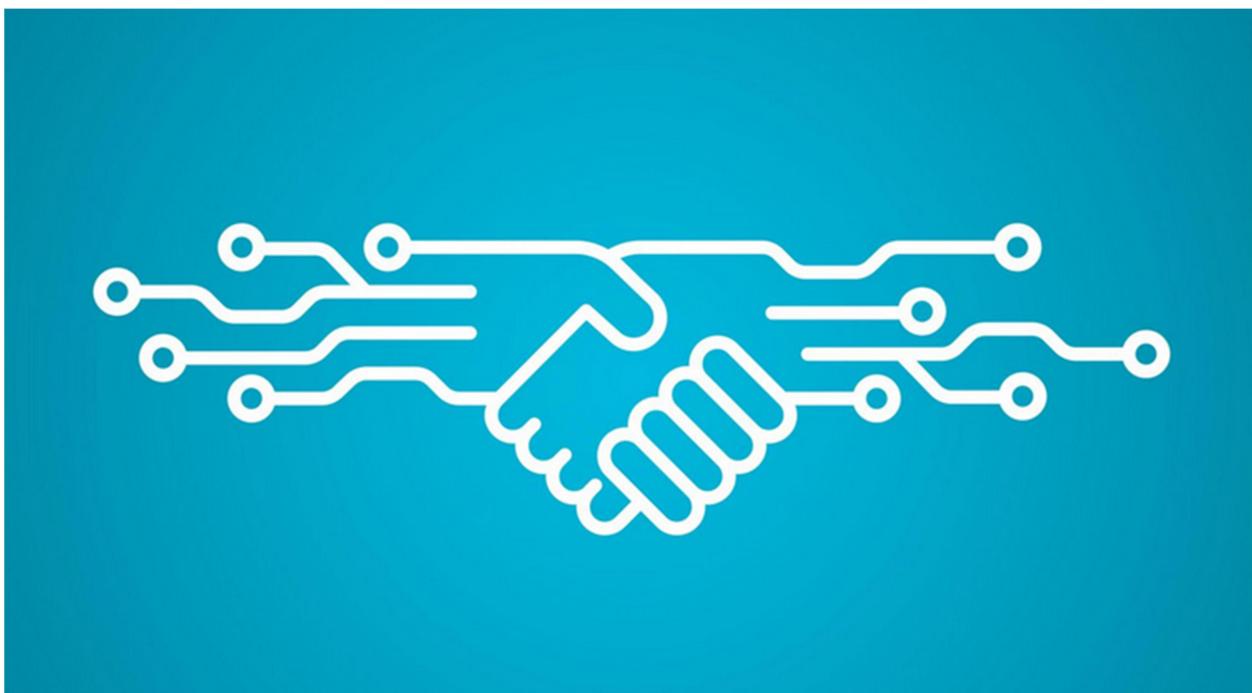


Plutus: Learning a smart-contract language



IOHK

Plutus: Learning a smart-contract language

Copyright © April 2022, IOHK (Input Output Hong Kong)

This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/4.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

The initial draft of this book was prepared by Luka Kurnjek. Majority of the text is taken from the Plutus pioneer program that was presented by Lars Brünjes. All program code in this book is the copyright of IOHK. The Plutus pioneer program is freely available at IOHK github page:

<https://github.com/input-output-hk/plutus-pioneer-program>

All pictures in this book are the copyright of IOHK if not otherwise noted. The picture on the front page is owned by cornellilj.org and published on following blog post:

<https://cornellilj.org/2018/02/08/smart-contracts-another-feather-in-uncitrals-cap/>

Table of Contents

1 Plutus introduction	5
1.1 Installation of Haskell and Plutus	5
1.2 Running the Plutus playground.....	7
1.3 The EUTxO model.....	10
1.4 Plutus code	12
2 Simple validation scripts	13
2.1 Low-level untyped validation scripts.....	13
2.2 High Level typed validation scripts.....	17
2.3 Homework.....	20
3 Time, parameterized contracts and the cardano testnet.....	22
3.1 Time.....	23
3.2 Parameterized Contracts.....	29
3.3 The Cardano testnet.....	33
3.4 Homework.....	38
4 Monads, Traces & Contracts	45
4.1 Monads.....	45
4.2 The emulator trace monad.....	52
4.3 The contract monad	54
4.4 Homework.....	58
5 Native tokens	61
5.1 Simple Minting Policy	63
5.2 More Realistic Minting Policy.....	65
5.3 NFT-s.....	67
5.4 Homework.....	70
6 Deployment.....	75
6.1 The minting policy	75
6.2 Minting with the CLI	76

6.3 Deployment Scenarios	79
6.4 The Contracts	80
6.5 Minting with the PAB	86
7 State Machines.....	93
7.1 Commit schemes	93
7.2 Implementation without State Machines	94
7.3 State Machines.....	109
7.4 Homework.....	124
8 Testing	131
8.1 State Machine Example: Token Sale	131
8.2 Automatic testing using emulator traces	140
8.3 Test Coverage.....	143
8.4 Interlude optics	144
8.5 Property based testing with QuickCheck	146
8.6 Property based testing of Plutus contracts	148
8.7 Homework.....	160
9 Resources.....	165

1 Plutus introduction

Plutus is the native smart contract language for Cardano. It is a Turing-complete language written in Haskell, and Plutus smart contracts are effectively Haskell programs. By using Plutus, you can be confident in the correct execution of your smart contracts. It draws from modern language research to provide a safe, full-stack programming environment based on Haskell, the leading purely-functional programming language. [2]

So in order to understand this book and its code examples one has to understand the basics of the Haskell programming language. The Haskell project contains a list of learning resources as books and tutorials under its Documentation section, from which some are free and some commercial: <https://www.haskell.org/documentation/>.

1.1 Installation of Haskell and Plutus

The installation instructions are written for a Unix style OS (e.g. Mac OS or Linux). In order to install the Haskell toolchain, which is composed of GHC (Glasgow Haskell compiler), Cabal and some other tools, you can use GHcup: <https://www.haskell.org/ghcup/>. You will need curl installed on your OS, before you can use the GHcup installation.

After the installation is completed check from a terminal if you can run GHCi, the GHC Repl:

```
[user@fedora ~]$ ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

The GHCi version you will see can of course be different and should not matter as long as it is above or the same as the version shown in this example.

To run Plutus code examples from this book, you will need the following GIT repositories:

- <https://github.com/input-output-hk/plutus-apps>
- <https://github.com/lukakurnjek/plutus-pioneer-program>

You will have to clone them with the git tool, that you will also need to install on your OS. Roughly speaking the plutus-apps repository contains the “off-chain” code for Plutus, that runs in a Wallet. It also references another repository called plutus, which contains the “on-chain” code for Plutus that runs on the Cardano blockchain.

To be able to build the code you will need to install the nix command line toolset. To install nix you can follow the instructions from their web-page: <https://nixos.org/download.html>.

Once nix is installed, you have to add the IOHK binary caches. You can do this by editing the /etc/nix/nix.conf, where you add the following lines:

```
substituters = https://hydra.iohk.io https://iohk.cachix.org https://cache.nixos.org/
trusted-public-keys = hydra.iohk.io:f/Ea+s+dFdN+3Y/G+FDgSq+a5NEWhJGzdjvKNGv0/EQ=
iohk.cachix.org-1:DpRUyj7h7V830dp/i6Nti+NE02/nhblbov/8MW7Rqoo= cache.nixos.org-
1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY=
```

If you don't have an /etc/nix/nix.conf or don't want to edit it, you may add the nix.conf lines to ~/.config/nix/nix.conf instead. You must be a trusted user to do this. If you are running NixOS you can, set the following NixOS options:

```
nix = {
  binaryCaches = [ "https://hydra.iohk.io" "https://iohk.cachix.org" ];
  binaryCachePublicKeys =
  [ "hydra.iohk.io:f/Ea+s+dFdN+3Y/G+FDgSq+a5NEWhJGzdjvKNGv0/EQ=" "iohk.cachix.org-
  1:DpRUyj7h7V830dp/i6Nti+NE02/nhblbov/8MW7Rqoo=" ];
};
```

The above nix configuration instructions are kept up-to-date in the plutus-apps repository [4].



A lot of dependencies are cached there and it will make the Plutus builds much faster.

The example code in this book comes directly from the plutus-pioneer-program git repository mentioned earlier. To build the code that is contained in the week folders follow these steps:

1. Open up a terminal and cd into a week folder of the plutus-pioneer-program repo, e.g. week01. You will need to figure out which commit of the plutus-apps repository this week uses. To do this open the cabal.project file, which contains various dependencies and scroll to the section *source-repository-package*:

```
source-repository-package
type: git
location: https://github.com/input-output-hk/plutus-apps.git
tag: 41149926c108c71831cf8d244c83b0ee4bf5c8a
```

2. Copy the commit under the tag section. cd into the plutus-apps repository and checkout the commit you just have copied:

```
[user@fedora ~/plutus-apps]$ git checkout 41149926c108c71831cf8d244c83b0ee4bf5c8a
```

3. Now run the command nix-shell. When you do this for the first time it can take a while until everything has built. After the build your command prompt will change to the nix shell. In this shell cd back into the week01 folder and run the cabal build command:

```
[nix-shell: ~/plutus-pioneer-program/code/week01]$ cabal build
```

4. This can also take some time if you build it the first time. When the build has finished you can start the GHC Repl with the *cabal repl* command:

```
[nix-shell: ~/plutus-pioneer-program/code/week01]$ cabal repl
Build profile: -w ghc-8.10.4.20210212 -O1
Preprocessing library for plutus-pioneer-program-week01-0.1.0.0..
GHCi, version 8.10.4.20210212: https://www.haskell.org/ghc/ :? for help
Ok, one module loaded.
Prelude Week01.EnglishAuction>
```

5. To leave the cabal repl simply type “:q”.

Another useful thing to have is the documentation for various Plutus libraries. You can build the documentation yourself. From inside the nix-shell in the plutus-apps folder run:

```
[nix-shell:~/plutus-apps]$ build-and-serve-docs
Serving HTTP on 0.0.0.0 port 8002 (http://0.0.0.0:8002/) ...
```

You can open the displayed address and port in your web-browser and will see the high-level documentation for Plutus. A more useful thing is the Plutus library documentation which you will find at this location: <http://0.0.0.0:8002/haddock>. Because is not just a static file but an actual web-server you can search through it by pressing Ctrl+S.

1.2 Running the Plutus playground

The Plutus playground is an interactive environment where you can compile your Plutus code and simulate it. In the simulation you can specify:

- how many wallets with how many ADA (native currency of Cardano) you want to have
- which wallet actions also called a transactions, you want to perform at which timeslots
- what are the input parameters for your transactions if there are any input fields present

These transactions will then be executed on the playground and you can view them interactively. To set up the playground perform the following steps:

1. cd into the plutus-apps repository and start the nix-shell.
2. Then cd into *plutus-playground-client/* folder and start the *plutus-playground-server*:

```
[nix-shell:~/plutus-apps/plutus-playground-client]$ plutus-playground-server
```

NOTE: The default server timeout is set to 80 seconds. If you want to increase the server timeout to e.g. 120 seconds you can pass a “-i 120s” argument to the command.

3. Open up another nix shell at the same location and start the playground client:

