapValidator _ c (Pool lp _) Swap ctx = validateSwap lp c ctx
mkUniswapValidator us c (Factory lps) Close ctx = validateCloseFactory us c lps ctx
mkUniswapValidator _ c (Pool _ _) Close ctx = validateClosePool us ctx
mkUniswapValidator _ c (Pool lp a) Remove ctx = validateRemove c lp a ctx
mkUniswapValidator _ c (Pool lp a) Add ctx = validateAdd c lp a ctx
mkUniswapValidator _ _ _ _ = False

{-# INLINABLE validateLiquidityForging #-}
validateLiquidityForging :: Uniswap -> TokenName -> ScriptContext -> Bool
validateLiquidityForging Uniswap{..} tn ctx
= case [
  1
  | i <- txInfoInputs $ scriptContextTxInfo ctx
  , let v = valueWithin i
  , isUnity v usCoin || isUnity v lpC
  ] of
  [] -> True
  [_] -> True
  [_', _] -> traceError "pool state forging without Uniswap input"
where
  lpC :: Coin Liquidity
  lpC = mkCoin (ownCurrencySymbol ctx) tn
```



The way it checks whether either the factory or pool is an input is via the coins that identify a factory or pool. So it checks whether this Uniswap factory coin is in the input or whether one of the pool coins is in the input.

And then we just have helper functions for all the various cases and they look quite long but it's all straightforward. It's basically what we saw in the diagram, just spelled out in detail so that all these conditions are satisfied for all the different cases.

```

{-# INLINABLE validateAdd #-}
-- | See 'Plutus.Contracts.Uniswap.OffChain.add'.
validateAdd :: Coin PoolState -> LiquidityPool -> Amount Liquidity -> ScriptContext -> Bool
validateAdd c lp liquidity ctx =
  traceIfFalse "pool stake token missing from input" (isUnit inVal c) &&
  traceIfFalse "output pool for same liquidity pair expected" (lp == fst outDatum) &&
  traceIfFalse "must not remove tokens" (delA >= 0 && delB >= 0) &&
  traceIfFalse "insufficient liquidity" (delL >= 0) &&
  traceIfFalse "wrong amount of liquidity tokens" (delL == calculateAdditionalLiquidity oldA oldB liquidity delA delB) &&
  traceIfFalse "wrong amount of liquidity tokens forged" (txInfoForge info == valueOf lC delL) &&
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    ownInput :: TxInInfo
    ownInput = findOwnInput' ctx

    ownOutput :: TxOut
    ownOutput = case [
      o
      | o <- getContinuingOutputs ctx
      , isUnit (txOutValue o) c
    ] of
      [o] -> o
      _ -> traceError "expected exactly on pool output"

    outDatum :: (LiquidityPool, Amount Liquidity)
    outDatum = case txOutDatum ownOutput of
      Nothing -> traceError "pool output datum hash not found"
      Just h -> findPoolDatum info h

    inVal, outVal :: Value
    inVal = valueWithin ownInput
    outVal = txOutValue ownOutput

    oldA = amountOf inVal aC
    oldB = amountOf inVal bC
    delA = amountOf outVal aC - oldA
    delB = amountOf outVal bC - oldB
    delL = snd outDatum - liquidity

    aC = lpCoinA lp
  
```

Finally, let's look at the off-chain code.

No surprises here, it's the usual boiler plate.

We define two different schemas. The idea is that one is for the entity that creates the Uniswap factory, and that only has one endpoint *start* and no parameters.

Then once that is created a second schema for people that make use of this Uniswap system, and all the contracts in here will be parameterized by the uniswap instance that the first action creates.

We make use of the state mechanism this monad writer mechanism that is accessible via *tell*, and basically for all the user operations, we have our own state, we call it *UserContractState*.

So there will be a helper contract that queries for all existing pools. So then the state would be using the *Pools* constructor and will return a list of pools in a simplified form - it's just a nested pair of pairs of coin and amount in each pool.

Now the helper function to query the existing funds of a wallet that will just return a value.

Then constructors for all the other operations. So if they have happened, then one of those will be the state. For example, if we did a swap, then afterwards the status will be updated to swapped. If we removed liquidity, it will be updated to removed and so on.

Then some names for the various tokens, so "Uniswap" will be the token name of the NFT in the Uniswap factory, "Pool State" will be the token name for the coins that identify the liquidity pools.

Then our usual boiler plate to actually get a script instance.

And the policy for the liquidity tokens.

Some various helper functions,

plutus-use-cases-0.1.0.0: Collection of smart contracts to develop the plutus/wallet interface

Instances · Quick Jump · Source · Contents · Index

## Plutus.Contracts.Uniswap.OffChain

Safe Haskell: None  
Language: Haskell2010

### Documentation

#### poolStateCoinFromUniswapCurrency

# Source

```

:: CurrencySymbol -- The currency identifying the Uniswap instance.
-> Coin PoolState

```

Gets the Coin used to identify liquidity pools.

#### liquidityCoin

# Source

```

:: CurrencySymbol -- The currency identifying the Uniswap instance.
-> Coin A -- One coin in the liquidity pair.
-> Coin B -- The other coin in the liquidity pair.
-> Coin Liquidity

```

Gets the liquidity token for a given liquidity pool.

#### data CreateParams

# Source

```

import Plutus.Contracts.Uniswap.Types
import qualified PlutusTx
import PlutusTx.Prelude
import Prelude
import Text.Printf (printf)

hiding (Semigroup (..), dropWhile, flip, unless)
as Haskell (Int, Semigroup (..), String, div, dropWhile, flip, show, (^))

data Uniswapping
instance Scripts.ScriptType Uniswapping where
  type instance RedeemerType Uniswapping = UniswapAction
  type instance DatumType Uniswapping = UniswapDatum

type UniswapOwnerSchema =
  BlockchainActions
  .\ Endpoint "start" ()

-- | Schema for the endpoints for users of Uniswap.
type UniswapUserSchema =
  BlockchainActions
  .\ Endpoint "create" CreateParams
  .\ Endpoint "swap" SwapParams
  .\ Endpoint "close" CloseParams
  .\ Endpoint "remove" RemoveParams
  .\ Endpoint "add" AddParams
  .\ Endpoint "pools" ()
  .\ Endpoint "funds" ()
  .\ Endpoint "stop" ()

-- | Type of the Uniswap user contract state.
data UserContractState =
  Pools [((Coin A, Amount A), (Coin B, Amount B))]
  | Funds Value
  | Created
  | Swapped
  | Added
  | Removed
  | Closed
  | Stopped
  deriving (Show, Generic, FromJSON, ToJSON)

```

```

./ Endpoint "funds" ()
./ Endpoint "stop" ()

-- | Type of the Uniswap user contract state.
data UserContractState =
  Pools [((Coin A, Amount A), (Coin B, Amount B))]
  Funds Value
  Created
  Swapped
  Added
  Removed
  Closed
  Stopped
  deriving (Show, Generic, FromJSON, ToJSON)

uniswapTokenName, poolStateTokenName :: TokenName
uniswapTokenName = "Uniswap"
poolStateTokenName = "Pool State"

uniswapInstance :: Uniswap -> Scripts.ScriptInstance Uniswapping
uniswapInstance us = Scripts.validator @Uniswapping
  ($$(PlutusTx.compile [| mkUniswapValidator |]))
  'PlutusTx.applyCode' PlutusTx.liftCode us
  'PlutusTx.applyCode' PlutusTx.liftCode c
  $$(PlutusTx.compile [| wrap |])
  where
    c :: Coin PoolState
    c = poolStateCoin us

    wrap = Scripts.wrapValidator @UniswapDatum @UniswapAction

uniswapScript :: Uniswap -> Validator
uniswapScript = Scripts.validatorScript . uniswapInstance

uniswapAddress :: Uniswap -> Ledger.Address
uniswapAddress = Ledger.scriptAddress . uniswapScript

uniswap :: CurrencySymbol -> Uniswap
uniswap cs = Uniswap $ mkCoin cs uniswapTokenName

```

```

deriving (Show, Generic, FromJSON, ToJSON)

uniswapTokenName, poolStateTokenName :: TokenName
uniswapTokenName = "Uniswap"
poolStateTokenName = "Pool State"

uniswapInstance :: Uniswap -> Scripts.ScriptInstance Uniswapping
uniswapInstance us = Scripts.validator @Uniswapping
  ($$(PlutusTx.compile [| mkUniswapValidator |]))
  'PlutusTx.applyCode' PlutusTx.liftCode us
  'PlutusTx.applyCode' PlutusTx.liftCode c
  $$(PlutusTx.compile [| wrap |])
  where
    c :: Coin PoolState
    c = poolStateCoin us

    wrap = Scripts.wrapValidator @UniswapDatum @UniswapAction

uniswapScript :: Uniswap -> Validator
uniswapScript = Scripts.validatorScript . uniswapInstance

uniswapAddress :: Uniswap -> Ledger.Address
uniswapAddress = Ledger.scriptAddress . uniswapScript

uniswap :: CurrencySymbol -> Uniswap
uniswap cs = Uniswap $ mkCoin cs uniswapTokenName

liquidityPolicy :: Uniswap -> MonetaryPolicy
liquidityPolicy us = mkMonetaryPolicyScript $
  $$(PlutusTx.compile [| \u t -> Scripts.wrapMonetaryPolicy (validateLiquidityForging u t) |])
  'PlutusTx.applyCode' PlutusTx.liftCode us
  'PlutusTx.applyCode' PlutusTx.liftCode poolStateTokenName

liquidityCurrency :: Uniswap -> CurrencySymbol
liquidityCurrency = scriptCurrencySymbol . liquidityPolicy

poolStateCoin :: Uniswap -> Coin PoolState
poolStateCoin = flip mkCoin poolStateTokenName . liquidityCurrency

-- | Gets the 'Coin' used to identify liquidity pools.

```

```

uniswap :: CurrencySymbol -> Uniswap
uniswap cs = Uniswap $ mkCoin cs uniswapTokenName

liquidityPolicy :: Uniswap -> MonetaryPolicy
liquidityPolicy us = mkMonetaryPolicyScript $
  $(PlutusTx.compile [| \u t -> Scripts.wrapMonetaryPolicy (validateLiquidityForging u t) |])
  'PlutusTx.applyCode' PlutusTx.liftCode us
  'PlutusTx.applyCode' PlutusTx.liftCode poolStateTokenName

liquidityCurrency :: Uniswap -> CurrencySymbol
liquidityCurrency = scriptCurrencySymbol . liquidityPolicy

poolStateCoin :: Uniswap -> Coin PoolState
poolStateCoin = flip mkCoin poolStateTokenName . liquidityCurrency

-- | Gets the 'Coin' used to identify liquidity pools.
poolStateCoinFromUniswapCurrency :: CurrencySymbol -- ^ The currency identifying the Uniswap instance.
--> Coin PoolState
poolStateCoinFromUniswapCurrency = poolStateCoin . uniswap

-- | Gets the liquidity token for a given liquidity pool.
liquidityCoin :: CurrencySymbol -- ^ The currency identifying the Uniswap instance.
--> Coin A -- ^ One coin in the liquidity pair.
--> Coin B -- ^ The other coin in the liquidity pair.
--> Coin liquidity
liquidityCoin cs coinA coinB = mkCoin (liquidityCurrency $ uniswap cs) $ lpTicker $ LiquidityPool coinA coinB

-- | Parameters for the @create@-endpoint, which creates a new liquidity pool.
data CreateParams = CreateParams
  { cpCoinA :: Coin A -- ^ One 'Coin' of the liquidity pair.
  , cpCoinB :: Coin B -- ^ The other 'Coin'.
  , cpAmountA :: Amount A -- ^ Amount of liquidity for the first 'Coin'.
  , cpAmountB :: Amount B -- ^ Amount of liquidity for the second 'Coin'.
  } deriving (Show, Generic, ToJSON, FromJSON, ToSchema)

-- | Parameters for the @swap@-endpoint, which allows swaps between the two different coins in a liquidity pool.
-- One of the provided amounts must be positive, the other must be zero.
data SwapParams = SwapParams
  { spCoinA :: Coin A -- ^ One 'Coin' of the liquidity pair.
  , spCoinB :: Coin B -- ^ The other 'Coin'.
  , spAmountA :: Amount A -- ^ The amount the first 'Coin' that should be swapped.
  , spAmountB :: Amount B -- ^ The amount of the second 'Coin' that should be swapped.
  } deriving (Show, Generic, ToJSON, FromJSON, ToSchema)

```

```

-- | Parameters for the @swap@-endpoint, which allows swaps between the two different coins in a liquidity pool.
-- One of the provided amounts must be positive, the other must be zero.
data SwapParams = SwapParams
  { spCoinA :: Coin A -- ^ One 'Coin' of the liquidity pair.
  , spCoinB :: Coin B -- ^ The other 'Coin'.
  , spAmountA :: Amount A -- ^ The amount the first 'Coin' that should be swapped.
  , spAmountB :: Amount B -- ^ The amount of the second 'Coin' that should be swapped.
  } deriving (Show, Generic, ToJSON, FromJSON, ToSchema)

-- | Parameters for the @close@-endpoint, which closes a liquidity pool.
data CloseParams = CloseParams
  { clpCoinA :: Coin A -- ^ One 'Coin' of the liquidity pair.
  , clpCoinB :: Coin B -- ^ The other 'Coin' of the liquidity pair.
  } deriving (Show, Generic, ToJSON, FromJSON, ToSchema)

-- | Parameters for the @remove@-endpoint, which removes some liquidity from a liquidity pool.
data RemoveParams = RemoveParams
  { rpCoinA :: Coin A -- ^ One 'Coin' of the liquidity pair.
  , rpCoinB :: Coin B -- ^ The other 'Coin' of the liquidity pair.
  , rpDiff :: Amount Liquidity -- ^ The amount of liquidity tokens to burn in exchange for liquidity from the pool.
  } deriving (Show, Generic, ToJSON, FromJSON, ToSchema)

-- | Parameters for the @add@-endpoint, which adds liquidity to a liquidity pool in exchange for liquidity tokens.
data AddParams = AddParams
  { apCoinA :: Coin A -- ^ One 'Coin' of the liquidity pair.
  , apCoinB :: Coin B -- ^ The other 'Coin' of the liquidity pair.
  , apAmountA :: Amount A -- ^ The amount of coins of the first kind to add to the pool.
  , apAmountB :: Amount B -- ^ The amount of coins of the second kind to add to the pool.
  } deriving (Show, Generic, ToJSON, FromJSON, ToSchema)

-- | Creates a Uniswap "factory". This factory will keep track of the existing liquidity pools and enforce that there will be at most one liquidity pool
-- for any pair of tokens at any given time.
start :: HasBlockchainActions => Contract w s Text Uniswap
start = do
  pkh <- pubKeyHash <$> ownPubKey
  cs <- fmap Currency.currencySymbol $
    mapError (pack . show @Currency.CurrencyError) $
      Currency.forgeContract pkh [(uniswapTokenName, 1)]
  let c = mkCoin cs uniswapTokenName
  us = uniswap cs

```

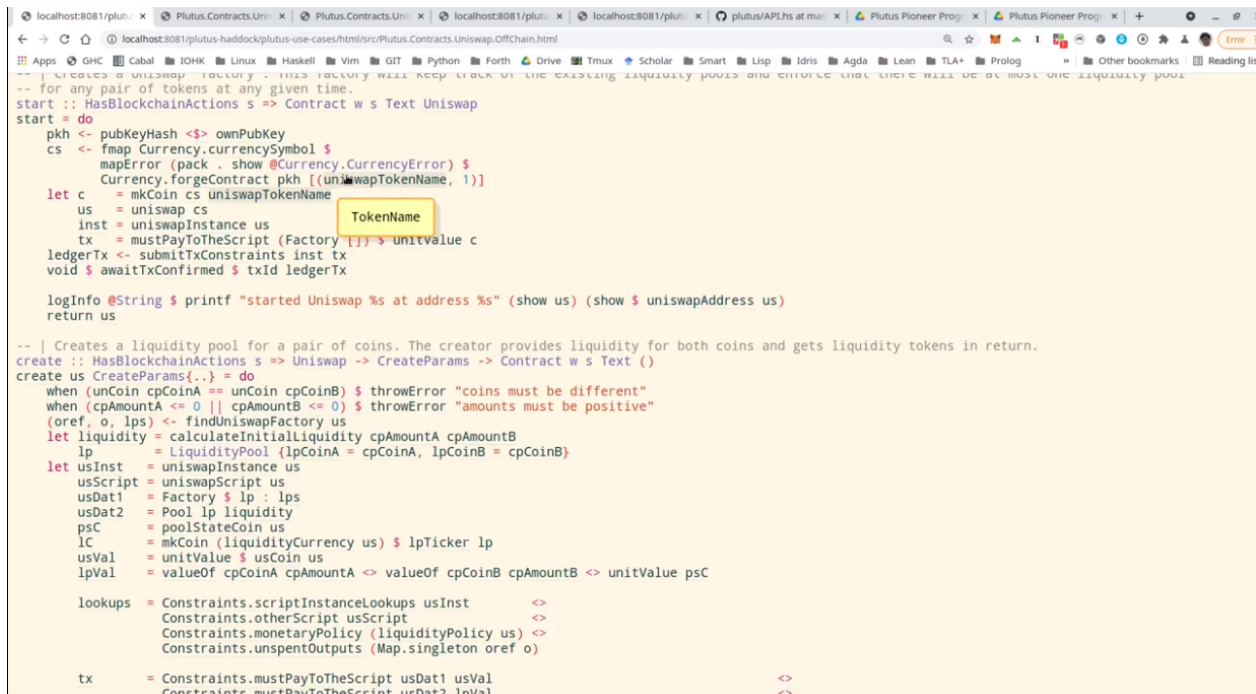
Then all the parameters for the endpoints. So, for example, if we want to create a pool we need to know the tokens and the amounts.

If you want to swap, it must know the tokens and how much to swap and the idea in the *SwapParams* datatype that one of the two last fields should be zero. So if you want to put in A and get out B, we've would specify the *spAmountA* for how many As we want to put in, but we would leave the *spAmountB* at zero, and the other way round if we want to swap Bs against As.

*CloseParams* for if you want to close a pool - we just have to specify which pool. So we give the two tokens that are in there.

*RemoveParams* - you have to specify the pool and how much liquidity we want to burn.

*AddParams* - again, identify the pool and how many As and how many Bs we want to add.