```
[nix-shell:~/plutus-apps/plutus-playground-client]$ npm start
...
Project is running at https://localhost:8009/
```

If you then go to your web-browser and open <https://localhost:8009/> you will see the Plutus playground (Figure 1). In the middle you will see the editor window, where you can copy paste the code from the plutus-pioneer-program repository. You can delete the default example code. On the right side you see the compile and simulate buttons. With the compile button you compile the code and a status bar at the bottom of the editor will tell you if the compilation succeeded. If not it will throw an error pointing to the line that makes trouble.

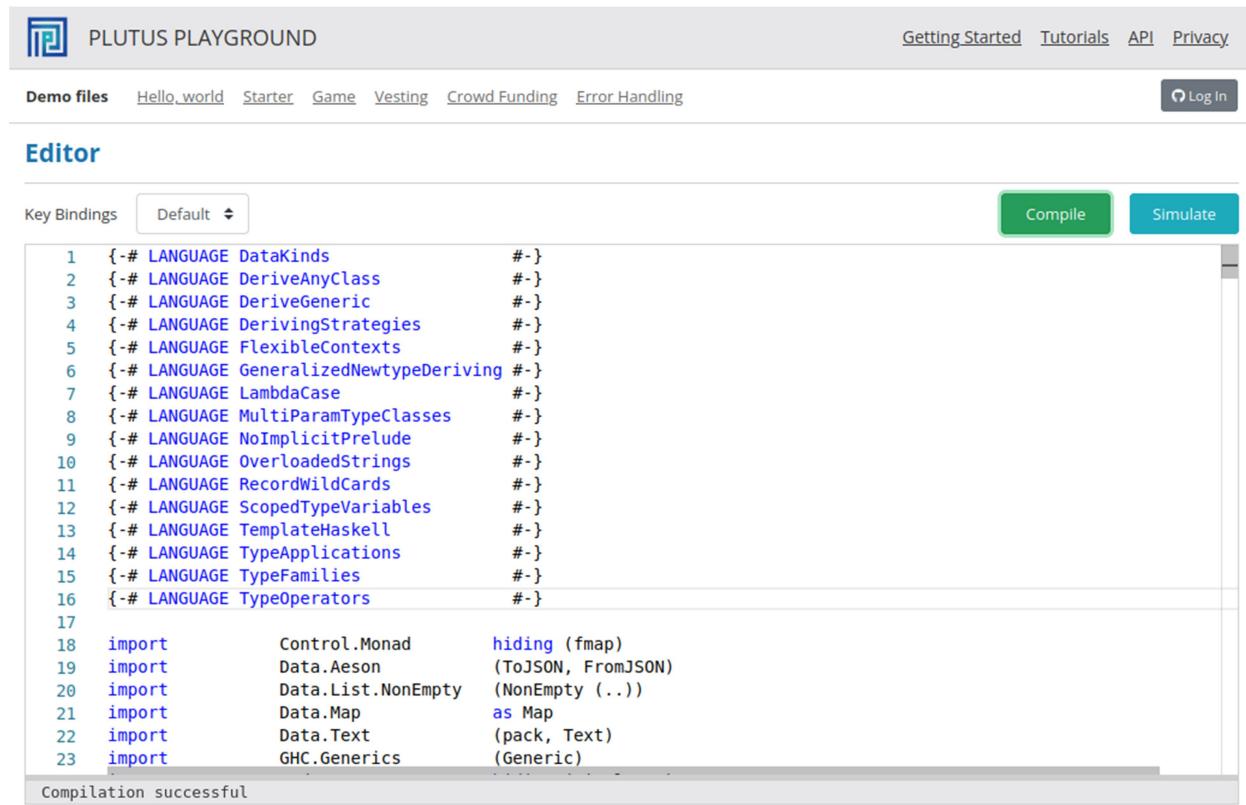


Figure 1 - Plutus playground editor

After the compilation successfully succeeded you can use the simulate button to open up the Simulate view (Figure 2). There you can add wallets, set amounts of Lovelace (= 0.000001 ADA) for initial funds of the wallets, trigger the available wallet functions and add wait actions. Once you are finished with defining your actions you can click on the Evaluate button. The transaction window will appear (Figure 3). There you will always see a genesis transaction. If there were actions defined, there will also be other transactions. You can then click on different slots in the upper row and the transactions for these slots will appear.

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	100000000
T	100000000

Available functions

bid	+	close	+	start	+
Pay to Wallet					

Wallet 2 ×

Opening Balances

Lovelace	100000000
T	100000000

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.

+
Add Wait Action

Evaluate
Transactions

Figure 2 - Putus playground simulation

Transactions
x

Blockchain

Click a transaction for details

Slot 0, Tx 0

Inputs
Transaction
Outputs

Slot 0, Tx 0

Tx: d8e1263889cf12a70c756adc03c7fd14437eec240f2b8e8f... copy

Validity: All time
Signatures:None

Forge

Ada	Lovelace	200000000
66		
T	T	200000000

Wallet 2

PubKeyHash 80a4f45b...

Ada	Lovelace	100000000
66		
T	T	100000000
Unspent		

Wallet 1

PubKeyHash a2c20c77...

Ada	Lovelace	100000000
66		
T	T	100000000
Unspent		

Figure 3 - Putus playground transactions



You can also use the online Plutus playground (<https://playground.plutus.iohkdev.io>) to compile your code. But it is not necessarily up-to-date with the code from the git repositories, so some Plutus code from the examples in this book may not compile.

The Auction example code in the folder week01 will not be covered here, since it is a too advanced example to start with. But you can look at the demonstration of this code in the videos “Auction Contract in the EUTxO-Model” and “Auction Contract on the Playground” that can both be found in the plutus-pioneer-program git repository under the Lecture #1 chapter.

1.3 The EUTxO model

The Cardano blockchain uses the Extended UTXO model that is a variant of the Unspent Transaction Output (UTXO) model used by Bitcoin. Transactions consume unspent outputs (UTXOs) from previous transactions and produce new outputs which can be used as inputs to later transactions. Unspent outputs are the liquid funds on the blockchain. Users do not have individual accounts, but rather have a software wallet on a smartphone or PC which manages UTXOs on the blockchain and can initiate transactions involving UTXOs owned by the user. Every core node on the blockchain maintains a record of all of the currently unspent outputs, the UTXO set. When outputs are spent, they are removed from the UTXO set. [3]

There are other models than UTXO. Ethereum, for example, uses a so-called account-based model, which is what a normal bank uses, where everybody has an account and each account has a balance. If you transfer money from one account to another, then the balances get updated accordingly. But in the UTXO model, the input is always the entire balance of an UTXO and the outputs are newly created UTXOs from which one of them could be belonging to the user that provided his UTXO as input and would represent his change amount. For every transaction there is a fee to pay denominated in ADA for the Cardano blockchain.

As soon as an output is used as input in a transaction, it becomes spent and can never be used again. The output is specified by an address which is represented by a public key hash and a value which represents the ADA amount and optionally native tokens. We call them public key addresses. An output’s address determines which transactions are allowed to ‘unlock’ the output and use it as an input. A transaction must be signed by the owner of the private key corresponding to the address that defines the input UTXO. Think of an address as a ‘lock’ that can only be ‘unlocked’ by the right ‘key’ – the correct signature. [2]

The extended UTXO model introduces in addition to public key addresses also script addresses, that can contain some arbitrary logic, that defines under which conditions the UTXO can be

spent. The address is unlocked by an arbitrary piece of data called the redeemer, which in the conventional UTXO model would be a private key. A UTXO also contains some arbitrary data called the datum, beside the amount of ADA sitting at the address. The datum together with the redeemer and the transaction context are the input information for a script that then chooses whether this transaction is valid and can be processed by a node on the network.

You can check the validity of a transaction in your wallet. If it is valid you can be sure it will be processed on the network, given the condition that all the UTXO inputs are still present at processing time. If they are not the transaction will simply fail and no fee will be charged to the user that sent the transaction. We call the script that validates a transaction the validator.

A transaction can be classified as a spending transaction that spends UTXOs or as a producing transaction that produces UTXOs. The truth is every transaction except the genesis transaction takes at least one UTXO as input and produces at least one UTXO as output. What we mean by the terms spending and producing is the context. Later in chapter 2 we will see code examples where we have a give function and a grab function. The give function gives some ADA to a script address and is for this reason a producing transaction. The grab function wants to collect ADA from a script address and is a spending transaction.

The producing transaction only has to include the hash of the script and the hash of the datum that both belong to the output UTXO. But optionally, it can include the datum and the script as well. If it does not, only a person that knows the datum by some other means not by looking at the blockchain would be able to ever spend such an output.

The spending transaction is responsible for providing the datum, the redeemer and the transaction context. It also provides the validator script. The script address is defined as a hash of the validator code written in Plutus core language, which also needs to be provided. The script addresses are publicly known.

As said the validator script takes the datum, the redeemer and the transaction context as input information. The input for the datum comes from each UTXO individually that are sitting at the script address. The datum can also be defined in the transaction context which is provided by the spending transaction. This makes it possible for the validator script to compare information from both datums and perform some arbitrary logic on them.

This limited view of the validator script that can see only inputs, outputs and the transaction that will be processed, has a security advantage compared to the Ethereum model, where the script can see the whole state of the blockchain. That enables Ethereum's scripts to be much more powerful but for this reason it's also very difficult to predict what a given script will do and that opens the door to all sorts of security issues. It can be mathematically proven that

every logic you can express in Ethereum you can also express in the extended UTxO model. And that makes it a much more safer and reliable transaction model compared to Ethereum.

1.4 Plutus code

The code for Plutus smart contracts is separated into two. First is the “on-chain” code, which consist of the validator function and some additional declarations and variables as the script address. This code gets compiled to Plutus Core language. It runs on the Cardano blockchain and once submitted it cannot be changed.

From the official documentation [2] we get the following description for Plutus Core:

Plutus Core is the scripting language used by Cardano to implement the EUTXO model. It is a simple, functional language similar to Haskell, and a large subset of Haskell can be used to write Plutus Core scripts. As a smart contract author, you don’t write any Plutus Core; rather, all Plutus Core scripts are generated by a Haskell compiler plugin called Plutus Tx.

The “off-chain” code is written in Haskell, just like the on-chain code, unlike Ethereum where the on-chain code is written in Solidity, but the off-chain code is written in JavaScript. That way, the business logic only needs to be written once. This logic can then be used in the validator script and in the code that builds the transactions that run the validator script. [2]

The off-chain code basically constructs the transaction and submits it to the blockchain. Since both the on-chain and off-chain code are written in Haskell they can reside in one Haskell file while testing your code, which allows them to share code between them.

2 Simple validation scripts

We said earlier that a script sitting on a UTXO address takes in 3 parameters: the datum, the redeemer and the transaction context, which is the submitted transaction with all the inputs and outputs. In the low-level implementation of Plutus these 3 parameters are represented with the same data type. In the high-level implementation you can use custom Haskell data types for datum and redeemer and a predefined type for the transaction context. You can use both of the implementations in your smart-contract code. The difference between them is code performance which is better for the low-level implementation data types.

The data type for the low-level implementation is called *BuiltinData*. It contains two conversion functions *builtinDataToData* and *dataToBuiltinData*, that can convert back and forth to the *Data* type. The Data type has its constructor exposed and has the following definition:

Constructors

Constr	Integer	[Data]
Map	[(Data, Data)]	
List	[Data]	
I	Integer	
B	ByteString	

Figure 4 - Data constructor

Both data types are contained in the *PlutusTx* module. To be able to assign a string value to the B ByteString constructor you need to import the language extension *OverloadedStrings*.

2.1 Low-level untyped validation scripts

Lets look at the code from the *Gift.hs* file in the week02 folder.

```
1 {-# LANGUAGE DataKinds      #-}
2 {-# LANGUAGE FlexibleContexts #-}
3 {-# LANGUAGE NoImplicitPrelude #-}
4 {-# LANGUAGE ScopedTypeVariables #-}
5 {-# LANGUAGE TemplateHaskell   #-}
6 {-# LANGUAGE TypeApplications  #-}
7 {-# LANGUAGE TypeFamilies     #-}
8 {-# LANGUAGE TypeOperators    #-}
9
```

```

10 module Week02.Gift where
11
12 import           Control.Monad      hiding (fmap)
13 import           Data.Map          as Map
14 import           Data.Text         (Text)
15 import           Data.Void         (Void)
16 import           Plutus.Contract
17 import           PlutusTx          (Data(..))
18 import qualified PlutusTx
19 import qualified PlutusTx.Builtin as Builtins
20 import           PlutusTx.Prelude  hiding (Semigroup(..), unless)
21 import           Ledger             hiding (singleton)
22 import           Ledger.Constraints as Constraints
23 import qualified Ledger.Scripts   as Scripts
24 import           Ledger.Ada        as Ada
25 import           Playground.Contract (printJson, printSchemas,
26                                       ensureKnownCurrencies, stage)
27 import           Playground.TH      (mkKnownCurrencies, mkSchemaDefinitions)
28 import           Playground.Types  (KnownCurrency(..))
29 import           Prelude            (IO, Semigroup(..), String)
30 import           Text.Printf        (printf)
31 {-# OPTIONS_GHC -fno-warn-unused-imports #-}
32
33 {-# INLINABLE mkValidator #-}
34 mkValidator :: BuiltinData -> BuiltinData -> BuiltinData -> ()
35 mkValidator _ _ _ = ()
36
37 validator :: Validator
38 validator = mkValidatorScript $$ (PlutusTx.compile [|| mkValidator ||])
39
40 valHash :: Ledger.ValidatorHash
41 valHash = Scripts.validatorHash validator
42
43 scrAddress :: Ledger.Address
44 scrAddress = scriptAddress validator
45
46 type GiftSchema =
47     Endpoint "give" Integer
48     .\`/ Endpoint "grab" ()
49
50 give :: AsContractError e => Integer -> Contract w s e ()
51 give amount = do
52     let tx = mustPayToOtherScript valHash (Datum $ Builtins.mkI 0) $
53         Ada.lovelaceValueOf amount

```

```

53     ledgerTx <- submitTx tx
54     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
55     logInfo @String $ printf "made a gift of %d lovelace" amount
56
57 grab :: forall w s e. AsContractError e => Contract w s e ()
58 grab = do
59     utxos <- utxosAt scrAddress
60     let orefs   = fst <$> Map.toList utxos
61         lookups = Constraints.unspentOutputs utxos      <>
62                   Constraints.otherScript validator
63         tx :: TxConstraints Void Void
64         tx      = mconcat [mustSpendScriptOutput oref $ Redeemer $ 
65                           Builtins.mkI 17 | oref <- orefs]
66     ledgerTx <- submitTxConstraintsWith @Void lookups tx
67     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
68     logInfo @String $ "collected gifts"
69
70 endpoints :: Contract () GiftSchema Text ()
71 endpoints = awaitPromise (give' `select` grab') >> endpoints
72 where
73     give' = endpoint @"give" give
74     grab' = endpoint @"grab" $ const grab
75
76 mkSchemaDefinitions ''GiftSchema
77 mkKnownCurrencies []

```

There are various language pragmas added in the beginning. One that is worth of noticing is the *NoImplicitPrelude* extension which allows you to use a custom prelude. That being said some standard Haskell functions you are used to may not work since we are importing *Plutus.Prelude* in the import section. To get a list of all functions of the custom prelude you can search the Plutus library documentation that you have learned to build for the keyword *Plutus.Prelude*.

In our code we first define the validator function called *mkValidator*. It accepts the datum, the redeemer and the context. We chose to use the *BuiltinData* datatype. Because of that the output of the validator is the unit (). We said earlier the validator only validates a given transaction and for that reason you would expect a Bool value as return type. This is the case if we use the high-level implementation. For the low-level we have instead the unit, which is returned if the transaction passed or the error type, that gets returned if it fails.

In our case the validation passes no matter the input arguments, which means that everyone can process this UTXO and take some ADA from it if it contains any. For this reason we call this

example gift. Next we want to define our *validator*. To get it we need to compile the validator to Plutus Core script. This is done in line 38.

To make use of the *mkValidatorScript* function we need to import *Ledger.Script*. The *compile* function from the *PlutusTx* module takes as input a syntax tree of a function which we can get if we put the oxford brackets `[/] mkValidator [/]` around our desired function. The *compile* function produces another syntax tree that is written in the Plutus core language. Then the `$$` symbol called *splice*, takes a syntax tree and splices it back to Haskell source code, which is what we need for input to our *mkValidatorScript* function. If you want to be able to call external helper functions from the validator function you need to add the *INLINABLE* pragma statement before the validator function (33).

Next we create the validator hash and the script address which contains the validator hash and staking information that is used when we stake to a stake pool. The *scriptAddress* function is contained in the module *Ledger.Address*. The definitions until now represent the on-chain code. Everything after that is the off-chain code.

First we define the endpoints that the user can use to interact with the blockchain from within his wallet. Our give endpoint takes an integer parameter, which will represent the amount of ADA we are willing to give and grab does not take any parameter because it just spends funds. Then we define the give and grab functions. For the give function we first define the transaction (52). The transaction says a certain amount of lovelace should be paid to the specified address and with this datum. Line 53 submits the transaction, line 54 awaits confirmation of the transaction and line 55 logs the provided information which can be seen on the playground.

For the grab function we first lookup all the UTXOs sitting at a given address (59). Then we get all the references of the UTXOs (60). Next we define lookups in order to tell the wallet how to construct the transaction (61). First we tell that we are looking for unspent outputs and second we provide the validator script. Then we define the transaction so that it consumes all the UTXOs (64). When we submit the transaction we provide the lookups in order to know how to find the UTXOs. Next we await for confirmation and after that we log a message.

In the endpoint function we give the user the choice of selecting between the 2 endpoints and then recourse to do the same again. In line 75 we generate the schema definition and in line 76 we call the *mkKnownCurrencies* function so that in the playground we have ADA available.

Let's look now at an example where the validator function fails. In order that we can use the *traceError* function we need also to import the *OverloadedStrings* language extension.