```

-- | Creates a Uniswap factory. This factory will keep track of the existing liquidity pools and enforce that there will be at most one liquidity pool
-- for any pair of tokens at any given time.
start :: HasBlockchainActions s => Contract w s Text Uniswap
start = do
  pkh <- pubKeyHash <$> ownPubKey
  cs <- fmap Currency.currencySymbol $
    mapError (pack . show @Currency.CurrencyError) $
      Currency.forgeContract pkh [(unswapTokenName, 1)]
  let c = mkCoin cs unswapTokenName
      us = unswap cs
      inst = unswapInstance us
      tx = mustPayToTheScript (Factory []) $ unitValue c
  ledgerTx <- submitTxConstraints inst tx
  void $ awaitTxConfirmed $ txId ledgerTx

  logInfo @String $ printf "started Uniswap %s at address %s" (show us) (show $ unswapAddress us)
  return us

-- | Creates a liquidity pool for a pair of coins. The creator provides liquidity for both coins and gets liquidity tokens in return.
create :: HasBlockchainActions s => Uniswap -> CreateParams -> Contract w s Text ()
create us CreateParams{..} = do
  when (unCoin cpCoinA == unCoin cpCoinB) $ throwError "coins must be different"
  when (cpAmountA <= 0 || cpAmountB <= 0) $ throwError "amounts must be positive"
  (oref, o, lps) <- findUniswapFactory us
  let liquidity = calculateInitialLiquidity cpAmountA cpAmountB
      lp = LiquidityPool {lpCoinA = cpCoinA, lpCoinB = cpCoinB}
  let usInst = unswapInstance us
      usScript = unswapScript us
      usDat1 = Factory $ lp : lps
      usDat2 = Pool lp liquidity
      psC = poolStateCoin us
      lc = mkCoin (liquidityCurrency us) $ lpTicker lp
      usVal = unitValue $ usCoin us
      lpVal = valueOf cpCoinA cpAmountA <> valueOf cpCoinB cpAmountB <> unitValue psC

  lookups = Constraints.scriptInstanceLookups usInst <>
    Constraints.otherScript usScript <>
    Constraints.monetaryPolicy (liquidityPolicy us) <>
    Constraints.unspentOutputs (Map.singleton oref o)

  tx = Constraints.mustPayToTheScript usDat1 usVal <>
    Constraints.mustPayToTheScript usDat2 lpVal <>

```

Now here we have the implementation.

So *start*, as we saw, sets up the whole system and it again makes use of this other use case we have used before, the *Currency.forgeContract* to mint the factory NFT that's then used to identify the Uniswap factory.

The *create* contract is the contract that creates a liquidity pool. We see all of these will be, as we mentioned before, identified by the Uniswap value, which is the result of this *start* contract here.

So we have *create*,

We have *close*, again parameterized by Uniswap,

*remove*,

*add*,

and *swap*.

All these functions also make use of the functions from the *Pools* module, that contain the business logic. So that will be used both in the validator, on the on-chain side, as well as on the off chain side in these contracts here.

The *pools* contract just queries the existing pools. So it looks for the factory UTxO and checks the datum, and, as we know, the datum of the factory contains the list of all pools.



```
-- | Creates a liquidity pool for a pair of coins. The creator provides liquidity for both coins and gets liquidity tokens in return.
create :: HasBlockchainActions s => Uniswap -> CreateParams -> Contract w s Text ()
create us CreateParams{..} = do
  when (unCoin cpCoinA == unCoin cpCoinB) $ throwError "coins must be different"
  when (cpAmountA <= 0 || cpAmountB <= 0) $ throwError "amounts must be positive"
  (oref, o, lps) <- findUniswapFactory us
  let liquidity = calculateInitialLiquidity cpAmountA cpAmountB
  lp = LiquidityPool {lpCoinA = cpCoinA, lpCoinB = cpCoinB}
  let usInst = uniswapInstance us
  usScript = uniswapScript us
  usDat1 = Factory $ lp : lps
  usDat2 = Pool lp liquidity
  psC = poolStateCoin us
  lC = mkCoin (liquidityCurrency us) $ lpTicker lp
  usVal = unitValue $ usCoin us
  lpVal = valueOf cpCoinA cpAmountA <+ valueOf cpCoinB cpAmountB <+ unitValue psC

  lookups = Constraints.scriptInstanceLookups usInst <+
    Constraints.otherScript usScript <+
    Constraints.monetaryPolicy (liquidityPolicy us) <+
    Constraints.unspentOutputs (Map.singleton oref o)

  tx = Constraints.mustPayToTheScript usDat1 usVal <+
    Constraints.mustPayToTheScript usDat2 lpVal <+
    Constraints.mustForgeValue (unitValue psC <+ valueOf lC liquidity) <+
    Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toData $ Create lp)

  ledgerTx <- submitTxConstraintsWith lookups tx
  void $ awaitTxConfirmed $ txId ledgerTx

  logInfo $ "created liquidity pool: " ++ show lp

-- | Closes a liquidity pool by burning all remaining liquidity tokens in exchange for all liquidity remaining in the pool.
close :: HasBlockchainActions s => Uniswap -> CloseParams -> Contract w s Text ()
close us CloseParams{..} = do
  ((oref1, o1, lps), (oref2, o2, lp, liquidity)) <- findUniswapFactoryAndPool us clpCoinA clpCoinB
  pkh <- pubKeyHash <$> ownPubKey
  let usInst = uniswapInstance us
  usScript = uniswapScript us
  usDat = Factory $ filter (/= lp) lps
  usC = usCoin us
```

```
logInfo $ "created liquidity pool: " ++ show lp

-- | Closes a liquidity pool by burning all remaining liquidity tokens in exchange for all liquidity remaining in the pool.
close :: HasBlockchainActions s => Uniswap -> CloseParams -> Contract w s Text ()
close us CloseParams{..} = do
  ((oref1, o1, lps), (oref2, o2, lp, liquidity)) <- findUniswapFactoryAndPool us clpCoinA clpCoinB
  pkh <- pubKeyHash <$> ownPubKey
  let usInst = uniswapInstance us
  usScript = uniswapScript us
  usDat = Factory $ filter (/= lp) lps
  usC = usCoin us
  psC = poolStateCoin us
  lC = mkCoin (liquidityCurrency us) $ lpTicker lp
  usVal = unitValue usC
  psVal = unitValue psC
  lVal = valueOf lC liquidity
  redeemer = Redeemer $ PlutusTx.toData Close

  lookups = Constraints.scriptInstanceLookups usInst <+
    Constraints.otherScript usScript <+
    Constraints.monetaryPolicy (liquidityPolicy us) <+
    Constraints.ownPubKeyHash pkh <+
    Constraints.unspentOutputs (Map.singleton oref1 o1 <+ Map.singleton oref2 o2)

  tx = Constraints.mustPayToTheScript usDat usVal <+
    Constraints.mustForgeValue (negate $ psVal <+ lVal) <+
    Constraints.mustSpendScriptOutput oref1 redeemer <+
    Constraints.mustSpendScriptOutput oref2 redeemer <+
    Constraints.mustIncludeDatum (Datum $ PlutusTx.toData $ Pool lp liquidity)

  ledgerTx <- submitTxConstraintsWith lookups tx
  void $ awaitTxConfirmed $ txId ledgerTx

  logInfo $ "closed liquidity pool: " ++ show lp

-- | Removes some liquidity from a liquidity pool in exchange for liquidity tokens.
remove :: HasBlockchainActions s => Uniswap -> RemoveParams -> Contract w s Text ()
remove us RemoveParams{..} = do
  (_, (oref, o, lp, liquidity)) <- findUniswapFactoryAndPool us rpCoinA rpCoinB
  pkh <- pubKeyHash <$> ownPubKey
  when (rpDiff < 1 || rpDiff >= liquidity) $ throwError "removed liquidity must be positive and less than total liquidity"
```



```

Constraints.mustForgeValue lVal
Constraints.mustSpendScriptOutput oreif redeemer

logInfo @String $ printf "val = %s, inVal = %s" (show val) (show inVal)
logInfo $ show lookups
logInfo $ show tx

ledgerTx <- submitTxConstraintsWith lookups tx
void $ awaitTxConfirmed $ txId ledgerTx

logInfo $ "added liquidity to pool: " ++ show lp

-- | Uses a liquidity pool two swap one sort of coins in the pool against the other.
swap :: HasBlockchainActions s => Uniswap -> SwapParams -> Contract w s Text ()
swap us SwapParams{..} = do
  unless (spAmountA > 0 && spAmountB == 0 || spAmountA == 0 && spAmountB > 0) $ throwError "exactly one amount must be positive"
  (., (oref, o, lp, liquidity)) <- findUniswapFactoryAndPool us spCoinA spCoinB
  let outVal = txOutValue $ txOutTxOut o
  let oldA = amountOf outVal spCoinA
  let oldB = amountOf outVal spCoinB
  (newA, newB) <- if spAmountA > 0 then do
    let outB = Amount $ findSwapA oldA oldB spAmountA
    when (outB == 0) $ throwError "no payout"
    return (oldA + spAmountA, oldB - outB)
  else do
    let outA = Amount $ findSwapB oldA oldB spAmountB
    when (outA == 0) $ throwError "no payout"
    return (oldA - outA, oldB + spAmountB)
  pkh <- pubKeyHash <$> ownPubKey
  logInfo @String $ printf "oldA = %d, oldB = %d, old product = %d, newA = %d, newB = %d, new product = %d" oldA oldB (unAmount oldA * unAmount oldB) newA newB

  let inst = uniswapInstance us
  val = valueOf spCoinA newA <> valueOf spCoinB newB <> unitValue (poolStateCoin us)

  lookups = Constraints.scriptInstanceLookups inst <>
    Constraints.otherScript (Scripts.validatorScript inst) <>
    Constraints.unspentOutputs (Map.singleton oreif o) <>
    Constraints.ownPubKeyHash pkh

  tx = mustSpendScriptOutput oreif (Redeemer $ PlutusTx.toData Swap) <>
    Constraints.mustPayToTheScript (PoolToLiquidity us)

```

```

logInfo $ "swapped with: " ++ show lp

-- | Finds all liquidity pools and their liquidity belonging to the Uniswap instance.
-- This merely inspects the blockchain and does not issue any transactions.
pools :: forall w s. HasBlockchainActions s => Uniswap -> Contract w s Text [((Coin A, Amount A), (Coin B, Amount B))]
pools us = do
  utxos <- utxoAt (uniswapAddress us)
  go $ snd <$> Map.toList utxos
  where
    go :: [TxOutTx] -> Contract w s Text [((Coin A, Amount A), (Coin B, Amount B))]
    go [] = return []
    go (o : os) = do
      let v = txOutValue $ txOutTxOut o
      if isUnity v c
      then do
        d <- getUniswapDatum o
        case d of
          Factory _ -> go os
          Pool lp _ -> do
            let coinA = lpCoinA lp
            let coinB = lpCoinB lp
            amtA = amountOf v coinA
            amtB = amountOf v coinB
            s = ((coinA, amtA), (coinB, amtB))
            logInfo $ "found pool: " ++ show s
            ss <- go os
            return $ s : ss
        else go os
      where
        c :: Coin PoolState
        c = poolStateCoin us

-- | Gets the caller's funds.
funds :: HasBlockchainActions s => Contract w s Text Value
funds = do
  pkh <- pubKeyHash <$> ownPubKey
  os <- map snd . Map.toList <$> utxoAt (pubKeyHashAddress pkh)
  return $ mconcat [txOutValue $ txOutTxOut o | o <- os]

getUniswapDatum :: TxOutTx -> Contract w s Text UniswapDatum
getUniswapDatum o = case txOutDatumHash $ txOutTxOut o of

```



```

ss <- go os
return $ s : ss

else go os

where
  c :: Coin PoolState
  c = poolStateCoin us

-- | Gets the caller's funds.
funds :: HasBlockchainActions s => Contract w s Text Value
funds = do
  pkh <- pubKeyHash <$> ownPubKey
  os <- map snd . Map.toList <$> utxoAt (pubKeyHashAddress pkh)
  return $ mconcat [txOutValue $ txOutTxOut o | o <- os]

getUniswapDatum :: TxOutTx -> Contract w s Text UniswapDatum
getUniswapDatum o = case txOutDatumHash $ txOutTxOut o of
  Nothing -> throwError "datumHash not found"
  Just h -> case Map.lookup h $ txData $ txOutTxTx o of
    Nothing -> throwError "datum not found"
    Just (Datum e) -> case PlutusTx.fromData e of
      Nothing -> throwError "datum has wrong type"
      Just d -> return d

findUniswapInstance :: HasBlockchainActions s => Uniswap -> Coin b -> (UniswapDatum -> Maybe a) -> Contract w s Text (TxOutRef, TxOutTx, a)
findUniswapInstance us c f = do
  let addr = uniswapAddress us
  logInfo @String $ printf "looking for Uniswap instance at address %s containing coin %s" (show addr) (show c)
  utxos <- utxoAt addr
  go [x | x@(_, o) <- Map.toList utxos, isUnit (txOutValue $ txOutTxOut o) c]
  where
    go [] = throwError "Uniswap instance not found"
    go ((oref, o) : xs) = do
      d <- getUniswapDatum o
      case f d of
        Nothing -> go xs
        Just a -> do
          logInfo @String $ printf "found Uniswap instance with datum: %s" (show d)
          return (oref, o, a)

findUniswapFactory :: HasBlockchainActions s => Uniswap -> Contract w s Text (TxOutRef, TxOutTx, [LiquidityPool])
findUniswapFactory us@Uniswap{..} = findUniswapInstance us usCoin $ \case
  Factory loc -> let loc

```

And finally *funds* just checks our own funds, the funds in the wallet, and returns them.

So these all return values but we want to write that in the state, and this is now done these endpoint definitions.

So first we have *ownerEndpoint* for setting up the whole system, which just uses the stop contract.

And then we have *userEndpoints*, which combine all these operations that a user can do.

Now there is no return value anymore, and instead we make use of the state. So we use the *Last* monoid again, so only the last *told* state will be kept.

And we also allow for error, so if there's an error in one of these contracts, then we will catch that error, but use a *Left* to write it in the state. If there was no error we write the appropriate user contract state value in the state with the *Right* constructor of *Either*.

Finally, we also have a *stop* endpoint that simply stops, it doesn't do anything. At any time you can invoke *stop* or one of the others, and if it was one of the others then recursively *userEndpoints* is called again, but in the case of *stop* not, so if the *stop* endpoint is ever called then the contract stops.

There are also tests for Uniswap contained in this Plutus use-cases library, but we won't look at them now.

Let's rather look at the Plutus PAB part and how you can write a front-end for Uniswap.

There is actually one also contained in the Plutus repo. It's in the *plutus-pab* library and in the *examples* folder there's a *Uniswap* folder that contains an example on how to do that.

This has been copied it into our Plutus Pioneer Program repo and slightly modified it to make it more suitable for our purposes.

When we look at the Cabal file for this week's code, there are two executables.

One *uniswap-pab*, which will run the PAB server, and then *uniswap-client*, which is a simple console based front-end for the Uniswap application.

You see, in the *other-modules* field there is a module *Uniswap* and that's listed in both. That contains some common definitions that are used by both parts.

So let's first look at that.