```
{-# LANGUAGE OverloadedStrings #-}  
mkValidator :: BuiltinData -> BuiltinData -> BuiltinData -> ()  
mkValidator _ _ _ = traceError "BURNT!"
```

Now the validation will fail and the Grab transaction will not be processed. For our next example we will take into account the redeemer (example *FortyTwo.hs*).

```
mkValidator :: BuiltinData -> BuiltinData -> BuiltinData -> ()
mkValidator _ r _
| r == Builtins.mkI 42 = ()
| otherwise             = traceError "wrong redeemer!"
```

We will also need to modify the off-chain code. The grab function will take now an input parameter that will be of type Integer.

```
1  type GiftSchema =
2      Endpoint "give" Integer
3      .\/ Endpoint "grab" Integer
4
5  grab :: forall w s e. AsContractError e => Integer -> Contract w s e ()
6  grab n = do
7      utxos <- utxosAt scrAddress
8      let orefs   = fst <$> Map.toList utxos
9      lookups = Constraints.unspentOutputs utxos      <>
10         Constraints.otherScript validator
11      tx :: TxConstraints Void Void
12      tx      = mconcat [mustSpendScriptOutput oref $ Redeemer $
13                      Builtins.mkI n | oref <- orefs]
14      ledgerTx <- submitTxConstraintsWith @Void lookups tx
15      void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
16      logInfo @String $ "collected gifts"
17
18 endpoints :: Contract () GiftSchema Text ()
19 endpoints = awaitPromise (give` `select` grab') >> endpoints
20 where
21     give' = endpoint @"give" give
22     grab' = endpoint @"grab" grab
```

From the original code in *Gift.hs* what changes are lines 3, 5, 6, 12-13 and 22. These examples were for the low-level data types in the validator function.

2.2 High Level typed validation scripts

Now let's look at an example where we use the high-level data types. We call it a typed validation script. The code can be found in *Typed.hs*. Here we provide only the significant parts.

```
1  import qualified Ledger.Typed.Scripts as Scripts
2
```

```

3  mkValidator :: () -> Integer -> ScriptContext -> Bool
4  mkValidator _ r _ = traceIfFalse "wrong redeemer" $ r == 42
5
6  data Typed
7  instance Scripts.ValidatorTypes Typed where
8      type instance DatumType Typed = ()
9      type instance RedeemerType Typed = Integer
10
11 typedValidator :: Scripts.TypedValidator Typed
12 typedValidator = Scripts.mkTypedValidator @Typed
13     $$($PlutusTx.compile [|| mkValidator ||])
14     $$($PlutusTx.compile [|| wrap ||])
15     where
16         wrap = Scripts.wrapValidator @() @Integer
17
18 validator :: Validator
19 validator = Scripts.validatorScript typedValidator
20
21 give :: AsContractError e => Integer -> Contract w s e ()
22 give amount = do
23     let tx = mustPayToTheScript () $ Ada.lovelaceValueOf amount
24     ledgerTx <- submitTxConstraints typedValidator tx
25     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
26     logInfo @String $ printf "made a gift of %d lovelace" amount

```

First we need to import *Ledger.Typed.Scripts* instead of *Ledger.Scripts*. We mentioned we can use arbitrary data types for the datum and redeemer. Since we don't care for the datum we just use unit. But for the script we need to use the *ScriptContext* data type (3). And the result will now be of type *Bool*. To compile the validator function to Plutus core we need to introduce a new type that encodes the information of the datum and the redeemer (6-9). We can pick an arbitrary name for this type. It doesn't need any constructors we just need to make it an instance of *Scripts.ValidatorTypes*. Next we compile our validator function but because we need a translation from our custom types to the low-level types we need to add the wrap function (14-16). This gives us a typed validator and we turn it into an un-typed validator (18-19). In the off-chain code we redefine our give function. When constructing the transaction we use now the function *mustPayToTheScript* that is a typed version for the case that the transaction we are constructing only involves one script. We now only provide as input the datum and the amount of ADA in lovelace. Then we also need to use the function *submitTxConstraints* to submit the transaction where we additionally provide as input our typed validator function. We also need the code for grab function introduced in the previous example and some code from the initial example, for the playground to work.

Now let's look at the functions that allow us to convert between the low-level and high-level data types. For this we need to look at the *PlutusTx.IsData.Class* module which contains the functions *toData* and *fromData*.

```
ghci> toData ()
Constr 0 []
ghci> fromData (Constr 0 []) :: Maybe ()
Just ()
ghci> fromData (Constr 1 []) :: Maybe ()
Nothing
ghci> toData (42 :: Integer)
I 42
ghci> fromData (I 42) :: Maybe Integer
Just 42
ghci> fromData (List []) :: Maybe Integer
Nothing
```

This works only for predefined instances of the *ToData* class, which you can get with the command “*:i ToData*”. If you want another data type you need to make it an instance of this class. But Plutus provides a mechanism that automatically does that for you. We look at this in our next example (*/IsData.hs*) where we use custom defined data types for our redeemer.

```
1 newtype MySillyRedeemer = MySillyRedeemer Integer
2
3 PlutusTx.unstableMakeIsData ''MySillyRedeemer
4
5 {-# INLINABLE mkValidator #-}
6 mkValidator :: () -> MySillyRedeemer -> ScriptContext -> Bool
7 mkValidator _ (MySillyRedeemer r) _ = traceIfFalse "wrong redeemer" $ r == 42
8
9 data Typed
10 instance Scripts.ValidatorTypes Typed where
11     type instance DatumType Typed = ()
12     type instance RedeemerType Typed = MySillyRedeemer
13
14 typedValidator :: Scripts.TypedValidator Typed
15 typedValidator = Scripts.mkTypedValidator @Typed
16     $$ (PlutusTx.compile [|| mkValidator ||])
17     $$ (PlutusTx.compile [|| wrap ||])
18     where
19         wrap = Scripts.wrapValidator @() @MySillyRedeemer
```

We first define our custom data type *MySillyRedeemer*. To make our data an instance of the *ToData* type class we can use a template Haskell function called *unstableMakeIsData* which does that for us. The syntax to provide a type is to use 2 single quotes in front of the type. If we manually try to convert it in the Repl we get the following result:

```
ghci> toData (MySillyRedeemer 42)
Constr 0 [I 42]
```

There is also a stable version of the template function which is more commonly used in production code. In our case we had only one data constructor but if there are many it's not clear how they will be ordered. The unstable version does not make any guarantees that between different Plutus version the constructor number corresponding to a given constructor will be preserved. Next the validator function now changes, where we pattern match the redeemer data type. Also the type instance and the wrapper function get updated. For the off-chain code only the transaction in the grab function has to be updated to:

```
tx = mconcat [mustSpendScriptOutput oref $ Redeemer $ PlutusTx.toBuiltinData
              (MySillyRedeemer r) | oref <- orefs]
```

2.3 Homework

Let's look now at an example where your custom data type is defined in record syntax. We will have two Booleans as input and the validator function should return True if both parameters are equal. You can find such an example in the *Solution2.hs* file from week02 examples.

```
1  {-# LANGUAGE DeriveAnyClass      #-}
2  {-# LANGUAGE DeriveGeneric       #-}
3
4  import           Data.Aeson        (FromJSON, ToJSON)
5  import           GHC.Generics     (Generic)
6  import           Playground.Contract (printJson, printSchemas,
                                         ensureKnownCurrencies, stage, ToSchema)
7
8  data MyRedeemer = MyRedeemer
9    { flag1 :: Bool
10   , flag2 :: Bool
11   } deriving (Generic, FromJSON, ToJSON, ToSchema)
12
13 -- This should validate if the two Booleans in the redeemer are equal!
14 mkValidator :: () -> MyRedeemer -> ScriptContext -> Bool
15 mkValidator () (MyRedeemer b c) _ = traceIfFalse "wrong redeemer" $ b == c
16
17 grab :: forall w s e. AsContractError e => MyRedeemer -> Contract w s e ()
18 grab r = do
19   utxos <- utxosAt scrAddress
20   let orefs   = fst <$> Map.toList utxos
21       lookups = Constraints.unspentOutputs utxos          <>
                  Constraints.otherScript validator
```

```

23      tx :: TxConstraints Void Void
24      tx      = mconcat [mustSpendScriptOutput oref $ Redeemer $
25                                PlutusTx.toBuiltinData r | oref <- orefs]
26      ledgerTx <- submitTxConstraintsWith @Void lookups tx
27      void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
28      logInfo @String $ "collected gifts"

```

Compared to our *Typed.hs* examples we need to add two new language extensions. After that we import 2 new libraries and from the *Playground.Contract* library we additionally import the *ToSchema* type class. Then we define our custom data type using record syntax where we derive the generic, JSON and schema type classes. Next we write our validator function where we can use pattern matching. And for the rest of the code the only thing that changes is the grab function. There we use now the *MyRedeemer* data type in the type signature and in the transaction definition (24) we provide the input parameter *r* that represents the data type *MyRedeemer*. In the Plutus playground if we define a grab action then we will see we have now two Boolean checkboxes with the above description of the record syntax functions (Figure 5). If both fields are checked or unchecked the transaction should be valid. Otherwise it should fail.

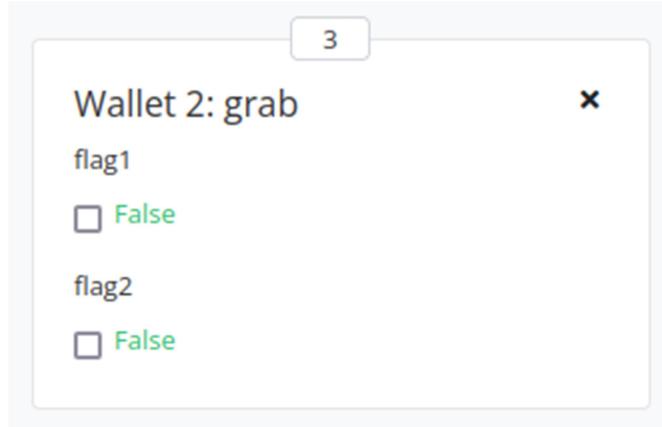


Figure 5 - Grab action

3 Time, parameterized contracts and the cardano testnet

In this chapter we will first time look at the script context. If we want to work with it we need to import the module *Ledger.Contexts* or just *Ledger* module which contains the data type *ScriptContext*. Script context defines two constructors that represent the transaction information and the script purpose (Figure 6). The script purpose constructor can be defined with four different parameters depending for what we use a script (Figure 8).

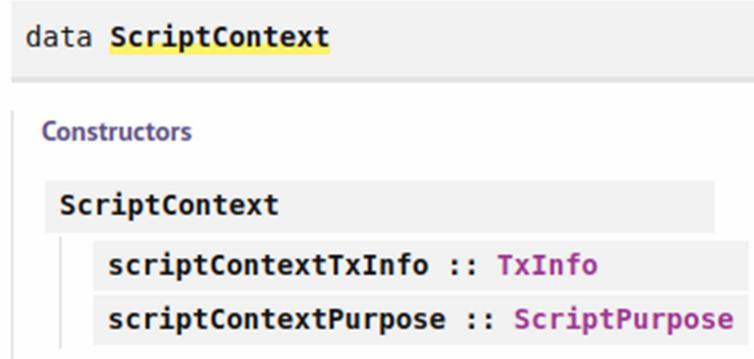


Figure 6 - Script context type

The transaction info data type is defined with various parameters that define properties of the transaction which are described in Figure 7. The *txInfoSignatories* parameter contains a list of public addresses which have signed this transaction. For producing transactions the *txInfoData* parameter is optional, whereas for spending transactions it is mandatory.

data TxInfo	
A pending transaction. This is the view as seen by validator scripts, so some details are stripped out.	
Constructors	
TxInfo	
<code>txInfoInputs :: [TxInInfo]</code>	Transaction inputs
<code>txInfoOutputs :: [TxOut]</code>	Transaction outputs
<code>txInfoFee :: Value</code>	The fee paid by this transaction.
<code>txInfoMint :: Value</code>	The <code>Value</code> minted by this transaction.
<code>txInfoDCert :: [DCert]</code>	Digests of certificates included in this transaction
<code>txInfoWdrl :: [(StakingCredential, Integer)]</code>	Withdrawals
<code>txInfoValidRange :: POSIXTimeRange</code>	The valid range for the transaction.
<code>txInfoSignatories :: [PubKeyHash]</code>	Signatures provided with the transaction, attested that they all signed the tx
<code>txInfoData :: [(DatumHash, Datum)]</code>	
<code>txInfoId :: TxId</code>	Hash of the pending transaction (excluding witnesses)

Figure 7 - Transaction info

data ScriptPurpose

Purpose of the script that is currently running

Constructors

Minting CurrencySymbol

Spending TxOutRef

Rewarding StakingCredential

Certifying DCert

Figure 8 - Script purpose

3.1 Time

If we can validate transactions in the wallet we should have a mechanism that prevents a transaction being executed on a node if it does not fall in a certain time range. And for this we have the *txInfoValidRange* field that is part of the transaction info and specifies the time range in which a transaction is valid. This is part of the general check procedure that happens before a transaction is executed. Part of this procedure is also checking that all the inputs are present, that the balances add up and that the fees are included. If these pre-check succeeds we can be sure that the validation will succeed if it was also successfully validated in the wallet. By default all transactions use an infinite time range.

The consensus protocol of Cardano called Ouroboros uses slots instead of using POSIX time. Currently a slot equals to 1 second but this can change in the future. A hard fork which would change the slot interval is known around 36 hours in advance. For this reason slot intervals should not have an upper bound that is too further in the future as 36 hours. If we look at the *POSIXTimeRange* data type in depth we get the following information:

```
type POSIXTimeRange = Interval POSIXTime
-----
data Interval a
Constructors:
  ivFrom :: LowerBound a
  ivTo :: UpperBound a
-----
data LowerBound a
Constructors:
  (Extended a) Closure
-----
data UpperBound a
```

```

Constructors:
  (Extended a) Closure
-----
type Closure = Bool
-----
data Extended a
Constructors:
  NegInf
  Finite a
  PosInf
-----
newtype POSIXTime
Constructors:
  getPOSIXTime :: Integer

```

Closure specifies whether the boundary is included or not. POSIX time is measured as the number of milliseconds since 1970-01-01 00:00:00. There are also some functions available associated with the *Interval* data type:

- *member*: Checks whether a value is in an interval.
- *interval*: takes in two parameters and constructs an *Interval a* with boundaries included.
- *from*: gives an interval *a* that includes all values that are greater than or equal to a.
- *to*: gives an interval *a* that includes all values that are smaller than or equal to a.
- *always*: An interval *a* that covers every slot.
- *never*: An interval *a* that is empty.
- *singleton*: An interval *a* that only contains the one a.
- *hull*: 'hull a b' is the smallest interval containing a and b intervals.
- *intersection*: 'intersection a b' is the largest interval that is contained in a and in b intervals, if it exists.
- *overlap*: checks weather two intervals have a value in common and returns a Boolean
- *contains*: checks weather the second interval is contained in the first one
- *isEmpty*: checks weather an interval is empty
- *before*: checks weather a given time is before the given interval
- *after*: checks weather a given time is after the given interval

All of these functions and data types are included in the *Ledger.Interval* module. Let's look at some code examples that use these functions:

```

ghci> interval (10 :: Integer) 20
Interval {ivFrom = LowerBound (Finite 10) True, ivTo = UpperBound (Finite 20) True}
ghci> member 9 $ interval (10 :: Integer) 20
False
ghci> member 10 $ interval (10 :: Integer) 20
True
ghci> member 29 $ from (30 :: Integer)

```

```

False
ghci> member 30 $ from (30 :: Integer)
True
ghci> member 31 $ to (30 :: Integer)
False
ghci> member 30 $ to (30 :: Integer)
True
ghci> intersection (interval (10 :: Integer) 20) $ interval 18 30
Interval {ivFrom = LowerBound (Finite 18) True, ivTo = UpperBound (Finite 20) True}
ghci> contains (to (100 :: Integer)) $ interval 30 80
True
ghci> contains (to (100 :: Integer)) $ interval 30 101
False
ghci> overlaps (to (100 :: Integer)) $ interval 30 101
True

```

Next let's look at the *Vesting.hs* example found in week03 folder where we create a script address with some ADA that can be redeemed after a certain date has passed.

```

1  import           Ledger.Constraints      (TxConstraints)
2  import qualified Ledger.Constraints    as Constraints
3  import           Prelude                  (IO, Semigroup (..), Show(..), String)
4
5  data VestingDatum = VestingDatum
6    { beneficiary :: PaymentPubKeyHash
7     , deadline   :: POSIXTime
8    } deriving Show
9
10 PlutusTx.unstableMakeIsData ''VestingDatum
11
12 {-# INLINABLE mkValidator #-}
13 mkValidator :: VestingDatum -> () -> ScriptContext -> Bool
14 mkValidator dat () ctx = traceIfFalse "beneficiary's signature missing"
                           signedByBeneficiary &&
                           traceIfFalse "deadline not reached" deadlineReached
15   where
16     info :: TxInfo
17     info = scriptContextTxInfo ctx
18
19     signedByBeneficiary :: Bool
20     signedByBeneficiary = txSignedBy info $ unPaymentPubKeyHash $
                           beneficiary dat
21
22     deadlineReached :: Bool
23     deadlineReached = contains (from $ deadline dat) $ txInfoValidRange info
24
25
26 data Vesting

```

```

27 instance Scripts.ValidatorTypes Vesting where
28     type instance DatumType Vesting = VestingDatum
29     type instance RedeemerType Vesting = ()
30
31 typedValidator :: Scripts.TypedValidator Vesting
32 typedValidator = Scripts.mkTypedValidator @Vesting
33     $$ (PlutusTx.compile [|| mkValidator ||])
34     $$ (PlutusTx.compile [|| wrap ||])
35     where
36         wrap = Scripts.wrapValidator @VestingDatum @()
37
38 validator :: Validator
39 validator = Scripts.validatorScript typedValidator
40
41 valHash :: Ledger.ValidatorHash
42 valHash = Scripts.validatorHash typedValidator
43
44 scrAddress :: Ledger.Address
45 scrAddress = scriptAddress validator
46
47 data GiveParams = GiveParams
48     { gpBeneficiary :: !PaymentPubKeyHash
49     , gpDeadline    :: !POSIXTime
50     , gpAmount      :: !Integer
51     } deriving (Generic, ToJSON, FromJSON, ToSchema)
52
53 type VestingSchema =
54     Endpoint "give" GiveParams
55     .\|/ Endpoint "grab" ()
56
57 give :: AsContractError e => GiveParams -> Contract w s e ()
58 give gp = do
59     let dat = VestingDatum
60             { beneficiary = gpBeneficiary gp
61             , deadline   = gpDeadline gp
62             }
63     tx  = Constraints.mustPayToTheScript dat $ Ada.lovelaceValueOf $
64             gpAmount gp
65     ledgerTx <- submitTxConstraints typedValidator tx
66     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
67     logInfo @String $ printf "made a gift of %d lovelace to %s
68                                     with deadline %s"
69             (gpAmount gp)
70             (show $ gpBeneficiary gp)
71             (show $ gpDeadline gp)

```

```

70
71 grab :: forall w s e. AsContractError e => Contract w s e ()
72 grab = do
73     now    <- currentTime
74     pkh    <- ownPaymentPubKeyHash
75     utxos <- Map.filter (isSuitable pkh now) <$> utxosAt scrAddress
76     if Map.null utxos
77         then logInfo @String $ "no gifts available"
78     else do
79         let orefs   = fst <$> Map.toList utxos
80         lookups = Constraints.unspentOutputs utxos <>
81                         Constraints.otherScript validator
82         tx :: TxConstraints Void Void
83         tx      = mconcat [Constraints.mustSpendScriptOutput oref
84                             unitRedeemer | oref <- orefs] <>
85                         Constraints.mustValidateIn (from now)
86         ledgerTx <- submitTxConstraintsWith @Void lookups tx
87         void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
88         logInfo @String $ "collected gifts"
89     where
90         isSuitable :: PaymentPubKeyHash -> POSIXTime -> ChainIndexTxOut -> Bool
91         isSuitable pkh now o = case _ciTxOutDatum o of
92             Left _           -> False
93             Right (Datum e) -> case PlutusTx.fromBuiltinData e of
94                 Nothing -> False
95                 Just d  -> beneficiary d == pkh && deadline d <= now
96
97 endpoints :: Contract () VestingSchema Text ()
98 endpoints = awaitPromise (give' `select` grab') >> endpoints
99 where
100    give' = endpoint @"give" give
101    grab' = endpoint @"grab" $ const grab
102
103 mkSchemaDefinitions ''VestingSchema
104 mkKnownCurrencies []

```

Compared to the last code example *Solution2.hs* from the week02 folder, we make a different import of *Ledger.Constraints* one qualified and one normal. And from the standard Prelude we also import the *Show* type class. Then we define the vesting datum, which contains the hash of the payment public key from the user that will retrieve the funds and the deadline after which the funds can be retrieved. We derive the *Show* type class so we can display our information in the console. Next we write our validator function, where we look at the datum and the context.

In the *where* clause we first define the transaction info. After that we implement our two conditions. For the first condition *signedByBeneficiary* we use the *txSignedBy* function that takes a transaction info and a public key hash and returns True if the transaction was signed with this key. The function expects a *PubKeyHash* variable which we get with *unPaymentPubKeyHash*.

```
Prelude Ledger> :i PaymentPubKeyHash
type PaymentPubKeyHash :: *
newtype PaymentPubKeyHash
= PaymentPubKeyHash {unPaymentPubKeyHash :: PubKeyHash}
-- Defined in 'Ledger.Address'
```

For the second condition *deadlineReached* we construct an interval that stretches from the deadline to infinity and if the validity interval *txInfoValidRange* from the transaction info object is contained in this interval the deadline is definitely reached. Next follows the code where we compile our validator and create the script address.

After that follows the off-chain code. First we define a data type called *GiveParams* that includes information necessary to construct the script address. We add an exclamation mark to the parameters so they get strictly evaluated. Then we write our give function where we first define the datum which we use in the next step when constructing the transaction.

For the grab function we can have more UTXOs sitting at the vesting address. First we get the current time and lookup our own payment public key hash. Then we get all suitable UTXOs, where we use the helper function *isSuitable*. It takes in our payment key, the current time and a UTXO and returns a Bool. Inside the body of the helper function we check the datum of the UTXO with the function *_ciTxOutDatum*. If it does not exist and only the hash exist we will get a Left type and can return false. If it exists we check the content of the datum. If there is a content present we can validate our two conditions. Else we also just return False.

After we filtered out the valid UTXOs we check if there are any. If there aren't we log a message else we construct a transaction that collects all of them in one transaction. In the transaction we also use the function *mustValidateIn* that creates a validity interval for the transaction (84). In the real world there could be too many UTXOs to collect them in one transaction which can be only of a limited size. But in this example we forget about this case.

If you try this code out in the playground and have a scenario where wallet one makes a gift to wallet two and wallet three, you will need a slot in between the two gift actions because we defined the code in such a way that we wait for confirmation of the transaction before we finish with the give action. So it will be again available after the transaction is confirmed. You will also need to provide the payment public key hash in the give action of the wallet you want to give the funds. You can get this with the following commands inside the Repl:

```

Ghci> import Wallet.Emulator
Ghci> knownWallet 2
Wallet 7ce812d7a4770bbf58004067665c3a48f28ddd58
Ghci> mockWalletPaymentPubKey $ knownWallet 2
80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7
Ghci> mockWalletPaymentPubKey $ knownWallet 3
2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c

```

The *Wallet* type represents real wallets like Daedalus and also mock wallets that are used in the playground. With the *mockWalletPaymentPubKey* function we get the public key hash of a specific wallet from the playground. Next you need to provide the deadline which needs to be in POSIX time. We can get this time with the following commands:

```

Ghci> import Ledger.Time
Ghci> import Ledger.TimeSlot
Ghci> import Data.Default
Ghci> slotToBeginPOSIXTime def 10
POSIXTime {getPOSIXTime = 1596059101000}

```

3.2 Parameterized Contracts

Let's look now at an example where our validation script takes an input parameter. The code for this can be found in the *Parameterized.hs* file in the week03 folder.