```

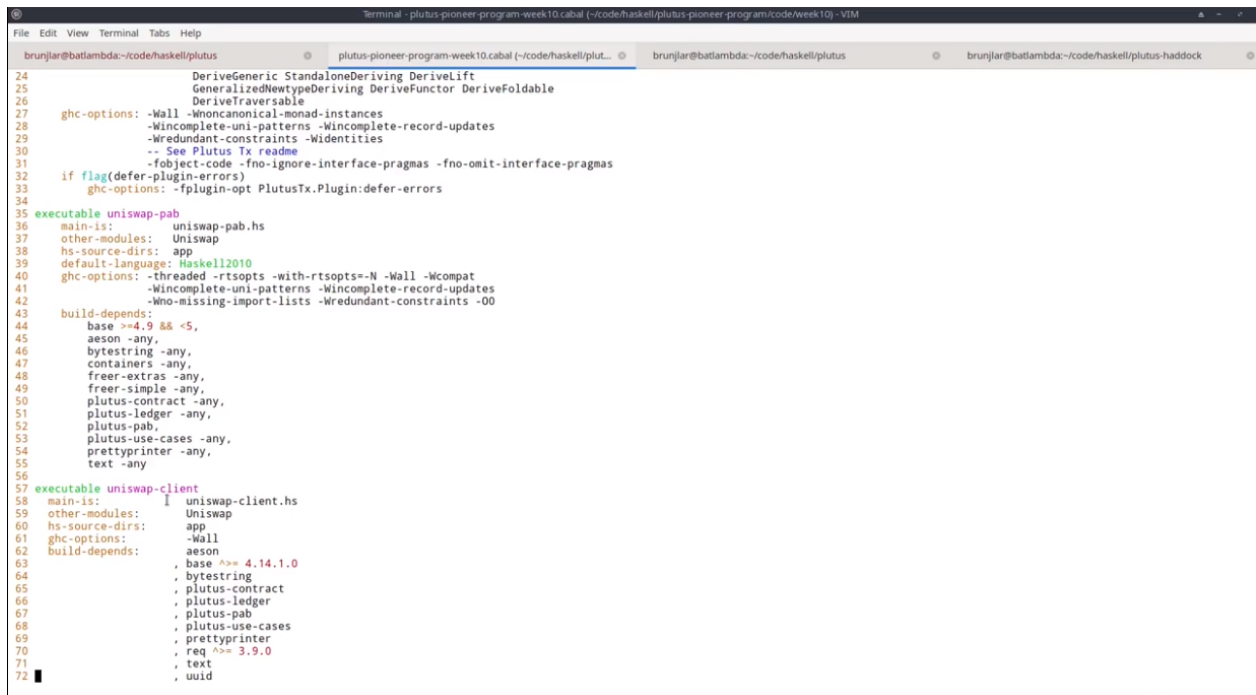
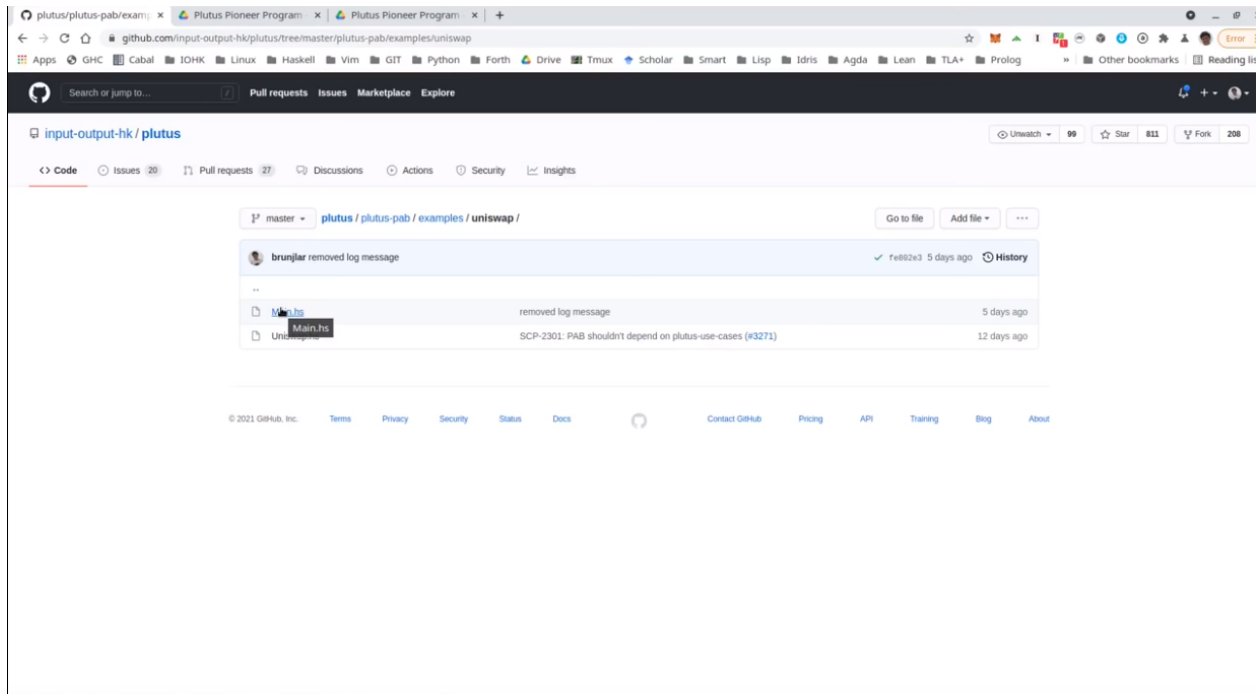
-- | Provides the following endpoints for users of a Uniswap instance:
--
--   [create@]: Creates a liquidity pool for a pair of coins. The creator provides liquidity for both coins and gets liquidity tokens in return.
--   [swap@]: Uses a liquidity pool two swap one sort of coins in the pool against the other.
--   [close@]: Closes a liquidity pool by burning all remaining liquidity tokens in exchange for all liquidity remaining in the pool.
--   [remove@]: Removes some liquidity from a liquidity pool in exchange for liquidity tokens.
--   [add@]: Adds some liquidity to an existing liquidity pool in exchange for newly minted liquidity tokens.
--   [pools@]: Finds all liquidity pools and their liquidity belonging to the Uniswap instance. This merely inspects the blockchain and does not issue any tr
--   [funds@]: Gets the caller's funds. This merely inspects the blockchain and does not issue any transactions.
--   [stop@]: Stops the contract.
userEndpoints :: Uniswap -> Contract (Last (Either Text UserContractState)) UniswapUserSchema Void ()
userEndpoints us =
  stop
  `select`
  ((f (Proxy @"create") (const Created) create          `select`
    f (Proxy @"swap") (const Swapped) swap              `select`
    f (Proxy @"close") (const Closed) close              `select`
    f (Proxy @"remove") (const Removed) remove          `select`
    f (Proxy @"add") (const Added) add                   `select`
    f (Proxy @"pools") Pools (\us' () -> pools us') `select`
    f (Proxy @"funds") Funds (\us () -> funds)) >> userEndpoints us)
  where
    f :: forall l a p.
      HasEndpoint l p UniswapUserSchema
    => Proxy l
    -> (a -> UserContractState)
    -> (Uniswap -> p -> Contract (Last (Either Text UserContractState)) UniswapUserSchema Text a)
    -> Contract (Last (Either Text UserContractState)) UniswapUserSchema Void ()
    f _ g c = do
      e <- runError $ do
        p <- endpoint @l
        c us p
        tell $ Last $ Just $ case e of
          Left err -> Left err
          Right us -> Right us

```

```

--   [create@]: Creates a liquidity pool for a pair of coins. The creator provides liquidity for both coins and gets liquidity tokens in return.
--   [swap@]: Uses a liquidity pool two swap one sort of coins in the pool against the other.
--   [close@]: Closes a liquidity pool by burning all remaining liquidity tokens in exchange for all liquidity remaining in the pool.
--   [remove@]: Removes some liquidity from a liquidity pool in exchange for liquidity tokens.
--   [add@]: Adds some liquidity to an existing liquidity pool in exchange for newly minted liquidity tokens.
--   [pools@]: Finds all liquidity pools and their liquidity belonging to the Uniswap instance. This merely inspects the blockchain and does not issue any tr
--   [funds@]: Gets the caller's funds. This merely inspects the blockchain and does not issue any transactions.
--   [stop@]: Stops the contract.
userEndpoints :: Uniswap -> Contract (Last (Either Text UserContractState)) UniswapUserSchema Void ()
userEndpoints us =
  stop
  `select`
  ((f (Proxy @"create") (const Created) create          `select`
    f (Proxy @"swap") (const Swapped) swap              `select`
    f (Proxy @"close") (const Closed) close              `select`
    f (Proxy @"remove") (const Removed) remove          `select`
    f (Proxy @"add") (const Added) add                   `select`
    f (Proxy @"pools") Pools (\us' () -> pools us') `select`
    f (Proxy @"funds") Funds (\us () -> funds)) >> userEndpoints us)
  where
    f :: forall l a p.
      HasEndpoint l p UniswapUserSchema
    => Proxy l
    -> (a -> UserContractState)
    -> (Uniswap -> p -> Contract (Last (Either Text UserContractState)) UniswapUserSchema Text a)
    -> Contract (Last (Either Text UserContractState)) UniswapUserSchema Void ()
    f _ g c = do
      e <- runError $ do
        p <- endpoint @l
        c us p
        tell $ Last $ Just $ case e of
          Left err -> Left err
          Right a -> Right $ g a
      stop :: Contract (Last (Either Text UserContractState)) UniswapUserSchema Void ()
      stop = do
        e <- runError $ endpoint @"stop"
        tell $ Last $ Just $ case e of
          Left err -> Left err
          Right () -> Right ()
        forall a w (s :: Row *) e.
          (HasEndpoint "stop" a s, AsContractError e) =>
            Contract w s e a

```



```

12
13 module Uniswap where
14
15 import Control.Monad           (forM_, when)
16 import Data.Aeson              (FromJSON, ToJSON)
17 import qualified Data.Semigroup as Semigroup
18 import Data.Text.Prettyprint.Doc (Pretty (..), viaShow)
19 import GHC.Generics             (Generic)
20 import Ledger
21 import Ledger.Constraints       as Value
22 import Ledger.Value             as Value
23 import Plutus.Contract          hiding (when)
24 import qualified Plutus.Contracts.Currency as Currency
25 import qualified Plutus.Contracts.Uniswap as Uniswap
26 import Wallet.Emulator.Types    (Wallet (..), walletPubKey)
27
28 data UniswapContracts =
29   Init
30   | UniswapStart
31   | UniswapUser Uniswap.Uniswap
32   deriving (Eq, Ord, Show, Generic)
33   deriving anyclass (FromJSON, ToJSON)
34
35 instance Pretty UniswapContracts where
36   pretty = viaShow
37
38 initContract :: Contract (Maybe (Semigroup.Last Currency.OneShotCurrency)) Currency.CurrencySchema Currency.CurrencyError ()
39 initContract = do
40   ownPK <- pubKeyHash <$> ownPubKey
41   let cs = Currency.forgeContract ownPK [(tn, fromIntegral (length wallets) * amount) | tn <- tokenNames]
42   let v = mconcat [Value.singleton cs tn amount | tn <- tokenNames]
43   forM_ wallets $ \w -> do
44     let pkh = pubKeyHash $ walletPubKey w
45     when (pkh /= ownPK) $ do
46       tx <- submitTx $ mustPayToPubKey pkh v
47       awaitTxConfirmed $ txId tx
48     tell $ Just $ Semigroup.Last cs
49   where
50     amount = 1000000
51
52 wallets :: [Wallet]
53 wallets = [Wallet i | i <- [1..4]]
54
55 tokenNames :: [TokenName]
56 tokenNames = ["A", "B", "C", "D"]
57
58 cidFile :: Wallet -> FilePath
59 cidFile w = "w" ++ show (getWallet w) ++ ".cid"
60
61 "app/Uniswap.hs" 60 lines --68%--

```

First of all, as we saw with oracle demo, we need some data type that captures the various instances we can run for the wallets. In this case, we have three.

*Init* hasn't been mentioned before, that has nothing specifically to do with Uniswap, this is just used to create some example tokens and distribute them in the beginning.

*UniswapStart* corresponds to the Uniswap start or Uniswap owner schema that we saw just now for setting up the whole system.

*UniswapUser* corresponds to the other part, to the various endpoints to interact with the system. And this class constructor is parameterized by a value of type *Uniswap*, which is the result of starting. So after having started the system, the result would be of type *Uniswap* and this is then needed to parameterize the client.

Then there is some boiler plate, then this *initContract* function that distributes the initial funds. So it again makes use of *forgeContract* that we have seen before.

And it now produces tokens with token names A, B, C, D with 1 million of each. Actually it also multiplies that by the number of wallets. So in this case, I want to use four wallets, wallets one to four, so it's actually 4 million of each of the tokens that will be forged.

And once they have been forged, they are sent from the forging wallet to all the other wallets. So one wallet forges four million of each, and then loops over the other wallets and sends them 1 million each.

So this is just needed to set up example tokens and distribute them amongst the wallets.

The *cidFile* function is just a helper function because in order to communicate the various contract instance IDs and other things we need, we use helper files and this is function gives the file name for a given wallet.

So now let's look at the PAB part.

First there is the boiler plate that we saw earlier to actually hook up the PAB mechanism with actual contracts. It uses the Uniswap contracts that we just defined with the three constructors *Init*, *UniswapStart* and *UniswapUser*.

So, *UniswapUser* user will use the *UniswapUser* schema that we defined before, *UniswapStart* will use the *UniswapOwner* schema that we defined before and *Init* will use a schema without endpoints.

```

27
28 data UniswapContracts =
29     Init
30     | UniswapStart
31     | UniswapUser Uniswap.Uniswap
32     deriving (Eq, Ord, Show, Generic)
33     deriving anyclass (FromJSON, ToJSON)
34
35 instance Pretty UniswapContracts where
36     pretty = viaShow
37
38 initContract :: Contract (Maybe (Semigroup.Last Currency.OneShotCurrency)) Currency.CurrencySchema Currency.CurrencyError ()
39 initContract = do
40     ownPK <- pubKeyHash <$> ownPubKey
41     cur <- Currency.forgeContract ownPK [(tn, fromIntegral (length wallets) * amount) | tn <- tokenNames]
42     let cs = Currency.currencySymbol cur
43     v = mconcat [Value.singleton cs tn amount | tn <- tokenNames]
44     forM_ wallets $ \w -> do
45         let pkh = pubKeyHash $ walletPubKey w
46         when (pkh /= ownPK) $ do
47             tx <- submitTx $ mustPayToPubKey pkh v
48             awaitTxConfirmed $ txId tx
49         tell $ Just $ Semigroup.Last cur
50     where
51         amount = 1000000
52
53 wallets :: [Wallet]
54 wallets = [Wallet i | i <- [1..4]]
55
56 tokenNames :: [TokenName]
57 tokenNames = ["A", "B", "C", "D"]
58
59 cidFile :: Wallet -> FilePath
60 cidFile w = "W" ++ show (getWallet w) ++ ".cid"

```

```

58     Success (Monoid.Last (Just (Right us))) -> Just us
59     -> Nothing
60     logString @(Builtin UniswapContracts) $ "Uniswap instance created: " ++ show us
61
62     forM_ wallets $ \w -> do
63         cid <- Simulator.activateContract w $ UniswapUser us
64         liftIO $ writeFile (cidFile w) $ show $ unContractInstanceId cid
65         logString @(Builtin UniswapContracts) $ "Uniswap user contract started for " ++ show w
66
67     void $ liftIO getLine
68
69     shutdown
70
71 handleUniswapContract ::
72     ( Member (Error PABError) effs
73     , Member (LogMsg (PABMultiAgentMsg (Builtin UniswapContracts))) effs
74     )
75     => ContractEffect (Builtin UniswapContracts)
76     -> Eff effs
77 handleUniswapContract = Builtin.handleBuiltin getSchema getContract where
78     getSchema = \case
79         UniswapUser _ -> Builtin.endpointsToSchemas @(Uniswap.UniswapUserSchema .\\ BlockchainActions)
80         UniswapStart -> Builtin.endpointsToSchemas @(Uniswap.UniswapOwnerSchema .\\ BlockchainActions)
81         Init -> Builtin.endpointsToSchemas @Empty
82     getContract = \case
83         UniswapUser us -> SomeBuiltin $ Uniswap.userEndpoints us
84         UniswapStart -> SomeBuiltin Uniswap.ownerEndpoint
85         Init -> SomeBuiltin US.initContract
86
87 handlers :: SimulatorEffectHandlers (Builtin UniswapContracts)
88 handlers =
89     Simulator.mkSimulatorHandlers @(Builtin UniswapContracts) []
90     $ interpret handleUniswapContract

```



And we connect these constructors with actual contracts. So *UniswapUser* with argument *us* will use the *userEndpoints* that we looked at earlier, *UniswapStart* will use the *ownerEndpoint*, and *Init* will use the *initContract* that we just defined, and that's just for demonstration to create these initial coins.

```
40 import Uniswap as US
41
42 main :: IO ()
43 main = void $ Simulator.runSimulationWith handlers $ do
44   logString @(Builtin UniswapContracts) "Starting Uniswap PAB webserver on port 8080. Press enter to exit."
45   shutdown <- PAB.Server.startServerDebug
46
47   cidInit <- Simulator.activateContract (Wallet 1) Init
48   cs <- flip Simulator.waitForState cidInit $ \json -> case fromJSON json of
49     Success (Just (Semigroup.Last cur)) -> Just $ Currency.currencySymbol cur
50     _ -> Nothing
51   - <- Simulator.waitUntilFinished cidInit
52
53   liftIO $ LB.writeFile "symbol.json" $ encode cs
54   logString @(Builtin UniswapContracts) $ "Initialization finished. Minted: " ++ show cs
55
56   cidStart <- Simulator.activateContract (Wallet 1) UniswapStart
57   us <- flip Simulator.waitForState cidStart $ \json -> case (fromJSON json :: Result (Monoid.Last (Either Text Uniswap.Uniswap))) of
58     Success (Monoid.Last (Just (Right us))) -> Just us
59     _ -> Nothing
60   logString @(Builtin UniswapContracts) $ "Uniswap instance created: " ++ show us
61
62   forM_ wallets $ \w -> do
63     cid <- Simulator.activateContract w $ UniswapUser us
64     liftIO $ writeFile (cidFile w) $ show $ unContractInstanceId cid
65     logString @(Builtin UniswapContracts) $ "Uniswap user contract started for " ++ show w
66
67   void $ liftIO getLine
68
69   shutdown
70
71 handleUniswapContract ::
72   Member (Error PABError) effs
```

Now we can look at the main program.

So in the *Simulator* monad, we execute certain things. First we set up the whole system, we start the server and get the handle to shut it down again.

The first thing is that Wallet 1 activates the *Init* contract. We know from looking at the code what that will do, it will mint all these example tokens, ABCD, 4 million of each, and then distribute them so that wallets one to four end up with 1 million of each of the four different tokens.

This will concurrently start this contract, but then immediately continue - it won't block - so we use the *waitForState* that we saw when we talked about oracles, to wait until *Init* returns.

What *Init* will do is that it will write the currency symbol of the forged example tokens into the state. So we wait until we see that and then we remember it and we wait until the *Init* contract has finished.

And then we write the currency symbol into the file *symbol.json*. We use the *encode* function from *Data.Aeson*, the json standard json library for Haskell.

Then again for Wallet 1, we start the Uniswap system. So we use the *UniswapStart* constructor and we again use *waitForState* to wait until we get the result. The result of the *UniswapStart* as we saw earlier will be a value of type *Uniswap*, and we need that value in order to parameterize the user contracts.

So we wait until we get this, and call it *us*, and now Uniswap, the system is running and now we can start the user instances for all the wallets.

So we loop over all wallets and activate the *UniswapUser* contract which is now parameterized by the *us* value we got in the previous step here.

Now we have these handles and in order to interact, to communicate, from the front-end with the server, we need these handles. So we write them into a file and this is where we use the helper function *cidFile* that we saw earlier.

So we will end up with four files *w1.cid* through to *w4.cid*, which contain these contract instance IDs for the four contracts.

Then we just wait until the user types a key and then we can shut down the server.

```

terminal - brunjar@batlabmda:~code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batlabmda:~code/haskell/plutus  uniswap-pab.hs (~code/haskell/plutus-pioneer-program/t...  brunjar@batlabmda:~code/haskell/plutus  brunjar@batlabmda:~code/haskell/plutus-haddock
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$ cabal run uniswap-pab
Up to date
[INFO] Slot 0: TxnValidate 75cb62c36a5fef382a3933edb2b93e67f05e3b0fe6362dd1ee17b94f6db215fa
[INFO] Starting Uniswap PAB webservice on port 8080. Press enter to exit.
[INFO] Starting PAB backend server on port: 8080
[INFO] Activated instance a8720a52-fbae-4839-8e41-9ddc32f17a05 on W1
[INFO] Slot 1: W1: Balancing an unbalanced transaction:
Tx:
Tx 4c37c89fab712bc8c4bc387fb0a2d95caa9cd3f587a79a6fcf7a7d1eb3e31d02:
{inputs:
collateral inputs:
outputs:
- Value (Map [(Map [("",1)])]) addressed to
addressed to ScriptCredential: c224b4a9f97ab768fd6d0cb6e5ec21c8be364e1d8987466db0bd2d3ee3971058 (no staking
credential)
forge: Value (Map [])
fee: Value (Map [(Map [("",10)])])
mps:
signatures:
validity range: Interval {ivFrom = LowerBound NegInf True, ivTo = UpperBound PosInf True}
data:
<>}
Requires signatures:
Utxo index:
[INFO] Slot 1: TxnValidate da806ba57684ef68c1e0e12bc50a6c38cfff38563259776ae8792387f9b3e8b3

```

Let's try this out with `cabal run uniswap-pab`.

A lot of stuff is happening. Remember, first we forge these example tokens ABCD, and then we need to distribute them to the other wallets.

Then we have to start the Uniswap system. And for that, we again have to first forge the Uniswap NFT that identifies the factory and then create the initial UTxO for the factory that contains an empty list of pools.

Now we see that all the *UniswapUser* contracts have started for each of the the four wallets.

If we look, we see the various files, so we can look at those.

So *symbol.json* is the currency symbol of the example tokens and we need that to refer to them.

And then we have these *w1.cid* - *w4.cid* files. If you look at one of those, they hold the contract instance IDs for the contract instances for the four wallets.

We need these in order to find the correct HTTP endpoints to communicate with them.

Let's look at the client next.

As for the oracle, this is also written that in Haskell using the same library for doing HTTP requests.

In the main program we expect one command line parameter, just a number from one to four, so that the main program knows for which wallet it's running.

Then we read the corresponding CID file to get the contract instance ID for that wallet. We read this *symbol.json* file to get the currency symbol of the example tokens. We use the *readFile* function the *ByteString* library, and *decode* comes from the *Data.Aeson* library to decode the json back to Haskell data type.

We check whether there was an error, and if not, we invoke the *go* function where we pass the contract instance ID and the currency symbol.

And then it's just a loop. We read a command from the console, and then depending on the command, we involve various helper functions. The commands exactly correspond to the endpoints we have, except for stop, which is not implemented.