```

1  {-# LANGUAGE MultiParamTypeClasses #-}
2
3  data VestingParam = VestingParam
4    { beneficiary :: PaymentPubKeyHash
5    , deadline    :: POSIXTime
6    } deriving Show
7
8  PlutusTx.makeLift ''VestingParam
9
10 {-# INLINABLE mkValidator #-}
11 mkValidator :: VestingParam -> () -> () -> ScriptContext -> Bool
12 mkValidator p () () ctx = traceIfFalse "beneficiary's signature missing"
                           signedByBeneficiary &&
                           traceIfFalse "deadline not reached" deadlineReached
13   where
14     info :: TxInfo
15     info = scriptContextTxInfo ctx
16
17
18     signedByBeneficiary :: Bool
19     signedByBeneficiary = txSignedBy info $ unPaymentPubKeyHash $
                           beneficiary p

```

```

20
21     deadlineReached :: Bool
22     deadlineReached = contains (from $ deadline p) $ txInfoValidRange info
23
24 data Vesting
25 instance Scripts.ValidatorTypes Vesting where
26     type instance DatumType Vesting = ()
27     type instance RedeemerType Vesting = ()
28
29 typedValidator :: VestingParam -> Scripts.TypedValidator Vesting
30 typedValidator p = Scripts.mkTypedValidator @Vesting
31     ($$(PlutusTx.compile [|| mkValidator ||])
32       `PlutusTx.applyCode` PlutusTx.liftCode p)
33     $$$(PlutusTx.compile [|| wrap ||])
34     where
35         wrap = Scripts.wrapValidator @() @()
36
37 validator :: VestingParam -> Validator
38 validator = Scripts.validatorScript . typedValidator
39
40 valHash :: VestingParam -> Ledger.ValidatorHash
41 valHash = Scripts.validatorHash . typedValidator
42
43 scrAddress :: VestingParam -> Ledger.Address
44 scrAddress = scriptAddress . validator
45
46 data GiveParams = GiveParams
47     { gpBeneficiary :: !PaymentPubKeyHash
48     , gpDeadline    :: !POSIXTime
49     , gpAmount      :: !Integer
50     } deriving (Generic, ToJSON, FromJSON, ToSchema)
51
52 type VestingSchema =
53     Endpoint "give" GiveParams
54     .\|/ Endpoint "grab" POSIXTime
55
56 give :: AsContractError e => GiveParams -> Contract w s e ()
57 give gp = do
58     let p = VestingParam
59         { beneficiary = gpBeneficiary gp
60         , deadline   = gpDeadline gp
61         }
62     tx = Constraints.mustPayToTheScript () $ Ada.lovelaceValueOf $
63           gpAmount gp
64     ledgerTx <- submitTxConstraints (typedValidator p) tx

```

```

63     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
64
65     logInfo @String $ printf "made a gift of %d lovelace to %s
66             with deadline %s"
67             (gpAmount gp)
68             (show $ gpBeneficiary gp)
69             (show $ gpDeadline gp)
70
71     grab :: forall w s e. AsContractError e => POSIXTime -> Contract w s e ()
72     grab d = do
73         now    <- currentTime
74         pkh    <- ownPaymentPubKeyHash
75         if now < d
76             then logInfo @String $ "too early"
77         else do
78             let p = VestingParam
79                 { beneficiary = pkh
80                 , deadline    = d
81                 }
82             utxos <- utxosAt $ scrAddress p
83             if Map.null utxos
84                 then logInfo @String $ "no gifts available"
85             else do
86                 let orefs   = fst <$> Map.toList utxos
87                 lookups = Constraints.unspentOutputs utxos      <>
88                           Constraints.otherScript (validator p)
89                 tx :: TxConstraints Void Void
90                 tx      = mconcat [Constraints.mustSpendScriptOutput
91                                     oref unitRedeemer | oref <- orefs] <>
92                           Constraints.mustValidateIn (from now)
93                 ledgerTx <- submitTxConstraintsWith @Void lookups tx
94                 void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
95                 logInfo @String $ "collected gifts"
96
97     endpoints :: Contract () VestingSchema Text ()
98     endpoints = awaitPromise (give` `select` grab') >> endpoints
99     where
100        give' = endpoint @"give" give
101        grab' = endpoint @"grab" grab
102
103 mkSchemaDefinitions ''VestingSchema
104
105 mkKnownCurrencies []

```

First we add an language extension called *MultiParamTypeClasses*. Other extensions and imports are the same as in the previous *Vesting.hs* example. Then we define the vesting parameter which has basically the same structure as in our vesting example.

Next we add our vesting parameter as an additional parameter to our validator function. The datum and the redeemer are set to unit (). Now during compilation all data should be known in advance but we can make an exception for the parameter if it is a static piece of data and not a function. We do this by calling the function *PlutusTx.liftCode* and then we combine it with the validator function that is now written in Plutus core with help of *PlutusTx.applyCode* function (31). Our type validator takes now as input the vesting parameter which can be seen in the type signature (29) and function definition (30). And additionally to that in the wrap function the datum changes now to unit (34).

To be able for this code to work we need a lift instance for our *VestingParam* parameter (8). And if there are multiple arguments in our parameter we need to add a the language pragma from line 1. Of course now in our validator function when we write our two conditions we read the data from the additional parameter instead of the datum. In line 26 the type of the datum changes to unit. When we define our validator, its hash and the script address instead of calling functions upon static piece of data we have now two functions that we need to combine with the dot notation (36 to 43).

When we define our endpoints we need an input parameter for grab which will be the POSIX time of the deadline. The payment public key hash we can get with the appropriate function. For the give parameter what changes compared to the previous code example is that we now input the vesting parameter variable together with the type validator when we submit the transaction and not as the datum contained in the transaction (62).

In the grab function we now add the POSIX time as an input parameter but the actual deadline is already set by the person who performs the give action. First we check if this parameter is larger than the current time (73) and if yes we log a »too early« message. If not we define our vesting parameter (76 to 78), then we get a list of all the UTXOs where we have to provide the vesting parameter as input to the script address (80). If there are any UTXOs then we construct the transaction and submit it same as in the vesting example, with the only difference that now in the lookups parameter we add the vesting variable to the validator (86). Compared to our last example where we filtered out the suitable UTXOs with the *isSuitable* function we now provide an input parameter to the script that automatically returns the UTXOs that match the public payment key hash and the exact deadline.

In the playground we have to provide the public key hash and the deadline for the give action. We also now specify the deadline we are looking for as an argument to grab. For this reason we

cannot grab two gifts at once if they exist. We have to perform two grab actions and the posix time in the grab action has to exactly match the posix time in the give action, to be able to retrieve the funds. One thing to mention is an obvious failure of the playground.

If we try to make a gift wait until slot 10, then make a grab and wait for 2 more slots, the grab will succeed. But if the waiting time after the grab is 1 slot then the grab will fail no matter what the waiting time after the give action is. But since we used a posix time of 10 slots as input for the give and grab action this is in contradiction with counting the time.

3.3 The Cardano testnet

Here we will show how to use the Cardano command line interface to deploy some code to the cardano test net or main net. To do this you must run a cardano node which you can find here:

<https://github.com/input-output-hk/cardano-node>

On the right side under Releases section click on the latest cardano node (Figure 9).

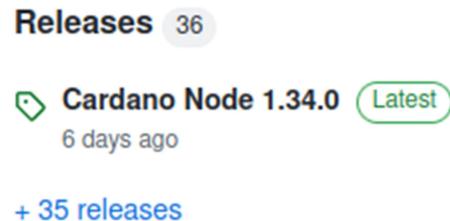


Figure 9 - Cardano node

Under the technical specifications chapter you have a downloads section where you can click on Hydra Binaries. A web-page opens where you can download your installer for a given OS. Once the cardano node is installed you can check your version with the command:

```
$ cardano-node --version
```

You will also need some configuration files which you can find again in the technical specifications chapter in the downloads section. Download the following test-net files:

- config
- byronGenesis
- shellyGenesis
- alonzoGenesis
- topology

Create a *testnet*/ folder on your PC and put the files into that folder. Then put the file *start-node-testnet.sh* into that folder, which can be found in the week03 folder of the plutos-pioneer-program git repository. This bash script starts the cardano done.

When you start the node for the first time you will need to wait for a couple of hours for the test net to synchronize. Beside the cardano node also the Cardano command line interface (CLI) gets installed. You can look at the help page of the client with the command:

```
$ cardano-cli --help
```

You will get a list of top-level commands. If you want to search through a command use the same pattern and call the help page on this command to get available sub-commands:

```
$ cardano-cli address --help
```

You can repeat this procedure on a sub-command of the address command. To try out a Plutus contract we will need two keys. Use the following commands from within the *testnet* folder to create your two keys:

```
$ cardano-cli address key-gen --verification-key-file 01.vkey --signing-key-file 01.skey  
$ cardano-cli address key-gen --verification-key-file 02.vkey --signing-key-file 02.skey
```

Next you will need to generate two payment addresses which will represent two wallets.

```
$ cardano-cli address build --payment-verification-key-file 01.vkey --testnet-magic 1097911063 --out-file 01.addr  
$ cardano-cli address build --payment-verification-key-file 02.vkey --testnet-magic 1097911063 --out-file 02.addr
```

You can find the test-net magic number in the *testnet-shelley-genesis.json* configuration file. Now we need some test ADA to be able to make transactions on the test-net. For this we use something called the Faucet: <https://testnets.cardano.org/en/testnets/cardano/tools/faucet/>.

The faucet is a web-based service that provides test ADA to users of the test-net. On the above web-page you past in your address that you can find by viewing your 01.addr and 02.addr files and then complete the test that checks you not a robot. After that click on the Request funds button. If you want to see if the test ADA arrived you can query the blockchain:

```
$ export CARDANO_NODE_SOCKET_PATH=node.socket  
$ cardano-cli query utxo --address $(cat 01.addr) --testnet-magic 1097911063
```

You should see on output of the transaction hash and transaction index which together define an UTXO and information about how much ADA was sent. If you want to add funds to your second address and don't have an API key you will have to wait 24 hours to be able to send

some test ADA again from the Faucet. An easier way is to send some ADA from your first address to your second address. To send the ADA you can use the *send.sh* bash script which can be found in week03/testnet folder of the pioneer git repository. You will need to change the tx-in field with your transaction hash and the index specified after the # symbol that you got from the previous query command. The transaction build command automatically calculates fees and creates change outputs in case your inputs are higher than the outputs.

There is a block on the Cardano test-net and main-net on average every twenty seconds so you will have to wait a bit before you check whether your transaction was processed. When it is you will have a new transaction has an UTXO that will contain the change ADA.

If we want now to use Plutus code in the cardano CLI we need to serialize and write to disk various Plutus types. We do this with the *Deploy.hs* code.

```
1 {-# LANGUAGE OverloadedStrings #-}
2 {-# LANGUAGE TypeApplications #-}
3
4 module Week03.Deploy
5   ( writeJSON
6   , writeValidator
7   , writeUnit
8   , writeVestingValidator
9   ) where
10
11 import           Cardano.Api
12 import           Cardano.Api.Shelley    (PlutusScript(..))
13 import           Codec.Serialise      (serialise)
14 import           Data.Aeson          (encode)
15 import qualified Data.ByteString.Lazy as LBS
16 import qualified Data.ByteString.Short as SBS
17 import           PlutusTx            (Data(..))
18 import qualified PlutusTx
19 import qualified Ledger
20
21 import           Week03.Parameterized
22
23 dataToScriptData :: Data -> ScriptData
24 dataToScriptData (Constr n xs) = ScriptDataConstructor n $ dataToScriptData <$> xs
25 dataToScriptData (Map xs)       = ScriptDataMap [(dataToScriptData x, dataToScriptData y) | (x, y) <- xs]
26 dataToScriptData (List xs)     = ScriptDataList $ dataToScriptData <$> xs
27 dataToScriptData (I n)         = ScriptDataNumber n
```

```

28 dataToScriptData (B bs)      = ScriptDataBytes bs
29
30 writeJSON :: PlutusTx.ToData a => FilePath -> a -> IO ()
31 writeJSON file = LBS.writeFile file . encode . scriptDataToJson
                  ScriptDataJsonDetailedSchema . dataToScriptData .
                  PlutusTx.toData
32
33 writeValidator :: FilePath -> Ledger.Validator ->
                  IO (Either (FileError ()) ())
34 writeValidator file = writeFileTextEnvelope @PlutusScript PlutusScriptV1
                  file Nothing . PlutusScriptSerialised . SBS.toShort .
                  LBS.toStrict . serialise . Ledger.unValidatorScript
35
36 writeUnit :: IO ()
37 writeUnit = writeJSON "testnet/unit.json" ()
38
39 writeVestingValidator :: IO (Either (FileError ()) ())
40 writeVestingValidator = writeValidator "testnet/vesting.plutus" $
                  validator $ VestingParam
41                  { beneficiary = Ledger.PaymentPubKeyHash
                  "c2ff616e11299d9094ce0a7eb5b7284b705147a822f4ffbd471f971a"
42                  , deadline    = 1643235300000
43                  }

```

The *Cardano.Api* is the Haskell library which the cardano CLI uses. It has its own data type called *ScriptData*. With the *writeJSON* function we convert some Plutus data to script data and write it in a JSON format to a file (30-31). On lines 23 to 28 we define the conversion function *dataToScriptData*. Then we define the *writeUnit* function which basically writes a unit object to the specified file (36-37). We also need our Plutus validator script that we have to convert to a script and write it to a file. We do this with the *writeValidator* function (33-34). We want to apply this function to our parameterized contract. We do this with the *writeVestingValidator* IO action where we need to specify the beneficiary and the deadline. Note that we also use here our *validator* object which is available because we imported the *Week03.Parameterized* module, which is defined in the *Parameterized.hs* file.

We get the payment public key has of our second address with this command:

```
$ cardano-cli address key-hash --payment-verification-key-file 02.vkey --out-file 02.pkh
```

We copy the hash from the file 02.pkh and past it into the code for the IO action. We can get our deadline from epochconverter.com where we can specify the date and time and get back

the POSIX time in milliseconds. We also need the actual address corresponding to the script. First we execute the command `writeVestingValidator` in the Repl. Then we execute:

```
$ cardano-cli address build-script --script-file vesting.plutus --testnet-magic 1097911063 --out-file vesting.addr
```

We can give now some ADA to the vesting address we just created. You can do this with the `give.sh` bash script also found in the testnet folder. In the `--tx-in` field you have to specify the new transaction hash and index of the UTXO sitting at address 1. This changes after you send some ADA to address 2, so you will need to do a query again. In the `transaction build` command we use the `--tx-out-datum-hash-file` option that computes the hash of a datum written to a JSON file. After we run this script we can again check if the ADA arrived:

```
$ cardano-cli query utxo --address $(cat vesting.addr) --testnet-magic 1097911063
```

The second wallet wants now to grab that ADA and we can use the `grab.sh` bash script to do this. Since this is a spending transaction we need to provide the following things:

- The transaction hash and id which we provide in the `--tx-in` option.
- The actual script which we provide in the `--tx-in-script-file` option.
- The datum which we provide in the `--tx-in-datum-file` option. We get the unit datum if we load our `Deploy.hs` file in the Repl and execute the command `writeUnit`.
- The redeemer (same as the datum) which we provide in the `--tx-in-redeemer-file` option.
- In the `--tx-in-collateral` option you specify an UTXO that belongs to yourself and contains only ADA and no native tokens. It represents the collateral which must be large enough to cover the costs if validation would fail. This is a special case where you can circumvent the validation process and you would get a failed transaction. The nodes that process your transaction have to be reimbursed for that or the system would be open to denial of service attacks. We can use the `02.addr` for collateral.
- In the `--required-signer-hash` field we specify the public key hash of our second address.
- The validity interval which we provide in the `--invalid-before` option. It has to be provided as slots. We get the current slot of the test-net with the command:

```
$ cardano-cli query tip --testnet-magic 1097911063
```

The slot time we specify should be after our deadline.

- The protocol parameters which we provide in `--protocol-params-file` option. You get the content of the file with the command:

```
$ cardano-cli query protocol-parameters --testnet-magic 1097911063 --out-file protocol.json
```

Once we configured everything the grab should work and if we would query the second address we should see that there are two UTXOs sitting at this address. For the main-net the procedure would be the same only that we would specify `--mainnet` option instead of `--testnet-magic`.

3.4 Homework

You can do the `Homework1.hs` and `Homework2.hs` examples by yourself. We will present here the solution and comment on it. For homework 1 we want to create a script address from which a beneficiary can retrieve his funds until a certain deadline. After the deadline we can retrieve the funds back to our own address. Here is the `Solution1.hs` code example.