```

credential) )
[INFO] Slot 56: TxnValidate cc8cf335a08dfd0d43c34cb85bbe4561eaa2112c694e166c2709b541127c087a
[INFO] Slot 65: W1: Balancing an unbalanced transaction:
Tx:
Tx defb0fe399b575bb4c63d12a1d9176b9667b587d27ea3834af223bae9eb520a2:
{inputs:
collateral inputs:
outputs:
- Value (Map [(2adddf64d97a95678d417ed616d9e698441718d772e2fcddec5fe62cf33856e96, Map [("Uniswap",1)])]) addressed to
addressed to ScriptCredential: 70647113cf77da57b10d601587d50b456a4af0b0ddc255b473260968e2a37890 (no staking credential)
forge: Value (Map [])
fee: Value (Map [(,Map [("",10)])])
scripts:
signatures:
validity range: Interval {ivFrom = LowerBound NegInf True, ivTo = UpperBound PosInf True}
data:
<[]>
Requires signatures:
Utxo index:
[INFO] Slot 65: TxnValidate 3f24da4175c48b2346329f00a0f1ee9c005bb645d228a3e1237baf79b2c64059
[INFO] b4969451-5784-41ea-b7e4-9f98758e7abe: "started Uniswap Uniswap {usCoin = Coin {unCoin = (2adddf64d97a95678d417ed616d9e698441718d772e2fcddec5fe62cf33856e96, \"Uniswap\")}} at address Address {addressCredential = ScriptCredential 70647113cf77da57b10d601587d50b456a4af0b0ddc255b473260968e2a37890, addressStakingCredential = Nothing}"
[INFO] Uniswap instance created: Uniswap {usCoin = Coin {unCoin = (2adddf64d97a95678d417ed616d9e698441718d772e2fcddec5fe62cf33856e96, \"Uniswap\")}}
[INFO] Activated instance 0d077eb7-470f-4c30-8619-40630cd01ab6 on W1
[INFO] Uniswap user contract started for Wallet 1
[INFO] Activated instance c9100b51-c02e-4afa-9e5d-5c374f90222a on W2
[INFO] Uniswap user contract started for Wallet 2
[INFO] Activated instance 6c9c6d05-433c-4e52-a750-0c1adbbfb405 on W3
[INFO] Uniswap user contract started for Wallet 3
[INFO] Activated instance b2c73f75-f968-4c15-800e-12c9af26d550 on W4
[INFO] Uniswap user contract started for Wallet 4

```

```

[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10] git add app/uniswap-client.hs
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10] git commit -S -m "logging requests and shell scripts to curl endpoints"
[week10 79e6ab5] logging requests and shell scripts to curl endpoints
10 files changed, 61 insertions(+), 1 deletion(-)
create mode 100755 code/week10/add.sh
create mode 100755 code/week10/close.sh
create mode 100755 code/week10/create.sh
create mode 100755 code/week10/funds.sh
create mode 100755 code/week10/logs.sh
create mode 100755 code/week10/pools.sh
create mode 100755 code/week10/remove.sh
create mode 100755 code/week10/status.sh
create mode 100755 code/week10/swap.sh
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10] git push teacher
Enumerating objects: 20, done.
Counting objects: 100% (20/20), done.
Delta compression using up to 8 threads
Compressing objects: 100% (15/15), done.
Writing objects: 100% (15/15), 2.39 KiB | 305.00 KiB/s, done.
Total 15 (delta 8), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (8/8), completed with 2 local objects.
To github.com:brunjar/plutus-pioneer-program.git
67daace..79e6ab5 week10 -> week10
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10] ls
add.sh  cabal.project  create.sh  funds.sh  LICENSE  plutus-pioneer-program-week10.cabal  remove.sh  swap.sh  W1.cid  W3.cid
app     close.sh       dist-newstyle  hie.yaml  logs.sh  pools.sh  status.sh  symbol.json  W2.cid  W4.cid
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10] cat symbol.json
{"unCurrencySymbol": "8a316f6dbbfc070ce13724269c2e10cfe367f6059af8316d84f7196f18d9346"}
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10] cat W1.cid
0d077eb7-470f-4c30-8619-40630cd01ab6

```



```

29 import Text.Read (readMaybe)
30 import Wallet.Emulator.Types (Wallet (..))
31
32 import Uniswap (cidFile, UniswapContracts)
33
34 main :: IO ()
35 main = do
36   w <- Wallet . read . head <$> getArgs
37   cid <- read <$> readFile (cidFile w)
38   mcs <- decode <$> LB.readFile "symbol.json"
39   case mcs of
40     Nothing -> putStrLn "invalid symbol.json" >> exitFailure
41     Just cs -> do
42       putStrLn $ "cid: " ++ show cid
43       putStrLn $ "symbol: " ++ show (cs :: CurrencySymbol)
44       go cid cs
45   where
46     go :: UUID -> CurrencySymbol -> IO a
47     go cid cs = do
48       cmd <- readCommandIO
49       case cmd of
50         Funds -> getFunds cid
51         Pools -> getPools cid
52         Create amtA tnA amtB tnB -> createPool cid $ toCreateParams cs amtA tnA amtB tnB
53         Add amtA tnA amtB tnB -> addLiquidity cid $ toAddParams cs amtA tnA amtB tnB
54         Remove amt tnA tnB -> removeLiquidity cid $ toRemoveParams cs amt tnA tnB
55         Close tnA tnB -> closePool cid $ toCloseParams cs tnA tnB
56         Swap amtA tnA tnB -> swap cid $ toSwapParams cs amtA tnA tnB
57       go cid cs
58
59 data Command =
60   Funds
61   | Pools
62   | Create Integer Char Integer Char

```

```

40   Nothing -> putStrLn "invalid symbol.json" >> exitFailure
41   Just cs -> do
42     putStrLn $ "cid: " ++ show cid
43     putStrLn $ "symbol: " ++ show (cs :: CurrencySymbol)
44     go cid cs
45   where
46     go :: UUID -> CurrencySymbol -> IO a
47     go cid cs = do
48       cmd <- readCommandIO
49       case cmd of
50         Funds -> getFunds cid
51         Pools -> getPools cid
52         Create amtA tnA amtB tnB -> createPool cid $ toCreateParams cs amtA tnA amtB tnB
53         Add amtA tnA amtB tnB -> addLiquidity cid $ toAddParams cs amtA tnA amtB tnB
54         Remove amt tnA tnB -> removeLiquidity cid $ toRemoveParams cs amt tnA tnB
55         Close tnA tnB -> closePool cid $ toCloseParams cs tnA tnB
56         Swap amtA tnA tnB -> swap cid $ toSwapParams cs amtA tnA tnB
57       go cid cs
58
59 data Command =
60   Funds
61   | Pools
62   | Create Integer Char Integer Char
63   | Add Integer Char Integer Char
64   | Remove Integer Char Char
65   | Close Char Char
66   | Swap Integer Char Char
67   deriving (Show, Read, Eq, Ord)
68
69 readCommandIO :: IO Command
70 readCommandIO = do
71   putStrLn "Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB"
72   s <- getLine

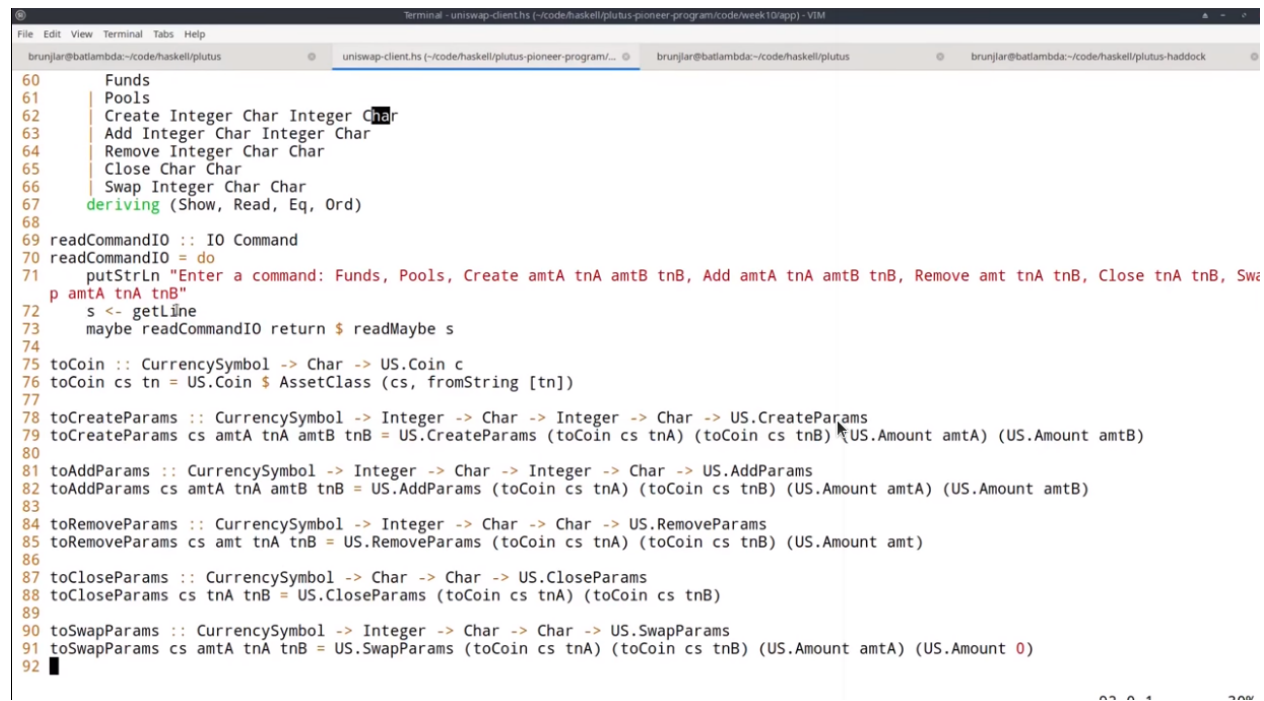
```

So we can query our funds, we can look for existing pools, we can create a pool, we can add liquidity to a pool, we can remove liquidity from a pool, we can close a pool, and we can swap, which is the whole point.

So for each of those, we have a constructor in the *Command* data type.

Because the currency symbol will always be *cs* - our example tokens, we don't need to parameterise the currency symbol. And because the token names are just A, B, C and D we can just use a character for that.

So for example, *Create Integer Character Integer Character* means create a liquidity pool with a certain amount of a given token and a certain amount of a second token.



```
60 Funds
61 Pools
62 Create Integer Char Integer Char
63 Add Integer Char Integer Char
64 Remove Integer Char Integer Char
65 Close Integer Char Integer Char
66 Swap Integer Char Integer Char
67 deriving (Show, Read, Eq, Ord)
68
69 readCommandIO :: IO Command
70 readCommandIO = do
71   putStrLn "Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amtA tnA amtB tnB, Close tnA tnB, Swap tnA tnA tnB"
72   s <- getLine
73   maybe readCommandIO return $ readMaybe s
74
75 toCoin :: CurrencySymbol -> Char -> US.Coin c
76 toCoin cs tn = US.Coin $ AssetClass (cs, fromString [tn])
77
78 toCreateParams :: CurrencySymbol -> Integer -> Char -> Integer -> Char -> US.CreateParams
79 toCreateParams cs amtA tnA amtB tnB = US.CreateParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amtA) (US.Amount amtB)
80
81 toAddParams :: CurrencySymbol -> Integer -> Char -> Integer -> Char -> US.AddParams
82 toAddParams cs amtA tnA amtB tnB = US.AddParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amtA) (US.Amount amtB)
83
84 toRemoveParams :: CurrencySymbol -> Integer -> Char -> Integer -> Char -> US.RemoveParams
85 toRemoveParams cs amtA tnA amtB tnB = US.RemoveParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amtA) (US.Amount amtB)
86
87 toCloseParams :: CurrencySymbol -> Integer -> Char -> Integer -> Char -> US.CloseParams
88 toCloseParams cs amtA tnA amtB tnB = US.CloseParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amtA) (US.Amount amtB)
89
90 toSwapParams :: CurrencySymbol -> Integer -> Char -> Integer -> Char -> US.SwapParams
91 toSwapParams cs amtA tnA amtB tnB = US.SwapParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amtA) (US.Amount amtB)
92
```

The *readCommandIO* function reads from the keyboard and then tries to pass that as a command. If it fails it will just recursively call the read command again. If it succeeds, it returns the command.

Then there are just various helper functions to convert something of type *Command* into the corresponding parameter types, like *CreateParams* or *AddParams* from the Uniswap module that we saw earlier.

The functions *showCoinHeader* and *showCoin* are just to make it look a bit prettier when we query the funds or the pools, and then we have the various endpoints and that all makes use of some helper functions.

There is *getStatus*, which we need in order to get something back from the contracts, and *callEndpoint* which uses the *Req* library, just as last time.

The interesting part is the request. It will be the post request, and we must give the instance ID. This is of type *UUID*, so we just convert it into a string and then pack it to a text because this HTTP library expects *Text*.

The request body depends on the third argument in the function.

The response will always be *Unit* and we just check whether we get a 200 status code or not.

The *getStatus* is a get request that invokes the *status* HTTP endpoint, again with the CID.

We have to tell it what we're dealing with so that's why we need the *UniswapContracts* type, and that's also why this Uniswap client executable also needs access to this Uniswap module.

And then we just check if the state is empty, which happens right in the beginning because that is before anything has told anything to the state. Then we wait a second and recurse and if there's a state (if it's *Just e*), then we know that this is of type *Either Text UserContractState*.

```

82 toAddParams cs amtA tnA amtB tnB = US.AddParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amtA) (US.Amount amtB)
83
84 toRemoveParams :: CurrencySymbol -> Integer -> Char -> Char -> US.RemoveParams
85 toRemoveParams cs amt tnA tnB = US.RemoveParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amt)
86
87 toCloseParams :: CurrencySymbol -> Char -> Char -> US.CloseParams
88 toCloseParams cs tnA tnB = US.CloseParams (toCoin cs tnA) (toCoin cs tnB)
89
90 toSwapParams :: CurrencySymbol -> Integer -> Char -> Char -> US.SwapParams
91 toSwapParams cs amtA tnA tnB = US.SwapParams (toCoin cs tnA) (toCoin cs tnB) (US.Amount amtA) (US.Amount 0)
92
93 showCoinHeader :: IO ()
94 showCoinHeader = printf "\n          token name          amount\n\n"          currency symbol
95
96 showCoin :: CurrencySymbol -> TokenName -> Integer -> IO ()
97 showCoin cs tn = printf "%64s %66s %15d\n" (show cs) (show tn)
98
99 getFunds :: UUID -> IO ()
100 getFunds cid = do
101   callEndpoint cid "funds" ()
102   threadDelay 2_000_000
103   go
104 where
105   go = do
106     e <- getStatus cid
107     case e of
108       Right (US.Funds v) -> showFunds v
109       _                  -> go
110
111 showFunds :: Value -> IO ()
112 showFunds v = do
113   showCoinHeader
114   forM_ (flattenValue v) \$ \(cs, tn, amt) -> showCoin cs tn amt

```

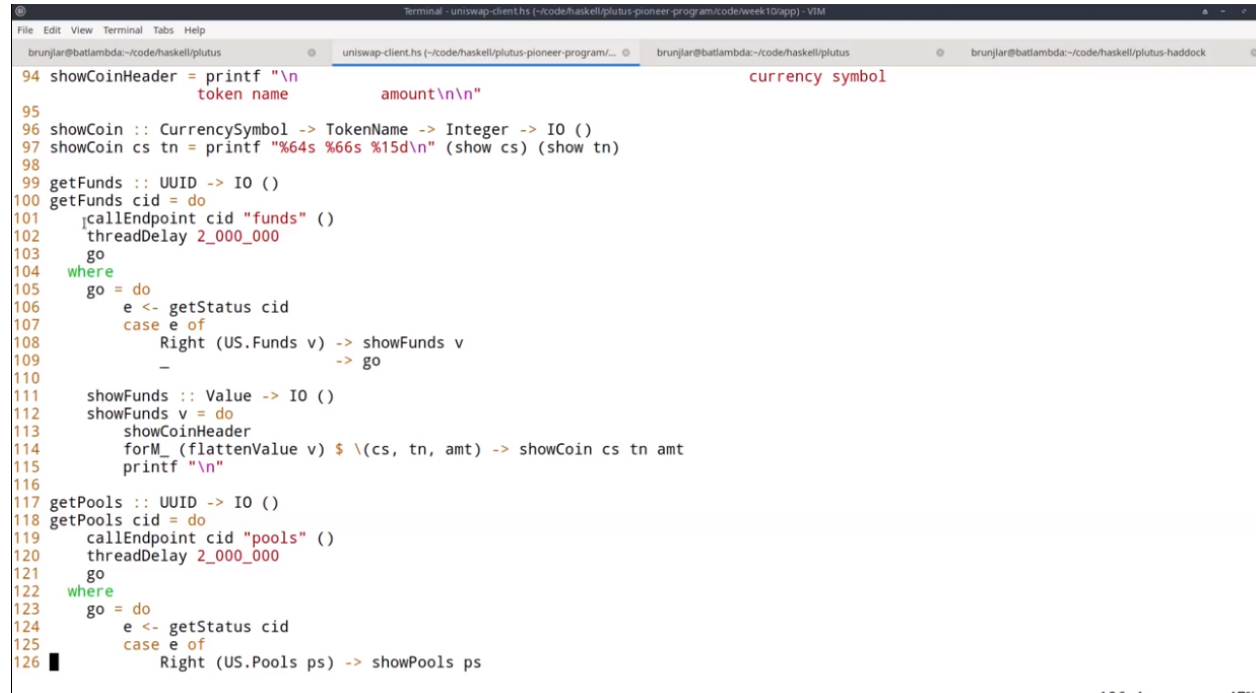
```

195 e <- getStatus cid
196 case e of
197   Right US.Swapped -> putStrLn "swapped"
198   Left err'        -> putStrLn $ "error: " ++ show err'
199   _                -> go
200
201 getStatus :: UUID -> IO (Either Text US.UserContractState)
202 getStatus cid = runReq defaultHttpConfig $ do
203   w <- req
204   GET
205   (http "127.0.0.1" /: "api" /: "new" /: "contract" /: "instance" /: pack (show cid) /: "status")
206   NoReqBody
207   (Proxy :: Proxy (JsonResponse (ContractInstanceClientState UniswapContracts)))
208   (port 8080)
209   case fromJSON $ observableState $ cicCurrentState $ responseBody w of
210     Success (Last Nothing) -> liftIO $ threadDelay 1_000_000 >> getStatus cid
211     Success (Last (Just e)) -> return e
212     _                      -> liftIO $ ioError $ userError "error decoding state"
213
214 callEndpoint :: ToJSON a => UUID -> String -> a -> IO ()
215 callEndpoint cid name a = handle h $ runReq defaultHttpConfig $ do
216   liftIO $ printf "\npost request to 127.0.1:8080/api/new/contract/instance/%s/endpoint/%s\n" (show cid) name
217   liftIO $ printf "request body: %s\n\n" $ B8.unpack $ encode a
218   v <- req
219   POST
220   (http "127.0.0.1" /: "api" /: "new" /: "contract" /: "instance" /: pack (show cid) /: "endpoint" /: pack name)
221   (ReqBodyJson a)
222   (Proxy :: Proxy (JsonResponse ()))
223   (port 8080)
224   when (responseStatusCode v /= 200) $
225     liftIO $ ioError $ userError $ "error calling endpoint " ++ name
226 where
227   h :: HttpException -> IO ()
228   h = ioError . userError . show

```

Recall this `UserContractState` had one constructor for each of the endpoints, but if there's an error during contract execution, we get the error message as a *Text*.

And if something went wrong, then we end in the third case. With these two functions - *getStatus* and *getEndpoint* - it's easy to write all the cases for the endpoints.



```
94 showCoinHeader = printf "\n\n          token name          amount\n\n\n"          currency symbol
95
96 showCoin :: CurrencySymbol -> TokenName -> Integer -> IO ()
97 showCoin cs tn = printf "%64s %66s %15d\n" (show cs) (show tn)
98
99 getFunds :: UUID -> IO ()
100 getFunds cid = do
101   callEndpoint cid "funds" ()
102   threadDelay 2_000_000
103   go
104 where
105   go = do
106     e <- getStatus cid
107     case e of
108       Right (US.Funds v) -> showFunds v
109       _                  -> go
110
111 showFunds :: Value -> IO ()
112 showFunds v = do
113   showCoinHeader
114   forM_ (flattenValue v) $ \(cs, tn, amt) -> showCoin cs tn amt
115   printf "\n"
116
117 getPools :: UUID -> IO ()
118 getPools cid = do
119   callEndpoint cid "pools" ()
120   threadDelay 2_000_000
121   go
122 where
123   go = do
124     e <- getStatus cid
125     case e of
126       Right (US.Pools ps) -> showPools ps
```

So let's maybe look at one, *getFunds*. We use the *callEndpoint* helper function that we just saw. For the endpoint named "funds", and in this case, the argument, the request body, is just *Unit*.

We wait for two seconds and then use the *getStatus* helper function. If we get a *Right*, then we show the funds, otherwise we recurse.

So we wait until we get the *Right*, because in this case "funds" should never fail. There's no way that can fail, therefore we can safely wait forever.

The *getPools* function is similar. It's more or less the same, except that instead of "funds", we have "pools".

Let's look at one more example, for creating a pool.

Again we call the endpoint and we wait for two seconds. Here something could actually go wrong. For example, if we try to create a pool where both coins are the same, or if we specify a larger liquidity than exists in the wallet, then we would get an error.

So in this case, if we get an error, we just log it to the console.

The cases for all the other endpoints are very similar.

```

109         -> go
110
111     showFunds :: Value -> IO ()
112     showFunds v = do
113         showCoinHeader
114         forM_ (flattenValue v) $ \(cs, tn, amt) -> showCoin cs tn amt
115         printf "\n"
116
117     getPools :: UUID -> IO ()
118     getPools cid = do
119         callEndpoint cid "pools" ()
120         threadDelay 2_000_000
121         go
122     where
123         go = do
124             e <- getStatus cid
125             case e of
126                 Right (US.Pools ps) -> showPools ps
127                 -> go
128
129     showPools :: [(US.Coin US.A, US.Amount US.A), (US.Coin US.B, US.Amount US.B)] -> IO ()
130     showPools ps = do
131         forM_ ps $ \(US.Coin (AssetClass (csA, tnA)), amtA), (US.Coin (AssetClass (csB, tnB)), amtB)) -> do
132             showCoinHeader
133             showCoin csA tnA (US.unAmount amtA)
134             showCoin csB tnB (US.unAmount amtB)
135
136     createPool :: UUID -> US.CreateParams -> IO ()
137     createPool cid cp = do
138         callEndpoint cid "create" cp
139         threadDelay 2_000_000
140         go
141     where
142         go = do

```

```

135     createPool :: UUID -> US.CreateParams -> IO ()
136     createPool cid cp = do
137         callEndpoint cid "create" cp
138         threadDelay 2_000_000
139         go
140     where
141         go = do
142             e <- getStatus cid
143             case e of
144                 Right US.Created -> putStrLn "created"
145                 Left err' -> putStrLn $ "error: " ++ show err'
146                 -> go
147
148     addLiquidity :: UUID -> US.AddParams -> IO ()
149     addLiquidity cid ap = do
150         callEndpoint cid "add" ap
151         threadDelay 2_000_000
152         go
153     where
154         go = do
155             e <- getStatus cid
156             case e of
157                 Right US.Added -> putStrLn "added"
158                 Left err' -> putStrLn $ "error: " ++ show err'
159                 -> go
160
161     removeLiquidity :: UUID -> US.RemoveParams -> IO ()
162     removeLiquidity cid rp = do
163         callEndpoint cid "remove" rp
164         threadDelay 2_000_000
165         go
166     where
167         go = do

```

## 10.3 Trying it Out

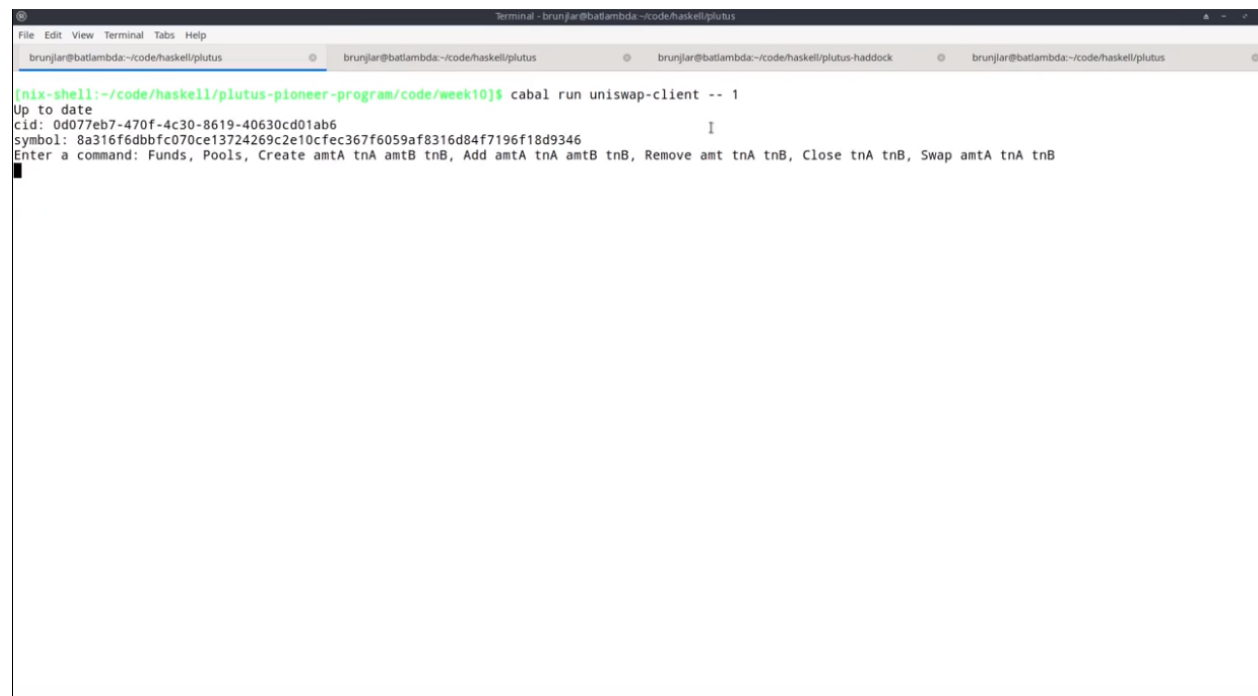
Now let's try it out.

Let's start three instances for wallets one, two and three, and try to recreate the scenario from the diagrams at the beginning.

We start it simply by using *cabal run* with a command line parameter for the number of the wallet.

```
cabal run uniswap-client -- 1
```

And we do the same for wallets 2 and 3.



```
terminal - brunjar@batilambda:~/code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus-haddock brunjar@batilambda:~/code/haskell/plutus
[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$ cabal run uniswap-client -- 1
Up to date
cid: 0d077eb7-470f-4c30-8619-40630cd01ab6
symbol: 8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
```

Here we see the log messages for the contract instance ID and the symbol for token that we can use.

So what can we do?

We can, for example, query our funds.

And we see that we have A, B, C, D with 1 million each and 100,000 Ada.

We can also look for pools, but right now, there shouldn't be any, and indeed none are listed.

So let's switch to Wallet 1, let's say this is Alice, Bob is 2 and Charlie is 3.

In the diagrams, we started with Alice setting up a liquidity pool for 1000 tokens A and 2000 B tokens.

So to do this here, we can type

```
Create 1000 'A' 2000 'B'
```

Remember that was of type *Char*, so we use single quotes.

We get the created status spec.

So it seems to have worked. We can query for pools again, and indeed there is one now.



```

terminal - brunjar@batilambda:~/code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus

[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$ cabal run uniswap-client -- 3
Up to date
cid: 6c9c6d05-433c-4e52-a750-0c1adbbfb405
symbol: 8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Funds

post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/funds
request body: []

currency symbol                                     token name      amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "A"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "B"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "C"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "D"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      ""             1000000000000

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB

```

```

terminal - brunjar@batilambda:~/code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus brunjar@batilambda:~/code/haskell/plutus

[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$ cabal run uniswap-client -- 3
Up to date
cid: 6c9c6d05-433c-4e52-a750-0c1adbbfb405
symbol: 8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Funds

post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/funds
request body: []

currency symbol                                     token name      amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "A"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "B"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "C"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "D"             1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      ""             1000000000000

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools

post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/pools
request body: []

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB

```

```
terminal - brunjar@batilambda:~code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus

[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$ cabal run uniswap-client -- 1
Up to date
cid: 0d077eb7-470f-4c30-8619-40630cd01ab6
symbol: 8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Create 1000 'A' 2000 'B'

post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/create
request body: {"cpAmountA":1000,"cpAmountB":2000,"cpCoinB":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"B"}},"cpCoinA":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"A"}}}

created
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
```

```
terminal - brunjar@batilambda:~code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus

[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$ cabal run uniswap-client -- 1
Up to date
cid: 0d077eb7-470f-4c30-8619-40630cd01ab6
symbol: 8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Create 1000 'A' 2000 'B'

post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/create
request body: {"cpAmountA":1000,"cpAmountB":2000,"cpCoinB":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"B"}},"cpCoinA":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"A"}}}

created
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools

post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/pools
request body: []

currency symbol token name amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 1000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 2000
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
```



We see that it has A and B and with the correct amounts, 1000 and 2000 respectively.

The next step was that Bob swaps 100A for Bs. So, in the console that is running Bob's wallet, we can write

```
Swap 100 'A' 'B'
```

Let's check how many funds Bob now has. As expected, he has 100 fewer As and 181 as many Bs.

```

[nix-shell:~/code/haskell/plutus-pioneer-program/code/week10]$ cabal run uniswap-client -- 2
Up to date
cid: c9100b51-c02e-4afa-9e5d-5c374f90222a
symbol: 8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Swap 100 'A' 'B'

post request to 127.0.1:8080/api/new/contract/instance/c9100b51-c02e-4afa-9e5d-5c374f90222a/endpoint/swap
request body: {"spAmountA":100,"spAmountB":0,"spCoinB":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},{"unTokenName":"B"}},"spCoinA":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},{"unTokenName":"A"}}}

swapped
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Funds

post request to 127.0.1:8080/api/new/contract/instance/c9100b51-c02e-4afa-9e5d-5c374f90222a/endpoint/funds
request body: {}

      currency symbol                                     token name      amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "A"              999900
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "B"              1000181
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "C"              1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      "D"              1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346      ""              99999981553

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB

```

Next Charlie added liquidity.

```
Add 400 'A' 800 'B'
```

Now we check the pools again.

It's 1500 and 2619. We had 1000 at the beginning, then 100 where added by Bob and now 400 by Charlie.

Now, if we go back to Alice, she wants to remove her liquidity. So let's first query her funds.

So she has less A and Bs now because she provided them as liquidity for the pool, but she has 1415 of the liquidity token.

So for example, she can burn the liquidity tokens and get tokens in exchange. She doesn't have to burn all, but in the diagram she did. So let's do this, so

```
Remove 1415 'A' 'B'
```

And let's query her funds again.

So now she doesn't have liquidity token anymore, but she got As and Bs back.

So she received 1869Bs and 1070 As.

The last step was that Charlie closes the pool. So let's switch to Charlie

```
Close 'A' 'B'
```

```
terminal - brunjar@batilambda:~code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus
Funds
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/funds
request body: []

currency symbol token name amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "C" 1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "D" 1000000
" " 10000000000

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/pools
request body: []
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Add 400 'A' 800 'B'
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/add
request body: {"apAmountB":800,"apCoinA":{"unAssetClass":[{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"}],"unTokenName":"A"}}, {"apAmountA":400,"apCoinB":{"unAssetClass":[{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"}],"unTokenName":"B"}}}}
added
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/pools
request body: []

currency symbol token name amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 1500
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 2619

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
```

```
terminal - brunjar@batilambda:~code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus brunjar@batilambda:~code/haskell/plutus
nix-shell:~code/haskell/plutus-pioneer-program/code/week10$ cabal run uniswap-client -- 1
Up to date
cid: 0d077eb7-470f-4c30-8619-40630cd01ab6
symbol: 8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Create 1000 'A' 2000 'B'
post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/create
request body: {"cpAmountA":1000,"cpAmountB":2000,"cpCoinB":{"unAssetClass":[{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"}],"unTokenName":"B"}}, {"cpCoinA":{"unAssetClass":[{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"}],"unTokenName":"A"}}}}
created
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools
post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/pools
request body: []

currency symbol token name amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 1000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 2000

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Funds
post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/funds
request body: []

currency symbol token name amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 " " 99999944228
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 999000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 998000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "C" 1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "D" 1000000
f08bce21c2e0eb79aba5c44dc19e68fe79bacf037fa7aca3b55ef317c4e1e9d9 0x97a4ba52971d7413c64dae621204387d9ec256bcbb1769195971f589456365 1415

Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
```

```

terminal - brunjar@batlamda:~code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batlamda:~code/haskell/plutus brunjar@batlamda:~code/haskell/plutus brunjar@batlamda:~code/haskell/plutus brunjar@batlamda:~code/haskell/plutus
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 2000
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Funds
post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/funds
request body: {}

currency symbol token name amount
" " 99999944228
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 999000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 998000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "C" 1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "D" 1000000
f08bce21c2e0eb79aba5c44dc19e68fe79bacf037fa7aca3b55ef317c4e1e9d9 0x97a4ba52971d7413c64dae621204387d9ec256bcba1769195971f589456365 1415
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Remove 1415 'A' 'B'
post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/remove
request body: {"rpCoinB":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"B"}}, {"rpDif
f":"1415","rpCoinA":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"A"}}}
removed
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Funds
post request to 127.0.1:8080/api/new/contract/instance/0d077eb7-470f-4c30-8619-40630cd01ab6/endpoint/funds
request body: {}

currency symbol token name amount
" " 99999917358
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 1000070
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 999869
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "C" 1000000
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "D" 1000000
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB

```

```

terminal - brunjar@batlamda:~code/haskell/plutus
File Edit View Terminal Tabs Help
brunjar@batlamda:~code/haskell/plutus brunjar@batlamda:~code/haskell/plutus brunjar@batlamda:~code/haskell/plutus brunjar@batlamda:~code/haskell/plutus
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/pools
request body: {}
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Add 400 'A' 800 'B'
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/add
request body: {"apAmountB":800,"apCoinA":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenNam
e":"A"}}, {"apAmountA":400,"apCoinB":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"B
"}}}}
added
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/pools
request body: {}

currency symbol token name amount
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "A" 1500
8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346 "B" 2619
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Close 'A' 'B'
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/close
request body: {"clpCoinB":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"B"}}, {"clpC
oinA":{"unAssetClass":{"unCurrencySymbol":"8a316f6dbbfc070ce13724269c2e10cfec367f6059af8316d84f7196f18d9346"},"unTokenName":"A"}}}
closed
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB
Pools
post request to 127.0.1:8080/api/new/contract/instance/6c9c6d05-433c-4e52-a750-0c1adbbfb405/endpoint/pools
request body: {}
Enter a command: Funds, Pools, Create amtA tnA amtB tnB, Add amtA tnA amtB tnB, Remove amt tnA tnB, Close tnA tnB, Swap amtA tnA tnB

```

And if now we look for pools, then again, we don't get any. So it all seems to work.

Finally, we will look at how to use the front-end without Haskell and just use something like curl.

Let's look at the file *status.sh* which you will find in the code folder.