```
1  import qualified Prelude          as P
2
3  data VestingDatum = VestingDatum
4    { beneficiary1 :: PaymentPubKeyHash
5    , beneficiary2 :: PaymentPubKeyHash
6    , deadline     :: POSIXTime
7    } deriving P.Show
8
9  PlutusTx.unstableMakeIsData ''VestingDatum
10
11 {-# INLINABLE mkValidator #-}
12 -- This should validate if either beneficiary1 has signed the transaction
13 -- and the current slot is before or at the deadline or if beneficiary2
14 -- has signed the transaction and the deadline has passed.
15 mkValidator :: VestingDatum -> () -> ScriptContext -> Bool
16 mkValidator dat () ctx
17   | (unPaymentPubKeyHash (beneficiary1 dat) `elem` sigs) &&
18     (to (deadline dat) `contains` range) = True
19   | (unPaymentPubKeyHash (beneficiary2 dat) `elem` sigs) &&
20     (from (1 + deadline dat) `contains` range) = True
21   | otherwise = False
22
23 where
24   info :: TxInfo
25   info = scriptContextTxInfo ctx
26
27   sigs :: [PubKeyHash]
28   sigs = txInfoSignatories info
29
30   range :: POSIXTimeRange
```

```

27     range = txInfoValidRange info
28
29 data Vesting
30 instance Scripts.ValidatorTypes Vesting where
31     type instance DatumType Vesting = VestingDatum
32     type instance RedeemerType Vesting = ()
33
34 typedValidator :: Scripts.TypedValidator Vesting
35 typedValidator = Scripts.mkTypedValidator @Vesting
36     $$($PlutusTx.compile [|| mkValidator ||])
37     $$($PlutusTx.compile [|| wrap ||])
38     where
39         wrap = Scripts.wrapValidator @VestingDatum @()
40
41 validator :: Validator
42 validator = Scripts.validatorScript typedValidator
43
44 scrAddress :: Ledger.Address
45 scrAddress = scriptAddress validator
46
47 data GiveParams = GiveParams
48     { gpBeneficiary :: !PaymentPubKeyHash
49     , gpDeadline    :: !POSIXTime
50     , gpAmount      :: !Integer
51     } deriving (Generic, ToJSON, FromJSON, ToSchema)
52
53 type VestingSchema =
54     Endpoint "give" GiveParams
55     .\|/ Endpoint "grab" ()
56
57 give :: AsContractError e => GiveParams -> Contract w s e ()
58 give gp = do
59     pkh <- ownPaymentPubKeyHash
60     let dat = VestingDatum
61         { beneficiary1 = gpBeneficiary gp
62         , beneficiary2 = pkh
63         , deadline     = gpDeadline gp
64         }
65     tx  = Constraints.mustPayToTheScript dat $ Ada.lovelaceValueOf $
66             gpAmount gp
67     ledgerTx <- submitTxConstraints typedValidator tx
68     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
69     logInfo @P.String $
70     printf "made a gift of %d lovelace to %s with deadline %s"
71             (gpAmount gp)

```

```

70          (P.show $ gpBeneficiary gp)
71          (P.show $ gpDeadline gp)
72
73  grab :: forall w s e. AsContractError e => Contract w s e ()
74  grab = do
75      now      <- currentTime
76      pkh      <- ownPaymentPubKeyHash
77      utxos   <- utxosAt scrAddress
78      let utxos1 = Map.filter (isSuitable $ \dat -> beneficiary1 dat == pkh &&
79                                now <= deadline dat) utxos
79      utxos2 = Map.filter (isSuitable $ \dat -> beneficiary2 dat == pkh &&
80                                now >  deadline dat) utxos
80      logInfo @P.String $ printf "found %d gift(s) to grab"
81                      (Map.size utxos1 P.+ Map.size utxos2)
81  unless (Map.null utxos1) $ do
82      let orefs   = fst <$> Map.toList utxos1
83      lookups = Constraints.unspentOutputs utxos1 P.<>
84                      Constraints.otherScript validator
85      tx :: TxConstraints Void Void
86      tx      = mconcat [Constraints.mustSpendScriptOutput oref
87                          unitRedeemer | oref <- orefs] P.<>
88                      Constraints.mustValidateIn (to now)
88      void $ submitTxConstraintsWith @Void lookups tx
89  unless (Map.null utxos2) $ do
90      let orefs   = fst <$> Map.toList utxos2
91      lookups = Constraints.unspentOutputs utxos2 P.<>
92                      Constraints.otherScript validator
93      tx :: TxConstraints Void Void
94      tx      = mconcat [Constraints.mustSpendScriptOutput oref $
95                          unitRedeemer | oref <- orefs] P.<>
96                      Constraints.mustValidateIn (from now)
96      void $ submitTxConstraintsWith @Void lookups tx
97  where
98      isSuitable :: (VestingDatum -> Bool) -> ChainIndexTxOut -> Bool
99      isSuitable p o = case _ciTxOutDatum o of
100          Left _           -> False
101          Right (Datum d) -> maybe False p $ PlutusTx.fromBuiltinData d
102
103 endpoints :: Contract () VestingSchema Text ()
104 endpoints = awaitPromise (give' `select` grab') >> endpoints
105  where
106      give' = endpoint @"give" give
107      grab' = endpoint @"grab" $ const grab
108
109 mkSchemaDefinitions ''VestingSchema

```

```
110
111 mkKnownCurrencies []
```

Compared to the *Vesting.hs* example our vesting datum changes were we add another beneficiary (3-7). Then our validator function changes were we cover 2 cases for which the transaction is valid (11-27). First the case the beneficiary retrieves the funds before the deadline. Second the case that the giver retrieves the funds after the deadline. The give function is practically the same as in the vesting example with the exception that the datum contains now 2 beneficiaries. The grab function however changes now. We construct 2 UTXO lists where for the first the current time should be before the deadline and for the second after (78-79). Then we define 2 cases where the first is that the first UTXO list is not empty and the second case is when the second UTXOs list is not empty. For each case we construct the transaction and submit it. The difference in the transactions is the validity interval (87 and 96). Some of the code is then the same as in the vesting example. The *isSuitable* function also changes. Now we take in a function and a transaction output and return a bool. We provide the functions in lines 78 and 79. In the body of the *isSuitable* function we use the *maybe* function which has following type signature:

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

If the third parameter is a Nothing the first parameter is returned. Else we use the function a to b on the third parameter extracted from the Maybe to produce the result of type b. From the code in lines 98-101 and 78-79 we can see that the datum gets provided from an UTXO.

For homework 2 we want to modify the *Parameterized.hs* example in such a way that the input information is split between the additional parameter that will carry the payment public key hash and the datum that will carry the deadline. Here is the code.

```
1 import Prelude (IO, Semigroup(..), Show(..), String, undefined)
2
3 {-# INLINABLE mkValidator #-}
4 mkValidator :: PaymentPubKeyHash -> POSIXTime -> () -> ScriptContext -> Bool
5 mkValidator pkh s () ctx =
6     traceIfFalse "beneficiary's signature missing" checkSig      &&
7     traceIfFalse "deadline not reached"                  checkDeadline
8   where
9     info :: TxInfo
10    info = scriptContextTxInfo ctx
11
12    checkSig :: Bool
13    checkSig = unPaymentPubKeyHash pkh `elem` txInfoSignatories info
```

```

14
15     checkDeadline :: Bool
16     checkDeadline = from s `contains` txInfoValidRange info
17
18 data Vesting
19 instance Scripts.ValidatorTypes Vesting where
20     type instance DatumType Vesting = POSIXTime
21     type instance RedeemerType Vesting = ()
22
23 typedValidator :: PaymentPubKeyHash -> Scripts.TypedValidator Vesting
24 typedValidator p = Scripts.mkTypedValidator @Vesting
25     ($$($PlutusTx.compile [|| mkValidator ||]) `PlutusTx.applyCode` 
26      $PlutusTx.liftCode p)
27     $$($PlutusTx.compile [|| wrap ||])
28     where
29         wrap = Scripts.wrapValidator @POSIXTime @()
30
31 validator :: PaymentPubKeyHash -> Validator
32 validator = Scripts.validatorScript . typedValidator
33
34 scrAddress :: PaymentPubKeyHash -> Ledger.Address
35 scrAddress = scriptAddress . validator
36
37 data GiveParams = GiveParams
38     { gpBeneficiary :: !PaymentPubKeyHash
39     , gpDeadline    :: !POSIXTime
40     , gpAmount      :: !Integer
41     } deriving (Generic, ToJSON, FromJSON, ToSchema)
42
43 type VestingSchema =
44     Endpoint "give" GiveParams
45     .\|/ Endpoint "grab" ()
46
47 give :: AsContractError e => GiveParams -> Contract w s e ()
48 give gp = do
49     let p = gpBeneficiary gp
50     d = gpDeadline gp
51     tx = Constraints.mustPayToTheScript d $ Ada.lovelaceValueOf $
52         gpAmount gp
53     ledgerTx <- submitTxConstraints (typedValidator p) tx
54     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
55     logInfo @String $ printf "made a gift of %d lovelace
56                             to %s with deadline %s"
57     (gpAmount gp)
58     (show $ gpBeneficiary gp)

```

```

56           (show $ gpDeadline gp)
57
58 grab :: forall w s e. AsContractError e => Contract w s e ()
59 grab = do
60     now    <- currentTime
61     pkh    <- ownPaymentPubKeyHash
62     utxos <- Map.filter (isSuitable now) <$> utxosAt (scrAddress pkh)
63     if Map.null utxos
64         then logInfo @String $ "no gifts available"
65     else do
66         let orefs   = fst <$> Map.toList utxos
67         lookups = Constraints.unspentOutputs utxos      <>
68                   Constraints.otherScript (validator pkh)
69         tx :: TxConstraints Void Void
70         tx      = mconcat [Constraints.mustSpendScriptOutput oref
71                           unitRedeemer | oref <- orefs] <>
72                           Constraints.mustValidateIn (from now)
73         ledgerTx <- submitTxConstraintsWith @Void lookups tx
74         void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
75         logInfo @String $ "collected gifts"
76 where
77     isSuitable :: POSIXTime -> ChainIndexTxOut -> Bool
78     isSuitable now o = case _ciTxOutDatum o of
79         Left _          -> False
80         Right (Datum e) -> case PlutusTx.fromBuiltinData e of
81             Nothing -> False
82             Just d  -> d <= now
83
84 endpoints :: Contract () VestingSchema Text ()
85 endpoints = awaitPromise (give` `select` grab') >> endpoints
86 where
87     give' = endpoint @"give" give
88     grab' = endpoint @"grab" $ const grab
89
90 mkSchemaDefinitions ''VestingSchema
91 mkKnownCurrencies []

```

Compared to the parameterized example we import the undefined parameter with the standard Prelude. Next we create our validator function where the additional parameter carries the payment public key hash. There is now no need to create a vesting parameter type. The deadline is provided in the datum. For the helper functions there is a difference in the *checkSig* function (13) that now uses the *txInfoSignatories* function instead of the *txSignedBy* function. It basically gives you a list of all signatories. The instance for the vesting type also changes where

the datum is now of type *POSIXTime* (20). And same holds true for the wrap function inside the type validator (28). The input parameter for the typed validator is now of type *PaymentPubKeyHash* (23) which is the same for the validator and script address. When constructing our endpoints the grab endpoint does not take in any parameter (44). For the give function the only difference is when we are constructing the transaction we specify the datum and input amount of lovelace (50). For the grab function we use rather the approach as in the *Vesting.hs* code example, where we filter out the UTXOs that have a suitable deadline (62). When we create the lookups we pass to the validator our own payment public key hash as input (68). When declaring the endpoints the grab endpoint has the *const* keyword prepended since it does not take any input parameters.

4 Monads, Traces & Contracts

In this chapter we will focus on the off-chain part of the Plutus contracts. The Plutus prelude contains functions that all have the INLINABLE pragma added which makes it possible to use these functions in the validation code that will then be compiled to Plutus core script. There are many Haskell libraries that weren't written taking into account the Plutus architecture so we can't use them for validation. For this reason the code for the validation script will be relatively simple and will not have many dependencies. The off-chain part however does not get compiled and is just Haskell code so we can use much more features in this code. The wallet code (off-chain code) is written in a special monad called the contract monad.

4.1 Monads

In this chapter we will briefly explain how monads in Haskell work. If you are familiar with this you can skip this chapter. Let's first look at input and output (IO). In Haskell we say to functions that work in the IO context actions. If you have a Haskell function that is not an action you can be sure it will always produce the same result given the same input parameters. This is called referential transparency. We also refer to the term "no side effects" which means that there is nothing from the outside world of a function that could be changing the state of the function. For example we can take a global variable that is used in a function and it could possibly change for setting a global state. For this reason also variables once they are declared in Haskell cannot be changed anymore except if we shadow it. An example of shadowing can be seen here:

```
foo = let x = 1
      in ((let x = 2 in x), x)
```

The final value of foo here is 2, which overwrote the value 1 set in the beginning. But such a case is rarely used in Haskell code. Haskell deals with side effects by using a IO type constructor that takes in one argument and is a member of the monad type class. This is then a IO action which is allowed to have side effects. An example of such an action is when we read some input from the terminal and compute a value from that input. The side effect here is that the input from the terminal can of course change. The monad type class implements 3 operators:

- the bind operator for which we use the `>>=` symbol
- the monad sequencing operator for which we use the `>>` symbol
- the return operator which for which we use the `return` keyword

Let's look at the type signatures of these three operators:

```
(>>=) :: m a -> (a -> m b) -> m b
```

```
(>>) :: m a -> m b -> m b  
return :: a -> m a
```

First we explain the return operator. It simply takes a value of type *a* and puts it into a context. So if we have for instance a String we can put it into a IO context with the following command:

```
ghci> a = return "Haskell" :: IO String  
ghci> :t a  
a :: IO String
```

Next let's explain the sequencing operator. From its type signature we would expect it takes as input two variables in a context and throws away the first one. Which is what it exactly does. This comes useful when you want to chain together function or action calls. For example let's look at code where we put together two *putStrLn* actions:

```
myText :: IO ()  
myText = putStrLn "Learning" >> putStrLn "Haskell"  
  
ghci> myText  
Learning  
Haskell
```

Now let's look at the final operator called bind which is the most useful one. It takes a type parameter in a context, then a function that converts the type specified in the first parameter but without the context to another type in a context which is then the result of this operator. So let's simply declare a variable that is of type *IO Int*, then a function that takes an *Int* and return an *IO Int* by adding 1 to it. And in the end we combine them with the bind operator.

```
monadExample1 :: IO Int  
monadExample1 = do  
    let a = return 1 :: IO Int  
    let func var = return (var + 1) :: IO Int  
    (a >>= func)
```

This function returns an *IO 2*. In case you are wondering what the *do* keyword does, it is a simplification or so called “syntactic sugar” for the following code:

```
monadExample :: IO Int  
monadExample =  
  (\a ->  
   a >>=  
   (\var ->  
     return (var + 1) :: IO Int  
   )) (return 1 :: IO Int)
```

This code has the exact same meaning and also returns *IO* 2. So the *do* keyword allows us to more elegantly write an expression without having to use the lambda functions to connect our expressions. One more thing to mention is the use of the operator *<-* in the *do* notation. It allows us to take a parameter in a context and assign it to a variable without that context.

```
helloName :: IO ()  
helloName = do  
    putStrLn "What is your name?"  
    name <- getLine  
    putStrLn ("Hello " ++ name)  
  
ghci> helloName  
What is your name?  
Luka  
Hello Luka
```

The *getLine* function returns an *IO String*, but the *name* parameter is only of type *String*. You could rewrite this code without the *do* notation in the following form:

```
helloName :: IO ()  
helloName =  
    putStrLn "What is your name?" >>  
    getLine >>=  
    (\name -> return ("Hello " ++ name)) >>=  
    putStrLn
```

What's important to note about the monad type class it supports other data constructors as well, not just *IO*. For instance also *Maybe* and *Lists* are also members of the monad type class. Let's have a look now at the *Maybe* type constructor. Here is the type signature:

```
data Maybe a = Nothing | Just a
```

It returns either a *Nothing* or a *Just* parameterized with the type *a*. Where can this type be used for example? For instance if we import the *readMaybe* function we can read a *String* and get a *maybe* value from it parameterized by a number type as integer.

```
ghci> import Text.Read (readMaybe)  
ghci> readMaybe "42" :: Maybe Int  
42  
ghci> readMaybe "42 + text" :: Maybe Int  
Nothing
```

Now we can show an example where we use the monad operators on a *Maybe* parameter. We want to write a function that takes 3 strings as input, then tries to convert them to integers and sums them together. The result will then be of type *Maybe Int*.

```

1 import Text.Read (readMaybe)
2
3 foo :: String -> String -> String -> Maybe Int
4 foo x y z = case readMaybe x of
5     Nothing -> Nothing
6     Just k -> case readMaybe y of
7         Nothing -> Nothing
8         Just l -> case readMaybe z of
9             Nothing -> Nothing
10            Just m -> Just (k + l + m)
11
12 bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
13 bindMaybe Nothing _ = Nothing
14 bindMaybe (Just x) f = f x
15
16 foo' :: String -> String -> String -> Maybe Int
17 foo' x y z = readMaybe x `bindMaybe` \k ->
18             readMaybe y `bindMaybe` \l ->
19             readMaybe z `bindMaybe` \m ->
20             Just (k + l + m)
21
22 foo'' :: String -> String -> String -> Maybe Int
23 foo'' x y z = threeInts (readMaybe x) (readMaybe y) (readMaybe z)
24
25 threeInts :: Monad m => m Int -> m Int -> m Int -> m Int
26 threeInts mx my mz =
27     mx >>= \k ->
28     my >>= \l ->
29     mz >>= \m ->
30     let s = k + l + m in return s
31
32 threeInts' :: Monad m => m Int -> m Int -> m Int -> m Int
33 threeInts' mx my mz = do
34     k <- mx
35     l <- my
36     m <- mz
37     let s = k + l + m
38     return s

```

First we define our *foo* function with the use of nested case statements (3-10). Then we define a second version *foo'* which uses the helper function *bindMaybe* that basically works as the bind operator and is just defined for maybe values (12-14). With the help of this function and nested lambda expressions we create the *foo'* function (16-20). Now we define a third version called *foo''* that uses the helper function called *threeInts*. It takes as input three integers in a

monad context and with the help of the bind operator and nested lambda functions computes the sum and return the result again within the monad context (25-30). We could of course use the `->` operator which is demonstrated in function `threeInts'` (32-38). Let's look now at another parameterized type called *Either*.

```
data Either a b = Left a | Right b
```

It is usually used in similar situations as `Maybe` but with a more descriptive option of returning another type instead of a `Nothing` if something goes wrong. The common way is to use the `Left` constructor to indicate the error and `Right` to return the result. We can now look at a similar code as before that uses the `Either` type instead of the `Maybe` type.

```
1 import Text.Read (readMaybe)
2
3 readEither :: Read a => String -> Either String a
4 readEither s = case readMaybe s of
5   Nothing -> Left $ "can't parse: " ++ s
6   Just a -> Right a
7
8 foo :: String -> String -> Either String Int
9 foo x y z = case readEither x of
10   Left err -> Left err
11   Right k -> case readEither y of
12     Left err -> Left err
13     Right l -> case readEither z of
14       Left err -> Left err
15       Right m -> Right (k + l + m)
16
17 bindEither :: Either String a -> (a -> Either String b) -> Either String b
18 bindEither (Left err) _ = Left err
19 bindEither (Right x) f = f x
20
21 foo' :: String -> String -> Either String Int
22 foo' x y z = readEither x `bindEither` \k ->
23           readEither y `bindEither` \l ->
24           readEither z `bindEither` \m ->
25           Right (k + l + m)
26
27 foo'' :: String -> String -> Either String Int
28 foo'' x y z = threeInts (readEither x) (readEither y) (readEither z)
```

With the `readEither` function we read a value from a string and return an error message if the string cannot be parsed to a value. The first version of the `foo` function uses again the nested case statements. Then we declare the `bindEither` function that has same structure as the

previous `bindMaybe` function. In our second version the `foo'` function we use `readEither`, `bindEither` and nested lambda functions similar to the maybe example. In the third version the `foo''` function has very little modifications compared to the maybe example. We replace only the read function with the either version but the `threeInts` function call stays the same. The reason for this is that it works purely with the monad type class and is not tied to a specific parameterized type as a maybe or a either type. Of course either is also a member of the monad type class which makes it possible for this function to work with either variables. Let's look now at the `Writer.hs` example where computations can produce also log outputs.

```

1  import Control.Monad
2
3  data Writer a = Writer a [String]
4      deriving Show
5
6  number :: Int -> Writer Int
7  number n = Writer n $ ["number: " ++ show n]
8
9  tell :: [String] -> Writer ()
10 tell xs = Writer () xs
11
12 foo :: Writer Int -> Writer Int -> Writer Int -> Writer Int
13 foo (Writer k xs) (Writer l ys) (Writer m zs) =
14     let
15         s = k + l + m
16         Writer _ us = tell ["sum: " ++ show s]
17     in
18         Writer s $ xs ++ ys ++ zs ++ us
19
20 bindWriter :: Writer a -> (a -> Writer b) -> Writer b
21 bindWriter (Writer a xs) f =
22     let
23         Writer b ys = f a
24     in
25         Writer b $ xs ++ ys
26
27 foo' :: Writer Int -> Writer Int -> Writer Int -> Writer Int
28 foo' x y z = x `bindWriter` \k ->
29             y `bindWriter` \l ->
30             z `bindWriter` \m ->
31             let s = k + l + m
32                 in tell ["sum: " ++ show s] `bindWriter` \_ ->
33                     Writer s []
34

```

```

35 foo'': Writer Int -> Writer Int -> Writer Int -> Writer Int
36 foo'' x y z = do
37     s <- threeInts x y z
38     tell ["sum: " ++ show s]
39     return s
40
41 instance Functor Writer where
42     fmap = liftM
43
44 instance Applicative Writer where
45     pure = return
46     ( <*>) = ap
47
48 instance Monad Writer where
49     return a = Writer a []
50     ( >>=) = bindWriter

```

First we define our writer type and derive show. It contains the type variable and a list of strings that represents a log message. Second we define our *number* function which transforms an integer to a *Writer* type by adding a short description. Then we define the *tell* function that creates a writer from a string. For our *foo* function we take in three logging parameters of type *Writer Int* and produce a logging parameter of the same type. The function sums up the three parameters, concatenates the log messages and adds a “sum” log at the end. Next we write the *bindWriter* function that works as the bind operator for our *Writer* type. We can define our second version the *foo'* function with the bind function, four lambda function and the *tell* function. In the last example we use again our *threeInts* function but in order for this to work we need to make our writer type an instance of monad. Because the functor type class is a superclass of applicative and applicative is a superclass of monad we need to make also instances for those type classes. When defining the functor and applicative instances we use functions from the *Control.Monad* module that have basically same type signatures as functions that belong to the applicative and functor type classes. The code in lines 35 to 39 we could also rewrite without the do notation and it would look like this:

```

foo'': Writer Int -> Writer Int -> Writer Int -> Writer Int
foo'' x y z =
    threeInts x y z >>= \s ->
    tell ["sum: " ++ show s] >>
    return s

```

4.2 The emulator trace monad

The emulator trace monad enables you to run your code from the Repl instead of running the code on the Plutus playground. The word trace here defines actions that we would normally performe on the playground (e.g. defining initial conditions, performing give and grab actions). The *Emulator.Trace* type is defined in the *Plutus.Trace.Emulator* module. It is defined with the effect *Eff* monad. We run an emulator trace with the *runEmulatorTrace* function (Figure 10).

```
runEmulatorTrace :: EmulatorConfig -> EmulatorTrace () -> ([EmulatorEvent], Maybe EmulatorErr, EmulatorState)
```

Run an emulator trace to completion, returning a tuple of the final state of the emulator, the events, and any error, if any.

Figure 10 - Run emulator trace function

It takes in as input an *EmulatorConfig* and *EmulatorTrace* parameters. The emulator config (Figure 11) is an instance of the *Default* type class so if we import *Data.Default* we can make an default parameter. The command *def :: EmulatorConfig* will return an initial distribution of 10 wallets with each of them containing a 100 ADA with a default configuration for slots and fees.

```
data EmulatorConfig # Source

Constructors

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.
  _slotConfig :: SlotConfig Set the start time of slot 0 and the length of one slot
  _feeConfig :: FeeConfig Configure the fee of a transaction
```

Figure 11 - Emulator config

There is also another function called *runEmulatorTracIO* that does not take in a config, it just uses the default configuration and it doesn't return the triple since it runs in IO. You can try to run it in the Repl with a unit emulator trace like this:

```
ghci> import Plutus.Trace.Emulator
ghci> import Data.Default
ghci> runEmulatorTracIO $ return ()
```

When running this function we get and output that first displays the genesis transaction that distributes the initial funds. Then we get a wait of 2 slots and after that we get the final balances. And because we didn't specify any transactions each of the 10 wallet has a 100 ADA. You usually specify an emulator config if you want to have more or less wallets or different initial funds. There is a variation of the IO function called *runEmulatorTracIO'*.