```
Terminal - brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10
File Edit View Terminal Tabs Help
brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10$ cat status.sh
#!/bin/sh
curl "http://localhost:8080/api/new/contract/instance/$(cat W$1.cid)/status" | jq ".cicCurrentState.observableState"
brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10$
```

This script expects one argument, that's the wallet.

Then we just *curl* to this URL and interpolate the content of the correct wallet file given by the first parameter.

And because the output is very unwieldy, we pipe it through *jq* and get the current state and observable state of the resulting json.

So if I try this right now for Wallet 1, for example, we see that wallet one at the moment has these amounts of the tokens ABCD.

At least that was the last status. Maybe it's not up to date.

Let's also look at the *funds.sh* script.

So that again, only takes one parameter, one argument, the wallet, in order to put the correct instance ID in the URL, and then uses the *funds* endpoint.

And this is a POST request, so we need a request body. This is *Unit* because the funds endpoint doesn't require any arguments except the *Unit* argument.

A bit more interesting is what to do with the post request that do have interesting arguments. For example, if now, Wallet 1 wants to create a pool again with 1000A and 2000 B. In that case we need a request body for the correct parameters for the *CreateParams*.

So let's look at this in *create.sh*

In principle, the curl is simple, so now again, contract instance ID but now with endpoint *create*. But the question is what to write in this body.

We use similar arguments to in the Haskell implementation. So first the wallet and then the A amount, A token, B amount, and B token.

```
terminal - brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10
File Edit View Terminal Tabs Help
brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10$ cat status.sh
#!/bin/sh
curl "http://localhost:8080/api/new/contract/instance/$(cat W$1.cid)/status" | jq ".cicCurrentState.observableState"
brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10$
```

```
terminal - brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10
File Edit View Terminal Tabs Help
brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10$ cat status.sh
{
  "getValue": [
    [
      {
        "unCurrencySymbol": ""
      },
      [
        {
          "unTokenName": ""
        },
        99999917358
      ]
    ],
    [
      {
        "unCurrencySymbol": "8a316f6dbbfc070ce13724269c2e10cfe367f6059af8316d84f7196f18d9346"
      },
      [
        {
          "unTokenName": "A"
        },
        1000070
      ],
      [
        {
          "unTokenName": "B"
        },
        999869
      ],
      [
        {
          "unTokenName": "C"
        },
        1000000
      ],
      [
        {
          "unTokenName": "D"
        },
        1000000
      ]
    ]
  ]
}
```

```
Terminal - brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10
File Edit View Terminal Tabs Help
brunjar@batlambda:~/code/haskell/plutus  brunjar@batlambda:~/code/haskell/plutus  brunjar@batlambda:~/code/haskell/plutus  brunjar@batlambda:~/code/haskell/plutus  brunjar@batlambda:~/code/haskell/plutus-p...
{
  "unCurrencySymbol": "2adddf64d97a95678d417ed616d9e698441718d772e2fcdec5fe62cf33856e96"
},
[
  [
    {
      "unTokenName": "Uniswap"
    },
    0
  ]
],
[
  {
    "unCurrencySymbol": "f08bce21c2e0eb79aba5c44dc19e68fe79bacf037fa7aca3b55ef317c4e1e9d9"
  },
  [
    {
      "unTokenName": "Pool State"
    },
    0
  ],
  [
    {
      "unTokenName": "\u0000x97a4ba52971d7413c64dae621204387d9ec256bcbb1769195971f589456365"
    },
    0
  ]
]
],
"tag": "Funds"
}
}
brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10$ cat funds.sh
#!/bin/sh
curl "http://localhost:8080/api/new/contract/instance/$(cat W$1.cid)/endpoint/funds" \
--header 'Content-Type: application/json' \
--data-raw '[]'
brunjar@batlambda:~/code/haskell/plutus-pioneer-program/code/week10$
```

So maybe we should first check whether it works.

```
./create.sh 1 1000 A 2000 B
```

And now if we wait a few seconds and then query the status, it should have updated.

We can now run

```
./status.sh 1
```

Now we have this new liquidity pool with A and B.

So the remaining question is, how did we get the JSON body for the curl call? It's hard to do this by hand, but if we look back at the Haskell output, what we did was here, for example, for create, to always write the URL we are calling and also the request body.

And we can check the code for this. If we look at uniswap-client. It is in the helper function *callEndpoint* on lines 216 and 217.

So we get the *a*, that's just a Haskell value with an instance of *ToJSON* so that can be encoded to JSON, and we just use encode from the Aeson library. This is now a byte string, but in order to write that to the console, we need two strings, so we use something from the ByteString library

```
Data.ByteString.Lazy.Char8
```

And we unpack this ByteString to a string and then log it. So this is a good way to figure out what requests bodies to use.

You don't, of course, have to write a whole program, you can also do that in the REPL. You just need a value of the correct type and then use Aeson to encode it and look at the result and then you see the shape of the json that is expected.





[illegible]

```

190 callEndpoint cid "swap" sp
191 threadDelay 2_000_000
192 go
193 where
194   go = do
195     e <- getStatus cid
196     case e of
197       Right US.Swapped -> putStrLn "swapped"
198       Left err'        -> putStrLn $ "error: " ++ show err'
199     -> go
200
201 getStatus :: UUID -> IO (Either Text US.UserContractState)
202 getStatus cid = runReq defaultHttpConfig $ do
203   w <- req
204   GET
205   (http "127.0.0.1" /: "api" /: "new" /: "contract" /: "instance" /: pack (show cid) /: "status")
206   NoRequestBody
207   (Proxy :: Proxy (JsonResponse (ContractInstanceClientState UniswapContracts)))
208   (port 8080)
209   case fromJSON $ observableState $ cicCurrentState $ responseBody w of
210     Success (Last Nothing) -> liftIO $ threadDelay 1_000_000 >> getStatus cid
211     Success (Last (Just e)) -> return e
212     -> liftIO $ ioError $ userError "error decoding state"
213
214 callEndpoint :: ToJSON a => UUID -> String -> a -> IO ()
215 callEndpoint cid name a = handle h $ runReq defaultHttpConfig $ do
216   liftIO $ printf "\npost request to 127.0.1:8080/api/new/contract/instance/%s/endpoint/%s\n" (show cid) name
217   liftIO $ printf "request body: %s\n" $ B8.unpack $ encode a
218   v <- req
219   POST
220   (http "127.0.0.1" /: "api" /: "new" /: "contract" /: "instance" /: pack (show cid) /: "endpoint" /: pack name)
221   (RequestBodyJson a)
222   (Proxy :: Proxy (JsonResponse ()))
223   (port 8080)
224   when (responseStatusCode v /= 200) $
225     liftIO $ ioError $ userError $ "error calling endpoint " ++ name
226 where
227   h :: HttpException -> IO ()
228   h = ioError . userError . show

```



## AWS NODE SETUP

I started with a fresh `t2.large` AWS Ubuntu EC2 instance using AMI `ami-0ff4c8fb495a5a50d` and adding a 60Gb data volume.

First, mount the data volume.

```
sudo mkfs -t xfs /dev/xvdh
sudo mkdir /data
sudo mount /dev/xvdh /data
sudo chown ubuntu:ubuntu /data
```

### 11.1 Setup the IOHK Cache

```
sudo mkdir -p /etc/nix
cat <<EOF | sudo tee /etc/nix/nix.conf
substituters = https://cache.nixos.org https://hydra.iohk.io https://iohk.cachix.org
trusted-public-keys = cache.nixos.org-1:6NCHdD59X43Io0gWypbMrAURkbJ16ZPMQFGspcDShjY=
↳ hydra.iohk.io:f/Ea+s+dFdN+3Y/G+FDgSq+a5NEWhJGzdjvKNGv0/EQ= iohk.cachix.org-
↳ 1:DpRUyj7h7V830dp/i6Nti+NE02/nhblbov/8MW7Rqoo=
EOF
```

### 11.2 Install Nix

We use a little trick here to let Nix use a symlinked directory. This is not recommended if you plan to have setups on multiple machines with potentially different configurations, but that doesn't matter here.

```
mkdir /data/nix
sudo ln -s /data/nix /nix
echo "export NIX_IGNORE_SYMLINK_STORE=1" >> ~/.bashrc
source ~/.bashrc
curl -L https://nixos.org/nix/install | sh
. /home/ubuntu/.nix-profile/etc/profile.d/nix.sh
```

## 11.3 Download the Cardano Node

```
cd /data
git clone https://github.com/input-output-hk/cardano-node
cd cardano-node
git checkout tags/1.29.0 -b 1_29_0
```

## 11.4 Build the node

```
nix-build -A scripts.alonzo-purple.node -o result/alonzo-purple/cardano-node-alonzo-
↳purple
nix-build -A cardano-cli -o result/alonzo-purple/cardano-cli
```

## 11.5 Start the node

```
cd /data/cardano-node/result/alonzo-purple
./cardano-node-alonzo-purple/bin/cardano-node-alonzo-purple
```

Leave this running and open another shell.

## 11.6 Setup some environment variables

```
echo "export CARDANO_CLI=/data/cardano-node/result/alonzo-purple/cardano-cli/bin/cardano-
↳cli" >> ~/.bashrc
echo "export TESTNET_MAGIC_NUM=8" >> ~/.bashrc
echo "export CARDANO_NODE_SOCKET_PATH=/data/cardano-node/result/alonzo-purple/state-node-
↳alonzo-purple/node.socket" >> ~/.bashrc
source ~/.bashrc
```

You can check on the status of the node with:

```
$CARDANO_CLI query tip --testnet-magic $TESTNET_MAGIC_NUM
```

You should see something like this:

```
{
  "epoch": 60,
  "hash": "eb9453a91760928b286ea5137d6f9325f89f78b9c643f1e789c63c74b1934fa3",
  "slot": 431693,
  "block": 21187,
  "era": "Mary",
  "syncProgress": "19.01"
}
```

When the node has fully synced you will see that the era has changed to Alonzo.

```
{  
  "epoch": 289,  
  "hash": "7521f071d0bfc050cde302f1352ed44c2fc74927f1e28afea1b1df2c4c012d5c",  
  "slot": 2079664,  
  "block": 102026,  
  "era": "Alonzo",  
  "syncProgress": "100.00"  
}
```

You can use `jq` if you ever want to get some specific information on its own, for example:

```
sudo apt update  
sudo apt install jq -y  
  
$CARDANO_CLI query tip --testnet-magic $TESTNET_MAGIC_NUM | jq -r '.syncProgress'  
100.0
```



## WALLETS AND FUNDS

## 12.1 Some Helper Scripts

I have a repo that contains a few helper scripts that I use. It's rough and ready, but saves a little time for some common tasks.

```
cd /data
git clone https://github.com/chris-moreton/plutus-scripts
```

### 12.1.1 Generate test addresses

Run the generate wallets script to generate a few addresses.

```
cd /data/plutus-scripts
./generateAddresses.sh
```

This script wraps the following command.

```
# example only, don't run this
$CARDANO_CLI address key-gen --verification-key-file main.vkey --signing-key-file main.
↪ skey
```

This will create some `.addr`, `.skey` and `.vkey` files in the `wallets` directory.

### 12.1.2 Use the Faucet

If you have access to the testnet faucet, transfer some test Ada to the main wallet.

```
./faucet.sh SECRET_KEY
```

Then, check that it has arrived. It should arrive within a minute or so.

```
cd /data/plutus-scripts
./balance.sh main
```

TxHash	TxIx	Amount
40f0fa60a71e247e3eca46147fc159080aa7667763ae8c3be00b2e48400bbccd		0
→ 1000000000000000 lovelace + TxOutDatumHashNone		

The `balance.sh` script wraps the following command.

```
# example only, don't run this
$CARDANO_CLI query utxo --address $(cat ./wallets/$1.addr) --testnet-magic $TESTNET_
↳MAGIC_NUM
```

### 12.1.3 Transfer some funds

We will transfer some funds to `wallet1`. This uses another helper script, which takes the sending wallet as an argument and then asks for the UTxO, amount and receiving wallet name.

```
./sendFromWallet.sh main
```

TxHash	TxIx	Amount
-----		
40f0fa60a71e247e3eca46147fc159080aa7667763ae8c3be00b2e48400bbccd		0
↳ 1000000000000 lovelace + TxOutDatumHashNone		

TX row number: 1  
Lovelace to send: 1000000000000  
Receiving wallet name: wallet1

Transaction successfully submitted.

The `sendFromWallet.sh` script wraps the following commands.

```
# example only, don't run these

$CARDANO_CLI transaction build \
  --tx-in ${FROM_UTXO} \
  --tx-out ${TO_WALLET_ADDRESS}+${LOVELACE_TO_SEND} \
  --change-address=${FROM_WALLET_ADDRESS} \
  --testnet-magic ${TESTNET_MAGIC_NUM} \
  --out-file tx.build \
  --alonzo-era

$CARDANO_CLI transaction sign \
  --tx-body-file tx.build \
  --signing-key-file ./wallets/${FROM_WALLET_NAME}.skey \
  --out-file tx.signed

$CARDANO_CLI transaction submit --tx-file tx.signed --testnet-magic $TESTNET_MAGIC_NUM
```

Check that it has arrived.

```
./balance.sh wallet1
```

TxHash	TxIx	Amount
bd7422ef2cd55d1c5a33601a3b75b080bc3742856e5ddb8dfdfae07f583c7af1	0	10000000000
↳ lovelace + TxOutDatumHashNone		

## ALWAYSSUCCEEDS SCRIPT

**Note:** These instructions should work as presented if you have followed the [AWS Node Setup](#) section and the [Wallets and Funds](#) sections. If not, you may need to improvise a little.

Plutus scripts get compiled down to the following format.

```
{
  "type": "PlutusScriptV1",
  "description": "",
  "cborHex": "4e4d0100003322220051200120011"
}
```

This is the `AlwaysSucceeds.plutus` script whose validator always succeeds, which means that anyone will be able to consume any UTxO sitting at its address. The script can be found in the `/data/plutus-scripts/scripts` directory.

### 13.1 Pay to the Script

Using the `payToScript.sh` helper script, you can send 990000000 lovelace from `wallet1` to the `AlwaysSucceeds.plutus` script with a datum of 6666, allowing for fees of 200000.

The helper script `payToScript.sh` constructs a transaction, and, as one of its inputs, it reads the `AlwaysSucceeds.plutus` file in the `scripts` directory.

```
./payToScript.sh 990000000 AlwaysSucceeds 6666 wallet1
```

TxHash	TxIx	Amount
-----		
bd7422ef2cd55d1c5a33601a3b75b080bc3742856e5ddb8dfdfae07f583c7af1	0	1000000000
↳ lovelace + TxOutDatumHashNone		
TX row number: 1		
Transaction successfully submitted.		

The `payToScript.sh` script wraps the following commands.

```
$CARDANO_CLI transaction build \
  --tx-in ${SELECTED_UTXO} \
  --tx-out ${TO_ADDR}+${PAYMENT} \
  --tx-out-datum-hash ${DATUM_HASH} \
  --change-address=${FROM_ADDR} \
```

(continues on next page)

(continued from previous page)

```

--testnet-magic $TESTNET_MAGIC_NUM \
--out-file tx.build \
--alonzo-era

$CARDANO_CLI transaction sign \
--tx-body-file tx.build \
--signing-key-file ./wallets/${SELECTED_WALLET_NAME}.skey \
--testnet-magic $TESTNET_MAGIC_NUM \
--out-file tx.signed \

$CARDANO_CLI transaction submit --tx-file tx.signed --testnet-magic $TESTNET_MAGIC_NUM

```

Check that the funds arrive in the script using the `contractBalance.sh` script. You may see a lot of UTxOs sitting at the `AlwaysSucceeds` script address and hopefully one of them will be yours.

The UTxOs at this address are now *locked* in the sense that they are guarded by a validator. The script address is the hash of the validator.

```

./contractBalance.sh AlwaysSucceeds

```

TxHash	TxIx	Amount
063a62b69e51296417687077f8df67f1b2fe1568830ad56fb0f04d22739e69e2	0	50000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "b7a4cc0f36854309590c132e75dad06a4f6045e57ac93e6dafc9bf0d0018247d"		
44412566ec42af806660fe9846a71b50eae1b7028116a3d666cab3ba1f02d7ee	0	
↳ 10000000000000000000 lovelace + TxOutDatumHashNone		
56382a3e1789df882114b2322787f1785eac71b19675ee88fd1dc6ca807ddc02	0	999888777
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		
843f4ffa4aafc5ed968d0a9f0fb8a203796b66327343246bfd8d4ca1d361c2f8	0	99000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		
8657ff66828f90ab7d45fb2e9f10286d9887f49bc83f7cf3d7b45e8fd1068aaf	0	10000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "9e1199a988ba72ffd6e9c269cadb3b53b5f360ff99f112d9b2ee30c4d74ad88b"		
8c5f24a4eee17773d2def2ee1493248b1c45c56e6851d6f330deee1dc23a21f	0	1010011010
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "915e807fa63409181d1533195753e3170587b1edc089be670ab483da8f9bcd48"		
8f75351368cc2521315ac9908f0532a00e996e35644cbd9db4d616a7122c7491	0	
↳ 979199655182 lovelace + TxOutDatumHashNone		
f441da5a5f0ee6057a98650bf4c2a4931906e37acfd2d705cb208eda48cef92	0	
↳ 100000000000 lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "df5078aee07dd171a343fb99d5fc1b5462fb3c94d82bf72dc1b77d9c0aceec29"		

The `contractBalance.sh` script is an extension of the `balance.sh`. It just has the extra step of determining the address of the contract.

In this case, the balance of UTxO number 4 is 99000000 and the datum hash is 9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710. We can check that this is the correct datum hash.

```

$CARDANO_CLI transaction hash-script-data --script-data-value 6666
9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710

```





(continued from previous page)

```

843f4ffa4aafc5ed968d0a9f0fb8a203796b66327343246bfd8d4ca1d361c2f8    0    990000000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"
8657ff66828f90ab7d45fb2e9f10286d9887f49bc83f7cf3d7b45e8fd1068aaf    0    100000000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"9e1199a988ba72ffd6e9c269cadb3b53b5f360ff99f112d9b2ee30c4d74ad88b"
8c5f24a4eeee17773d2ddef2ee1493248b1c45c56e6851d6f330deee1dc23a21f    0    1010011010L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"915e807fa63409181d1533195753e3170587b1edc089be670ab483da8f9bcd48"
8f75351368cc2521315ac9908f0532a00e996e35644cbd9db4d616a7122c7491    0    L
↳979199655182 lovelace + TxOutDatumHashNone
f441da5a5f04ee6057a98650bf4c2a4931906e37acfd2d705cb208eda48cef92    0    L
↳100000000000 lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"df5078aee07dd171a343fb99d5fc1b5462fb3c94d82bf72dc1b77d9c0aceec29"

```

TX row number: 4

Select Collateral UTxO

Wallet Name: fees

	TxHash	TxIx	Amount
-----	-----	-----	-----
	7678d8d6b95ed026d7c690fb53419bdaa580cb00c56450ac3bd97712dd71ca4e	0	10000000000L
	↳lovelace + TxOutDatumHashNone		

TX row number: 1

Transaction successfully submitted.

The getFromScript.sh script wraps the following commands.

```
$CARDANO_CLI query protocol-parameters --testnet-magic $TESTNET_MAGIC_NUM > params.json
```

```

$CARDANO_CLI transaction build \
  --tx-in ${COLLATERAL_TX} \
  --tx-in ${SCRIPT_UTXO} \
  --tx-in-datum-value "${DATUM_VALUE}" \
  --tx-in-redeemer-value "${REDEEMER_VALUE}" \
  --tx-in-script-file $SCRIPT_FILE \
  --tx-in-collateral=${COLLATERAL_TX} \
  --change-address=${FEE_ADDR} \
  --tx-out ${TO_ADDR}+${PAYMENT} \
  --tx-out-datum-hash ${DATUM_HASH} \
  --out-file tx.build \
  --testnet-magic $TESTNET_MAGIC_NUM \
  --protocol-params-file "params.json" \
  --alonzo-era

$CARDANO_CLI transaction sign \
  --tx-body-file tx.build \
  --signing-key-file ./wallets/${SIGNING_WALLET}.skey \
  --testnet-magic $TESTNET_MAGIC_NUM \
  --out-file tx.signed \

```

(continues on next page)

(continued from previous page)

```
$CARDANO_CLI transaction submit --tx-file tx.signed --testnet-magic $TESTNET_MAGIC_NUM
```

Let's check that it arrived in wallet1 as expected.

```
./balance.sh wallet1
```

TxHash	TxIx	Amount
ee22529028220bb2d2cbda634fbe982602afd5baf7f173341e2c8f9157e2912d	1	99000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		



## ALWAYSFAILS SCRIPT

---

**Note:** These instructions should work as presented if you have followed the [AWS Node Setup](#) section and the [Wallets and Funds](#) sections. If not, you may need to improvise a little.

---

Now we will try similar transactions with a script whose validator always fails. This time we will lose our collateral.

### 14.1 Sending

First, lock some lovelace in the AlwaysFails script which will be picked up from the `./scripts` directory of the cloned `plutus-scripts` repository.

```
./payToScript.sh 990000000 AlwaysFails 6666 wallet1
Wallet Name: wallet1
```

TxHash	TxIx	Amount
060aa2af10655a4b893bb4b828aa2288a5a18f1dd8941a7f99ffe8d3bd1d71f1	1	990000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		
c1658d652d7e318ae990da7973a0bbca6d787130e079102a16c1b3568ccfe8df	0	
↳ 99900831551 lovelace + TxOutDatumHashNone		

```
TX row number: 2
Transaction successfully submitted.
```

Check that it has arrived.

```
./contractBalance.sh AlwaysFails
```

TxHash	TxIx	Amount
0913d72c55b3d6e765eb51f1a5da1436ea0554a89d499fe3398d20517f0b455e	0	990000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		
29f82f40603c4328e6efffb7c6e8851fe9540d18ddc0930b188896f6b016e141	0	1000000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "ee5c9e2778c6c398366c5b9cfd67a888081f7626ca0ac392faca5981e59ff759"		
5589e823cf148597cbf64dc7cb5ebcd3957d5fc83c3521b281daa9f9c490c8ab	0	999888777
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		

## 14.2 Grabbing

Now we try to get some funds from the script, but it can't succeed.

```
./getFromScript.sh AlwaysFails 6666 42 wallet1
```

```
Select Script UTx0
```

TxHash	TxIx	Amount
0913d72c55b3d6e765eb51f1a5da1436ea0554a89d499fe3398d20517f0b455e	0	990000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		
29f82f40603c4328e6efffb7c6e8851fe9540d18ddc0930b188896f6b016e141	0	1000000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"ee5c9e2778c6c398366c5b9cfd67a888081f7626ca0ac392faca5981e59ff759"		
5589e823cf148597cbf64dc7cb5ebcd3957d5fc83c3521b281daa9f9c490c8ab	0	999888777
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"		

```
TX row number: 1
```

```
Select Collateral UTx0
```

```
Wallet Name: fees
```

TxHash	TxIx	Amount
ee22529028220bb2d2cbda634f982602afd5baf7f173341e2c8f9157e2912d	0	889819803
↳lovelace + TxOutDatumHashNone		

```
TX row number: 1
```

Command failed: transaction build Error: The following scripts have execution failures: the script **for** transaction input 1 (in the order of the TxIds) failed with The Plutus

↳script evaluation failed: An error has occurred: User error:

The provided Plutus code called 'error'.

Command failed: transaction submit Error: Error **while** submitting tx:

↳ShelleyTxValidationError ShelleyBasedEraAlonzo (ApplyTxError [UtxowFailure,

↳(WrappedShelleyEraFailure (UtxoFailure (ValueNotConservedUTx0 (Value 0 (fromList [])),

↳(Value 99900831551 (fromList []))))) ,UtxowFailure (WrappedShelleyEraFailure,

↳(UtxoFailure (BadInputsUTx0 (fromList [TxInCompact (TxId {\_unTxId = SafeHash

↳"c1658d652d7e318ae990da7973a0bbca6d787130e079102a16c1b3568ccfe8df"}) 0]))))])

## HELLOWORLD SCRIPT

---

**Note:** These instructions should work as presented if you have followed the [AWS Node Setup](#) section and the [Wallets and Funds](#) sections. If not, you may need to improvise a little.

---

### 15.1 Compiling Plutus Scripts

First, we need to go into a Nix shell. The shell from `cardano-node` doesn't work for me, so I cloned the `plutus` repository and used its shell instead.

```
cd /data
git clone https://github.com/input-output-hk/plutus
cd plutus
nix-shell
```

Now we will clone the `Alonzo-testnet` repo.

```
cd /data
git clone https://github.com/input-output-hk/Alonzo-testnet
```

Now, we can change to the `plutus-helloworld` directory and run the code.

```
cd /data/Alonzo-testnet/resources/plutus-sources/plutus-helloworld
```

If you look in the `plutus-helloworld.cabal` file in, you'll find some executables defined, one of which is called `plutus-helloworld`.

Running this executable will create compiled Plutus code as output.

```
cabal update
cabal run plutus-helloworld
```

This outputs, among other things, something like the following.

```
ExBudget {_exBudgetCPU = ExCPU 2443000, _exBudgetMemory = ExMemory 370}
```

These value represent the expected CPU and memory usage required to run the script. Fees for running scripts are calculated using these values.

It also outputs a file called `result.plutus`, which contains the compiled Plutus code.

```
{
  "type": "PlutusScriptV1",
  "description": "",
  "cborHex":
  "585e585c01000033322232323232323232223335300c333500b00a0033004120012009235007009008230024830af38f1"
}
```

We can now move this script to our `scripts` directory and give it a more useful name.

```
mv result.plutus /data/plutus-scripts/scripts/HelloWorld.plutus
```

Now we will lock some funds in the script. We will use the datum `79600447942433`. You will see from the comments in the `HelloWorld.hs` file that this is the `hello world` message converted to an Integer and shortened to fit within the 8-byte limit for an `int` datum.

```
cd /data/plutus-scripts

./payToScript.sh 62500000 HelloWorld 79600447942433 wallet1
Wallet Name: wallet1
```

TxHash	TxIx	Amount
060aa2af10655a4b893bb4b828aa5a18f1dd8941a7f99ffe8d3bd1d71f1	1	990000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710" ae892df5c983f98dff69e7a748d8d8609f823e7a67c9d2a8834d9bd1927a91ae		
↳99801663102	lovelace + TxOutDatumHashNone	

```
TX row number: 2
Transaction successfully submitted.
```

The `hello world` value is defined in the script as

```
hello :: Data
hello = I 0x48656c6c6f21
```

And the validator will check that the datum matches this value. Any UTxO at this script address with a different datum will not be spendable.

```
helloWorld :: Data -> Data -> Data -> ()
helloWorld datum redeemer context = if datum P.== hello then () else (P.error ())
```

So, let's lock some more lovelace in the script but with a different datum.

```

./payToScript.sh 32500000 HelloWorld 89600447942433 wallet1
Wallet Name: wallet1

```

TxHash	TxIx	Amount
ca54c8370065e3d385720b9f863d115a6ffb54dfc5b517965cd1a9b02bd34ac9	1	665000000

```

└─lovelace + TxOutDatumHashNone
TX row number: 1
Transaction successfully submitted.

```

Let's look at the UTXOs locked in the script.



```
./contractBalance.sh HelloWorld
```

We see a lot of them, including our first one which has a datum hash of 8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1. We know this is the hash of the correct datum, so we should be able to spend this UTxO. We don't see any others with this datum hash, which means that those ones are locked forever.

```
./contractBalance.sh HelloWorld
```

TxHash	TxIx	Amount
0dfec1295895d877edf15f323df63f43aa4501bfc8ee0483512c13550b6f4a65 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"ee5c9e2778c6c398366c5b9cfd67a888081f7626ca0ac392faca5981e59ff759"	0	980000000_
867226273c7de3bbbe9e94f2451bafc10f20a66d3018142e87490349c92b9db0 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"ea7db071348c44ee3ba423fbf61c57edc91167f78835d037c5c7503ed1a5fa5d"	0	325000000_
325704fa84cc1bfbbf69688a21d66157ddc7145be92567b6068d31de31bbb33c ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"74fa514fdca080be51ea4fce15f6033b754c5dc3455cb9db8dfd930623a2b4bb"	0	1000000000_
5ae97b8af41817ed4866360d10b48d9a535421fe3ee3497a09e6f4fe2d44251e ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"52d800d39486d8234e08050de3fa06296497a3a44343b4494801eb502ce38f93"	0	1000000000_
71c1fcf524dbd24be33e27ed9a0f9e3a8648609401d47c41c66299573052dbbd ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"	0	6666666_
79fc6f53b741f2dacb6f46ee723a99b4590566f71eaa970f2c859539a2621fbf ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"c22bc0debed6579418c5a07e591e5623eac7f5bf0b0d1906e7b35c7adfe66e7d"	0	980000000_
978681fa039c391bf37b1f2ba57312a10e2b823cf68c01fb627973b36064e452 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"4250ea713ad7ba3b121621a8d14d8e39a4300065314b7ce9a40526acf992c8e3"	1	512000000_
a05563264c201047514185b38f6af833cc62122074aba5d217506b2be4f5955e ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"	0	625000000_
a9b481492d5633dbf03cc92ea59e1c8f5b5c1b1f63abe9b3915a4cabf716602e ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"7cfec515f56d4413375aa9775f5de15ee60180861e9eaa954bcf9d015054857c"	0	1000000000_
bede7e2a28c9bdbc5e52752756c30b93ac7523cacdf42ad474fdc04ec750dee5 ↳500000000000 lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"b26b5f203ca94dda3d333621ea493f7ca26fef90fd1cc5fa678b38737371cb79"	0	_
dd58557b3e780703126a60e340fb13a681b47793771e5980dbb0ef479c905db6 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"ff92136db7aec02e06f0d93ed5cbea5d33360061e5d3cab51a827d65fdeb33ad"	0	1000000000_
e97ca5246c9b1565250e2cc5078d770564463a57c13125a37e4906c1f7dc0680 ↳500000000000 lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"9ad30ffde0d1931ed4f145fa0a0d320a067051bfab1b08cbdb79e9f26df55df3"	0	_
f6179d20172fec17caae32791623da52e2aa3bff04304389f693672aa1e3dae3 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"	0	1000000000_

We will try to get some funds from the one with the 89600447942433 datum.

If we pass in a datum that is not 89600447942433, we'll get the following error. The 42 here is a redeemer value that is ignored in this case.

```
./getFromScript.sh HelloWorld 69600447942433 42 wallet1
...
...
Command failed: transaction build Error: The following scripts have execution failures:
↳ the script for transaction input 0 (in the order of the TxIds) failed with The Plutus
↳ script witness has the wrong datum (according to the UTxO). The expected datum value
↳ has hash "8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"
Command failed: transaction submit Error: Error while submitting tx:
↳ ShelleyTxValidationError ShelleyBasedEraAlonzo (ApplyTxError [UtxowFailure
↳ (WrappedShelleyEraFailure (UtxoFailure (ValueNotConservedUTxO (Value 0 (fromList []))
↳ (Value 9738994653 (fromList []))))) ,UtxowFailure (WrappedShelleyEraFailure
↳ (UtxoFailure (BadInputsUTxO (fromList [TxInCompact (TxId {_unTxId = SafeHash
↳ "8ae6249f7df83f3f465e843c419985a816f27f0ff8ab5214ae7dc68c20d52da7"}) 0]))))])]
```

So, we'll pass in the matching datum.

```
./getFromScript.sh HelloWorld 89600447942433 42 wallet1

=====
Select Script UTxO
=====
=====

```

TxHash	TxIx	Amount
0dfec1295895d877edf15f323df63f43aa4501bfc8ee0483512c13550b6f4a65	0	980000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "ee5c9e2778c6c398366c5b9cfd67a888081f7626ca0ac392faca5981e59ff759"		
2d9400af7637b05b34e96c66781f087c7a28c7da8b4482b98897807dfe84efd6	0	325000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "3519b7fbee1f70218539524e3b50ba8fa67b6d769cfa6fee4d4356e800342956"		
...		
a05563264c201047514185b38f6af833cc62122074aba5d217506b2be4f5955e	0	625000000
↳ lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳ "8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"		

```
TX row number: 2
=====
Select Collateral UTxO
=====
=====
Wallet Name: fees

```

TxHash	TxIx	Amount
65e74914ad63e7e3372da140a3285b3d7eea879d02d6b814cf1f23df40c44418	0	10000000000
↳ lovelace + TxOutDatumHashNone		
c5b11e878a7dcebe5a52eb32eff5d83c3c76e35d13a2106aab811535dff5e3f6	0	10000000000
↳ lovelace + TxOutDatumHashNone		

```
TX row number: 1
Receiving Wallet: wallet2

Command failed: transaction submit Error: Error while submitting tx:
↳ ShelleyTxValidationError ShelleyBasedEraAlonzo (ApplyTxError [UtxowFailure
↳ (WrappedShelleyEraFailure (UtxoFailure (UtxosFailure (ValidationTagMismatch (IsValid
↳ True))))))])]
```

This time, the datums match, but the value of the datum is incorrect, and we fail validation and so do not submit the

transaction to the blockchain.

So, let's try to get some funds from the UTxO with the hello world message as a datum. The validator script will let us unlock that one.

```
./getFromScript.sh HelloWorld 79600447942433 42 wallet1
```

```
=====
```

```
Select Script UTxO
```

```
=====
```

TxHash	TxIx	Amount
0dfec1295895d877edf15f323df63f43aa4501bfc8ee0483512c13550b6f4a65 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"ee5c9e2778c6c398366c5b9cfd67a888081f7626ca0ac392faca5981e59ff759"	0	980000000 <sub>␣</sub>
2d9400af7637b05b34e96c66781f087c7a28c7da8b4482b98897807dfe84efd6 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"3519b7fbee1f70218539524e3b50ba8fa67b6d769cfa6fee4d4356e800342956"	0	325000000 <sub>␣</sub>
325704fa84cc1bfbbf69688a21d66157ddc7145be92567b6068d31de31bbb33c ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"74fa514fdca080be51ea4fce15f6033b754c5dc3455cb9db8dfd930623a2b4bb"	0	1000000000 <sub>␣</sub>
5ae97b8af41817ed4866360d10b48d9a535421fe3ee3497a09e6f4fe2d44251e ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"52d800d39486d8234e08050de3fa06296497a3a44343b4494801eb502ce38f93"	0	1000000000 <sub>␣</sub>
71c1fcf524dbd24be33e27ed9a0f9e3a8648609401d47c41c66299573052dbbd ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"	0	6666666 <sub>␣</sub>
79fc6f53b741f2dacb6f46ee723a99b4590566f71eaa970f2c859539a2621fbf ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"c22bc0debed6579418c5a07e591e5623eac7f5bf0b0d1906e7b35c7adfe66e7d"	0	980000000 <sub>␣</sub>
978681fa039c391bf37b1f2ba57312a10e2b823cf68c01fb627973b36064e452 ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"4250ea713ad7ba3b121621a8d14d8e39a4300065314b7ce9a40526acf992c8e3"	1	512000000 <sub>␣</sub>
a05563264c201047514185b38f6af833cc62122074aba5d217506b2be4f5955e ↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra ↳"8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"	0	625000000 <sub>␣</sub>

```
TX row number: 8
```

```
=====
```

```
Select Collateral UTxO
```

```
=====
```

```
Wallet Name: fees
```

TxHash	TxIx	Amount
c5b11e878a7dcebe5a52eb32eff5d83c3c76e35d13a2106aab811535dff5e3f6 ↳lovelace + TxOutDatumHashNone	0	1000000000 <sub>␣</sub>

```
TX row number: 1
```

```
Transaction successfully submitted.
```

Now, after a minute or so, when we look at the UTxOs in wallet1, we see that we have got some new lovelace.

```
./balance.sh wallet1
```

TxHash	TxIx	Amount
-----		

(continues on next page)

(continued from previous page)

```

060aa2af10655a4b893bb4b828aa2288a5a18f1dd8941a7f99ffe8d3bd1d71f1      1      990000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"
11248966667145ab998933075d4eca6305e96b58e03684c7d1c8100a410c17df      1      625000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"
7760e20988b77005a557e937e493aaf98103ec46e6b9ddc90ed27485bf8602d0      0
↳99706326204 lovelace + TxOutDatumHashNone

```

We did not lose our collateral, although we did get charged some fees.

```

./balance.sh fees
TxHash                                TxIx      Amount
-----
11248966667145ab998933075d4eca6305e96b58e03684c7d1c8100a410c17df      0
↳99999577494 lovelace + TxOutDatumHashNone

```

And the UTxO we spent has gone, but has been replaced with a new UTxO with a reduced balance, but the same datum, leaving us free to take more funds from it later.

```

./contractBalance.sh HelloWorld
TxHash                                TxIx      Amount
-----
099a2a3d025d4e30e95410be19d67e3a27b6c237b378ac8e3f89806d7d1922a7      2      425000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"
0dfec1295895d877edf15f323df63f43aa4501bfc8ee0483512c13550b6f4a65      0      980000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"ee5c9e2778c6c398366c5b9cfd67a888081f7626ca0ac392faca5981e59ff759"
2d9400af7637b05b34e96c66781f087c7a28c7da8b4482b98897807dfe84efd6      0      325000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"3519b7fbee1f70218539524e3b50ba8fa67b6d769cfa6fee4d4356e800342956"
...

```

## HELLOWORLD, BYTESTRINGS AND REDEEMER

**Note:** These instructions should work as presented if you have followed the [AWS Node Setup](#) section and the [Wallets and Funds](#) sections. If not, you may need to improvise a little.

The next thing to run is a script that checks that the datum matches a given person's name and that the redeemer matches a given birthday.

A brief look at some of the code gives an idea as to how this works.

```
person :: PersonDetails
person = PersonDetails { pName = "Sam Jones", pDob = "1974/12/23" }

{-# INLINABLE helloWorld #-}

helloWorld :: PersonDetails -> P.ByteString -> P.ByteString -> ScriptContext -> P.Bool
helloWorld thePerson datum redeemer context =
    pName thePerson P.== datum      P.&&
    pDob thePerson P.== redeemer
```

Examining the code will give an insight into how to use ByteString parameters. It will also show a basic parameterized contract script. Even though its parameters in this case are hard-coded, they still have the effect of generating a different script address for different person names.

The code can be found in `/data/plutus-scripts/plutus-sources/plutus-helloworld/src/Cardano/PlutusExample/HelloWorld`.

We can compile it as follows.

```
cd /data/plutus-scripts/plutus-sources/plutus-helloworld/
cabal run plutus-helloworld-person
```

This time the code will output a file with the name `plutus-helloworld-person.plutus`. We will move this to our scripts directory.

```
mv helloworld-person.plutus ../../scripts/HelloWorldPerson.plutus
```

Then we can pay some funds to this script.

```
cd /data/plutus-scripts
./payToScript.sh 62500000 HelloWorldPerson "\"Sam Jones\"" wallet1
Wallet Name:  wallet1
```

TxHash

TxIx

Amount

(continues on next page)

(continued from previous page)

```
-----
060aa2af10655a4b893bb4b828aa2288a5a18f1dd8941a7f99ffe8d3bd1d71f1      1      99000000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"
11248966667145ab998933075d4eca6305e96b58e03684c7d1c8100a410c17df      1      62500000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"
7760e20988b77005a557e937e493aaf98103ec46e6b9ddc90ed27485bf8602d0      0      L
↳99706326204 lovelace + TxOutDatumHashNone
TX row number: 3
Transaction successfully submitted.
```

I submitted two more transactions to make a total of three - two with the correct datum and one with the wrong datum.

**Note:** When running subsequent transactions, you may be presented with the option to use the same UTxO that was used for a previous transaction. This won't be allowed, so you need to choose a different UTxO, or, if none exist with enough funds, you need to wait for the previous transaction to complete, by which time a new UTxO (the change from the previous transaction) will be available.

```
./payToScript.sh 72500000 HelloWorldPerson "\"Sam Jones\"" wallet1
./payToScript.sh 72500000 HelloWorldPerson "\"Sammy Jones\"" wallet1
```

```
./contractBalance.sh HelloWorldPerson
```

TxHash	TxIx	Amount
48b33ea5694c8b7d65384eb67470bdc28202d7fe211a60045d0b667c795a22b6	0	62500000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"b3c689968968928e5b87c4a74675b85f311c475a011ec2f168261ce0ae85774a"		
ce0b7f4978b7cd6dae6946a1e150964908491583cacb9436085ac52975ee56c8	0	72500000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"b3c689968968928e5b87c4a74675b85f311c475a011ec2f168261ce0ae85774a"		
e81da06411acbf518cd3e988de27455db757ad5dcd39bf403bdc1c173880593d	0	72500000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"7cfec515f56d4413375aa9775f5de15ee60180861e9eaa954bcf9d015054857c"		

Now we have to put some ugly strings on the command line because I'm not very good with *bash*. The third argument is the redeemer.

```
./getFromScript.sh HelloWorldPerson "\"Sam Jones\"" "\"1974/12/23\"" wallet1
```

```
=====
Select Script UTxO
=====
```

TxHash	TxIx	Amount
48b33ea5694c8b7d65384eb67470bdc28202d7fe211a60045d0b667c795a22b6	0	62500000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"b3c689968968928e5b87c4a74675b85f311c475a011ec2f168261ce0ae85774a"		
ce0b7f4978b7cd6dae6946a1e150964908491583cacb9436085ac52975ee56c8	0	72500000L
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra		
↳"b3c689968968928e5b87c4a74675b85f311c475a011ec2f168261ce0ae85774a"		

(continues on next page)

(continued from previous page)

```
e81da06411acf518cd3e988de27455db757ad5dcd39bf403bdc1c173880593d  0  725000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"7cfec515f56d4413375aa9775f5de15ee60180861e9eaa954bcf9d015054857c"
TX row number: 2

=====
Select Collateral UTx0
=====

Wallet Name: fees

      TxHash                                     TxIx      Amount
-----
099a2a3d025d4e30e95410be19d67e3a27b6c237b378ac8e3f89806d7d1922a7  0  889798683
↳lovelace + TxOutDatumHashNone
TX row number: 1
Receiving Wallet: wallet2
Transaction successfully submitted.
```

After waiting a minute, we check that we managed to grab the funds.

```
./contractBalance.sh HelloWorldPerson
TxHash                                     TxIx      Amount
-----
48b33ea5694c8b7d65384eb67470bdc28202d7fe211a60045d0b667c795a22b6  0  625000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"b3c689968968928e5b87c4a74675b85f311c475a011ec2f168261ce0ae85774a"
e81da06411acf518cd3e988de27455db757ad5dcd39bf403bdc1c173880593d  0  725000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"7cfec515f56d4413375aa9775f5de15ee60180861e9eaa954bcf9d015054857c"

./balance.sh wallet1
TxHash                                     TxIx      Amount
-----
060aa2af10655a4b893bb4b828aa2288a5a18f1dd8941a7f99ffe8d3bd1d71f1  1  990000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"9e478573ab81ea7a8e31891ce0648b81229f408d596a3483e6f4f9b92d3cf710"
11248966667145ab998933075d4eca6305e96b58e03684c7d1c8100a410c17df  1  625000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"8fb8d1694f8180e8a59f23cce7a70abf0b3a92122565702529ff39baf01f87f1"
776df51578af3b840edebf1cb7d0466ce7c60e5230ffe461cf017e2c631c3de7  0  _
↳99498320857 lovelace + TxOutDatumHashNone
8fffaa8ca32ff2af2e984a85ece19efaf5aebac5527e0954ca009cb54954acc4  1  725000000
↳lovelace + TxOutDatumHash ScriptDataInAlonzoEra
↳"b3c689968968928e5b87c4a74675b85f311c475a011ec2f168261ce0ae85774a"
```

If you pass an incorrect datum, you will receive an error message and the transaction will not enter validation.

```
./getFromScript.sh HelloWorldPerson "\"Sammy Jones\"" "\"1974/12/23\"" wallet1
...
...
Command failed: transaction submit Error: Error while submitting tx:
↳ShelleyTxValidationError ShelleyBasedEraAlonzo (ApplyTxError [UtxowFailure
↳(MissingRequiredDatums (fromList [SafeHash
↳"b3c689968968928e5b87c4a74675b85f311c475a011ec2f168261ce0ae85774a"])) (fromList
↳[SafeHash "d658ccd4fce5643c6186657cc2f88f2d110acb88c8b94cd90d9acb088562a19a"])))
```





## MINTING TOKENS

---

**Note:** These instructions should work as presented if you have followed the [AWS Node Setup](#) section and the [Wallets](#) and [Funds](#) sections. If not, you may need to improvise a little.

---

We will mint a native token.

### 17.1 Create Policy

First, we will create a policy that allows a new wallet, `minter` to mint tokens.

```
./createPolicy.sh minter
```

This will create `minter.skey` and `minter.vkey` in the `wallets` directory, and `minter.script` in the `policies` directory.

### 17.2 Fund a Minting Wallet

Next, we will add some funds to the `wallet2` wallet.

```
./sendFromWallet.sh main
Wallet Name: main
TxHash                                TxIx      Amount
-----
b5e89af6cc624465c169349c5217fac20d6c9193a2df04ae63bd8a21f15d5ece    0          ₿
↳ 899899659010 lovelace + TxOutDatumHashNone
TX row number: 1
Lovelace to send: 10000000000
Receiving wallet name: wallet2
Transaction successfully submitted.
```

Then check the balance until it arrives.

```
./balance.sh wallet2
TxHash                                TxIx      Amount
-----
fbc3d7e3d3b802eaa2a1e9f4df2a017796abb2c7dd83a476f04f6b92391adea8    1          10000000000 ₿
↳ lovelace + TxOutDatumHashNone
```

## 17.3 Mint Tokens

Now, we will mint 100 Chess tokens in wallet2.

```
./mint.sh minter Chess 100 wallet2
1000000000
Wallet Name: wallet2
```

	TxHash	TxIx	Amount
-----	fbe3d7e3d3b802eaa2a1e9f4df2a017796abb2c7dd83a476f04f6b92391adea8	1	1000000000
↳ lovelace + TxOutDatumHashNone			

```
TX row number: 1
Transaction successfully submitted.
```

This mint.sh script builds and executes the following commands.

```
$CARDANO_CLI transaction build \
--tx-in ${FROM_UTXO} \
--tx-out ${FROM_WALLET_ADDRESS}+$TOKEN_COUNT+"$TOKEN_COUNT ${POLICY_ID}.${COIN_NAME}" \
--change-address=${FROM_WALLET_ADDRESS} \
--mint="$TOKEN_COUNT ${POLICY_ID}.${COIN_NAME}" \
--mint-script-file="./policies/$POLICY_NAME.script" \
--testnet-magic ${TESTNET_MAGIC_NUM} \
--out-file tx.build \
--witness-override 2 \
--alonzo-era

$CARDANO_CLI transaction sign \
--tx-body-file tx.build \
--signing-key-file ./wallets/${FROM_WALLET_NAME}.skey \
--signing-key-file ./wallets/${POLICY_NAME}.skey \
--out-file tx.signed

$CARDANO_CLI transaction submit --tx-file tx.signed --testnet-magic $TESTNET_MAGIC_NUM
```

Then we can wait for the tokens to arrive.

```
./balance.sh wallet2
```

TxHash	TxIx	Amount
-----		
2064de58563aab461f4deb9a8414558699361252b5a655d7c0ad23e1d90595fd	0	899822003
↳ lovelace + TxOutDatumHashNone		
2064de58563aab461f4deb9a8414558699361252b5a655d7c0ad23e1d90595fd	1	1000000000
↳ lovelace + 1000000000 a5bdebd0371758aeeb3b116432724fc6bf6c9caf186485baee7ee4d9.Chess +		
↳ TxOutDatumHashNone		

We can use the burn.sh script to burn some of the tokens in a UTxO.

First, create the wallet.

```
./createWallet.sh bernie
```

Then add some Ada.

```
./sendFromWallet.sh main
Wallet Name: main
```

TxHash	TxIx	Amount
d22a8f5fc2bac7b851f14899e9af0556e394d884d0cddf2b2e29f76866a3310f	1	1000000000
↳ lovelace + 1000000000 be40ec5a43dc88640f8ce1b7262b32918581ff9d6891d94bf4315ba5 +		
↳ TxOutDatumHashNone		
f66a19065873436b0d131493f544ed6e7287acc65d3627d6551bcdae39685fdd	0	
↳ 888899324928 lovelace + TxOutDatumHashNone		

TX row number: 2  
Lovelace to send: 10000000000  
Receiving wallet name: bernie  
Transaction successfully submitted.

Wait a while for it to arrive.

```
./balance.sh bernie
```

TxHash	TxIx	Amount
a24f7d302ff49974463638259df93010087e0067edee11171072322e7b3e2123	1	
↳ 100000000000 lovelace + TxOutDatumHashNone		

Then mint 100 Rook tokens.

```
/mint.sh minter Rook 100 bernie
Wallet Name: bernie
```

TxHash	TxIx	Amount
a24f7d302ff49974463638259df93010087e0067edee11171072322e7b3e2123	1	
↳ 100000000000 lovelace + TxOutDatumHashNone		

TX row number: 1  
Transaction successfully submitted.

Wait for it to arrive.

```
./balance.sh bernie
```

TxHash	TxIx	Amount
7e9cdb8f12c88100a431537ed8274d021f29d11a07ca5f31c66702f2e94f46cd	0	9899824555
↳ lovelace + TxOutDatumHashNone		
7e9cdb8f12c88100a431537ed8274d021f29d11a07ca5f31c66702f2e94f46cd	1	1000000000
↳ lovelace + 1000000000 a5bdebd0371758aeeb3b116432724fc6bf6c9caf186485baee7ee4d9.Rook +		
↳ TxOutDatumHashNone		

Then burn 10 Rook tokens.

```
./burn.sh minter Rook 10 bernie
100000000
Wallet Name: bernie
```

TxHash	TxIx	Amount
7e9cdb8f12c88100a431537ed8274d021f29d11a07ca5f31c66702f2e94f46cd	0	9899824555
↳ lovelace + TxOutDatumHashNone		
7e9cdb8f12c88100a431537ed8274d021f29d11a07ca5f31c66702f2e94f46cd	1	1000000000
↳ lovelace + 1000000000 a5bdebd0371758aeeb3b116432724fc6bf6c9caf186485baee7ee4d9.Rook +		
↳ TxOutDatumHashNone		

(continued on next page)

(continued from previous page)

TX row number: 2  
Transaction successfully submitted.

And, once the transaction has been executed, we will see the bernie wallet has only 90 of the Rook tokens left.

```
./balance.sh bernie
TxHash                                TxIx      Amount
-----
7e9cdb8f12c88100a431537ed8274d021f29d11a07ca5f31c66702f2e94f46cd    0          9899824555
↳ lovelace + TxOutDatumHashNone
894154c7322ef79a7f94db08066b00eb842b9636d3bf712a4f0add7c7fbf303    0          89824555
↳ lovelace + TxOutDatumHashNone
894154c7322ef79a7f94db08066b00eb842b9636d3bf712a4f0add7c7fbf303    1          10000000
↳ lovelace + 90000000 a5bdebd0371758aeb3b116432724fc6bf6c9caf186485baee7ee4d9.Rook +
↳ TxOutDatumHashNone
```

## INSTALL THE CARDANO NODE

I started with two fresh `t2.large` AWS Ubuntu EC2 instances using AMI `ami-0ff4c8fb495a5a50d`, each with a 60Gb data volume.

Perform these steps on each of the instances.

### 18.1 Mount the data volume

```
sudo mkfs -t xfs /dev/xvdh
sudo mkdir /data
sudo mount /dev/xvdh /data
sudo chown ubuntu:ubuntu /data
```

### 18.2 Setup some environment variables

```
echo "export PATH=\"~/.local/bin:$PATH\"" >> ~/.bashrc
echo "export LD_LIBRARY_PATH=\"/usr/local/lib:$LD_LIBRARY_PATH\"" >> ~/.bashrc
echo "export PKG_CONFIG_PATH=\"/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH\"" >> ~/.bashrc
echo "export CARDANO_NODE_SOCKET_PATH=\"/data/Pool/node/db.socket\"" >> ~/.bashrc
echo "export CABAL_VERSION=3.2.0.0" >> ~/.bashrc
echo "export CARDANO_TAG=1.29.0" >> ~/.bashrc
echo "export GHC_VERSION=8.10.2" >> ~/.bashrc

source ~/.bashrc
```

### 18.3 Install some dependencies

```
sudo apt-get update -y
sudo apt-get install automake build-essential pkg-config libffi-dev libgmp-dev libssl-
↳ dev libtinfo-dev libsystemd-dev zlib1g-dev make g++ tmux git jq wget libncursesw5
↳ libtool autoconf -y
```

## 18.4 Install Cabal

```
mkdir /data/Downloads
cd /data/Downloads

wget https://downloads.haskell.org/~cabal/cabal-install-$CABAL_VERSION/cabal-install-
↳$CABAL_VERSION-x86_64-unknown-linux.tar.xz
tar -xf cabal-install-$CABAL_VERSION-x86_64-unknown-linux.tar.xz
rm cabal-install-$CABAL_VERSION-x86_64-unknown-linux.tar.xz cabal.sig

mkdir -p ~/.local/bin
mv cabal ~/.local/bin/
cabal update
```

## 18.5 Install GHC

```
wget https://downloads.haskell.org/ghc/$GHC_VERSION/ghc-$GHC_VERSION-x86_64-deb9-linux.
↳tar.xz
tar -xf ghc-$GHC_VERSION-x86_64-deb9-linux.tar.xz
rm ghc-$GHC_VERSION-x86_64-deb9-linux.tar.xz
cd ghc-$GHC_VERSION
./configure
sudo make install
```

## 18.6 Install libsodium

```
cd /data/Downloads
git clone https://github.com/input-output-hk/libsodium
cd libsodium
git checkout 66f017f1

./autogen.sh
./configure
make
sudo make install
```

## 18.7 Build the Cardano node

```
cd /data
git clone https://github.com/input-output-hk/cardano-node.git
cd cardano-node
git fetch --all --tags
git checkout tags/$CARDANO_TAG
cabal build all
```

## 18.8 Copy the binaries

```
cp -p dist-newstyle/build/x86_64-linux/ghc-$GHC_VERSION/cardano-node-$CARDANO_TAG/x/  
↳ cardano-node/build/cardano-node/cardano-node ~/.local/bin/  
cp -p dist-newstyle/build/x86_64-linux/ghc-$GHC_VERSION/cardano-cli-$CARDANO_TAG/x/  
↳ cardano-cli/build/cardano-cli/cardano-cli ~/.local/bin  
cardano-cli --version
```

## 18.9 Get the config files

```
cd /data/Pool  
mkdir node  
cd node  
  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/testnet-config.json  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/testnet-shelley-genesis.json  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/testnet-byron-genesis.json  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/testnet-topology.json  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/mainnet-config.json  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/mainnet-byron-genesis.json  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/mainnet-shelley-genesis.json  
wget https://hydra.iohk.io/job/Cardano/cardano-node/cardano-deployment/latest-finished/  
↳ download/1/mainnet-topology.json
```