```
runEmulatorTraceIO' :: TraceConfig -> EmulatorConfig -> EmulatorTrace () -> IO ()
```

It takes in two additional parameters. The trace config has the following type signature.

```
TraceConfig
  :: (Wallet.Emulator.MultiAgent.EmulatorEvent' -> Maybe String)
    -> GHC.IO.Handle.Types.Handle -> TraceConfig
```

The first parameter is a function that basically filters out events that we are interested in and the second parameter is a handle with could represent the console standard output or error or it could be a file handle. We could use the file handle for displaying the output of the IO' function in a file. Now let's look at the *Trace.hs* file where we define a trace.

```
1  {-# LANGUAGE TypeApplications #-}
2  {-# LANGUAGE DataKinds          #-}
3
4  module Week04.Trace where
5
6  import Control.Monad.Freer.Extras as Extras
7  import Data.Default                (Default(..))
8  import Data.Functor                (void)
9  import Ledger.TimeSlot
10 import Plutus.Trace
11 import Wallet.Emulator.Wallet
12
13 import Week04.Vesting
14
15 -- EmulatorTrace a
16
17 test :: IO ()
18 test = runEmulatorTraceIO myTrace
19
20 myTrace :: EmulatorTrace ()
21 myTrace = do
22     h1 <- activateContractWallet (knownWallet 1) endpoints
23     h2 <- activateContractWallet (knownWallet 2) endpoints
24     callEndpoint @"give" h1 $ GiveParams
25         { gpBeneficiary = mockWalletPaymentPubKeyHash $ knownWallet 2
26         , gpDeadline    = slotToBeginPOSIXTime def 20
27         , gpAmount      = 10000000
28         }
29     void $ waitUntilSlot 20
30     callEndpoint @"grab" h2 ()
31     s <- waitNSlots 2
32     Extras.logInfo $ "reached " ++ show s
```

In addition to some standard Plutus modules we import also the *Vesting.hs* example (13). Then we define the emulator trace monad with a do block (21). Before we can call an endpoint we have to start the contract itself which we do in lines 22 and 23. As input parameter the function *activateContractWallet* takes the wallet on which to activate a contract and the contract itself. The result we get is a contract handle. Then we call an endpoint with the *callEndpoint* function where we pass in a type level string which is defined with the @ symbol and works because we imported the language extension *TypeApplications*. We also pass in the handle and the input parameters (24-28). Then we wait until slot 20 where we use the *void* function that transforms a result to unit (29) and then call the grab endpoint from the wallet 2 (30). After that we wait another 2 slots and log a message. To run this example execute *test* from the repl.

4.3 The contract monad

The contract monad comes with four type parameters w, s, e and a. The a represents the result. The w allows the contract to write log messages. The purpose of this logging is to communicate between different contracts or pass information to the outside world. The s specifies what endpoints are available in the contract. The e specifies the type of error messages. Let's have a look now at some contract monad examples from the *Contract.hs* file.

```

1 {-# LANGUAGE OverloadedStrings #-}
2 {-# LANGUAGE TypeApplications #-}
3 {-# LANGUAGE DataKinds #-}
4 {-# LANGUAGE TypeOperators #-}
5
6 module Week04.Contract where
7
8 import Control.Monad.Freer.Extras as Extras
9 import Data.Functor                (void)
10 import Data.Text                   (Text, unpack)
11 import Data.Void                  (Void)
12 import Plutus.Contract           as Contract
13 import Plutus.Trace.Emulator    as Emulator
14 import Wallet.Emulator.Wallet
15
16 -- Contract w s e a
17
18 myContract1 :: Contract () Empty Text ()
19 myContract1 = do
20     void $ Contract.throwError "BOOM!"
21     Contract.logInfo @String "hello from the contract"
22

```

```

23 myTrace1 :: EmulatorTrace ()
24 myTrace1 = void $ activateContractWallet (knownWallet 1) myContract1
25
26 test1 :: IO ()
27 test1 = runEmulatorTraceIO myTrace1
28
29 myContract2 :: Contract () Empty Void ()
30 myContract2 = Contract.handleError
31     (\err -> Contract.logError $ "caught: " ++ unpack err)
32     myContract1
33
34 myTrace2 :: EmulatorTrace ()
35 myTrace2 = void $ activateContractWallet (knownWallet 1) myContract2
36
37 test2 :: IO ()
38 test2 = runEmulatorTraceIO myTrace2
39
40 type MySchema = Endpoint "foo" Int .\| Endpoint "bar" String
41
42 myContract3 :: Contract () MySchema Text ()
43 myContract3 = do
44     awaitPromise $ endpoint @"foo" Contract.logInfo
45     awaitPromise $ endpoint @"bar" Contract.logInfo
46
47 myTrace3 :: EmulatorTrace ()
48 myTrace3 = do
49     h <- activateContractWallet (knownWallet 1) myContract3
50     callEndpoint @"foo" h 42
51     callEndpoint @"bar" h "Haskell"
52
53 test3 :: IO ()
54 test3 = runEmulatorTraceIO myTrace3
55
56 myContract4 :: Contract [Int] Empty Text ()
57 myContract4 = do
58     void $ Contract.waitNSlots 10
59     tell [1]
60     void $ Contract.waitNSlots 10
61     tell [2]
62     void $ Contract.waitNSlots 10
63
64 myTrace4 :: EmulatorTrace ()
65 myTrace4 = do
66     h <- activateContractWallet (knownWallet 1) myContract4
67

```

```

68      void $ Emulator.waitNSlots 5
69      xs <- observableState h
70      Extras.logInfo $ show xs
71
72      void $ Emulator.waitNSlots 10
73      ys <- observableState h
74      Extras.logInfo $ show ys
75
76      void $ Emulator.waitNSlots 10
77      zs <- observableState h
78      Extras.logInfo $ show zs
79
80  test4 :: IO ()
81  test4 = runEmulatorTraceIO myTrace4

```

For the first example (18) we choose a contract where we do not want to write any log messages so we choose unit for w type parameter. We also do not need any endpoints so we choose the *Empty* type. For error messages we choose the type text. And we are also not interested in the result so we put a unit for type parameter a. First we throw an error message with the *throwError* function for which the input is of type Text (20). There are two possible *throwError* functions one from *Plutus.Trace.Emulator* module and the other from *Plutus.Contract* module. So we have to specify which one we are using. Then we log a message (21), which should not be confused with the w type variable which uses also a *logInfo* function from the *Control.Monad.Freer.Extras* module. This is just simple logging. The *logInfo* function is polymorphic which means it can take as input parameters various types so we specify the type of the string as @String. It could also be of type Text since we imported the *Data.Text* module and the *Overloaded* extension. We test the contract with *myTrace1* (23). There we activate the contract wallet and do not care about the result (24) and in the test statement we run the trace (26-27). For this example if we run it the log message from line 21 will not be shown because the contract will stop when the error on line 20 is thrown. So the execution of a contract is stopped at the point where an exception is raised.

In our next example we show how to catch and handle exceptions. We will run the contract 1 but catch the exception. Contract 2 has the e variable of Void which means that it can't throw an error because void does not contain any variables in contrast to unit which contains one variable also called unit. The *handleError* function has the following type signature:

```
handleError :: (e -> Contract w s e' a) -> Contract w s e a -> Contract w s e' a
```

It takes a function and a contract as input parameters and if there is no error it returns the result of type a, if there is however an error the function is applied to the error type variable

and the contract produced by the function will be executed. When we call *handleError* we first provide the function (31) where the *err* variable is of type Text and the lambda function creates a contract where we only log the error which is now of type string. And then we add the first contract as input parameter. If we run now the test2 action we will catch the error, which means contract 1 will not be executed and instead the contract from the lambda function is executed. Let's look now at the third example where we take the *s* type parameter in account.

First we define a type synonym for the schema where endpoint types are declared (40). The “foo” and “bar” in this declarations are types not strings which is possible due to the *DataKinds* language extension. We use the type operator “. $\backslash\wedge$ ” that enables us to combine more endpoints and for that we need the *TypeOperators* language extension. Then we define contract 3 where we use the *endpoint* function.

```
endpoint :: (a -> Contract w s e b) -> Promise w s e b
```

This function takes in a function that changes the endpoint value *a* (in our case an *Int* or a *String*) to a contract and then returns a promise parameterized by the same types. A promise is a blocked contract waiting to be triggered by an outside stimulus, which would be in our case when a user tries to call the endpoint from his wallet and with an integer or a string as input. So when we call the *endpoint* function we first tell it which endpoint to invoke i.e. what will be the type of *a* (in our case an integer or a string) and then provide it with *logInfo* function which can take an integer or string and produce a contract that just logs that variable. With the *awaitPromise* function we turn a promise into a contract (44-45). In the trace action we now need the handle to be able to call an endpoint (49). The *callEndpoint* function takes in a contract handle and an endpoint value and then it invokes this endpoint. First we need to specify which endpoint to call and then we pass in our two parameters (50-51). If we look now at the off-chain code of the *Vesting.hs* contract we see how the give and grab contracts are called inside the endpoint contract. What we didn't describe in this example is the use of the *select* function that is used in the vesting endpoint contract. It takes in two contracts and returns the contract that makes progress first, discarding the other one.

In the final example 4 we look at the *w* type parameter that is responsible for communication between different contracts and the outside world. The *w* type cannot be of an arbitrary type but a type that is an instance of the type class monoid. We will use a list of integers which is an instance of monoid (56). In the body of our contract we first define a wait for 10 slots and throw away the result (58). Then we use the tell function that has following type signature:

```
tell :: w -> Contract w s e ()
```

So we provide a writer and the `tell` function creates a contract. We repeat this step two times (58-62). In the emulator trace we first activate the contract and then wait for 5 slots (66-68). We look up the state of a running contract with the `observableState` function. As argument it takes a handle of the running contract. Then we log the result we get (70). When we test this example we see that the first log returns just an empty list [], the second log returns [1] and the third log returns [1,2]. The state in the beginning is a *mempty* monoid and then each time you call the `tell` function the `mappend` function is used to update the state. On the real blockchain the user can interact with the contract by invoking endpoints and the contract can communicate back its state with the use of the `tell` function.

4.4 Homework

For this homework we want to write an emulator trace for a simple contract that pays some ADA to a give public key. The trace should take in 2 integer parameters that represent the amount of lovelace and it should call the pay contract twice. The recipient should be wallet two. Here is the code from the `Solution.hs` file.

```

1  {-# LANGUAGE DataKinds          #-}
2  {-# LANGUAGE DeriveAnyClass     #-}
3  {-# LANGUAGE DeriveGeneric      #-}
4  {-# LANGUAGE NumericUnderscores #-}
5  {-# LANGUAGE OverloadedStrings  #-}
6  {-# LANGUAGE TypeApplications   #-}
7  {-# LANGUAGE TypeOperators      #-}
8
9  module Week04.Solution where
10
11 import Data.Aeson           (FromJSON, ToJSON)
12 import Data.Functor          (void)
13 import Data.Text              (Text, unpack)
14 import GHC.Generics          (Generic)
15 import Ledger
16 import Ledger.Ada            as Ada
17 import Ledger.Constraints    as Constraints
18 import Plutus.Contract        as Contract
19 import Plutus.Trace.Emulator as Emulator
20 import Wallet.Emulator.Wallet
21
22 data PayParams = PayParams
23   { ppRecipient :: PaymentPubKeyHash
24   , ppLovelace  :: Integer
25   } deriving (Show, Generic, FromJSON, ToJSON)

```

```

26
27 type PaySchema = Endpoint "pay" PayParams
28
29 payContract :: Contract () PaySchema Text ()
30 payContract = do
31     pp <- awaitPromise $ endpoint @"pay" return
32     let tx = mustPayToPubKey (ppRecipient pp) $ lovelaceValueOf $
33         ppLovelace pp
34     handleError (\err -> Contract.logInfo $ "caught error: " ++ unpack err) $ void $ submitTx tx
35
36 payTrace :: Integer -> Integer -> EmulatorTrace ()
37 payTrace x y = do
38     h <- activateContractWallet (knownWallet 1) payContract
39     let pkh = mockWalletPaymentPubKeyHash $ knownWallet 2
40     callEndpoint @"pay" h $ PayParams
41         { ppRecipient = pkh
42         , ppLovelace = x
43         }
44     void $ Emulator.waitNSlots 1
45     callEndpoint @"pay" h $ PayParams
46         { ppRecipient = pkh
47         , ppLovelace = y
48         }
49     void $ Emulator.waitNSlots 1
50
51 payTest1 :: IO ()
52 payTest1 = runEmulatorTraceIO $ payTrace 10_000_000 20_000_000
53
54 payTest2 :: IO ()
55 payTest2 = runEmulatorTraceIO $ payTrace 1000_000_000 20_000_000

```

First we create the type *PayParams* where we specify the recipient and the amount of lovelace we want to pay (22-25). Then we define the schema which has only the pay endpoint available. Next we define our pay contract that is parameterized by our schema and uses the Text type for the error message. In the first line of our contract we invoke the pay endpoint by immediately returning contract without any side effects (31). This means that we return the input parameters defined in the *PayParams* data type that are in a contract monad context. Next we create a transaction where we use the *mustPayToPubKey* function which takes in a public key hash (32). Then we handle the error when we submit the transaction. In our case the error will be triggered when we try to submit a transaction with a larger amount of lovelace than the wallet has available (33). Our trace will take in two parameters that represent the amount of

lovelace we want to pay. First we activate the contract and then we define the payment public key hash of wallet 2 (38-39).

Then we call our endpoints with the give parameters and wait for one slot after each endpoint call (40-49). In the end we create two IO actions that run our emulator traces with 2 different values. In the first run both payments should succeed but in the second run the first payment should produce an error because it tries to spend more funds as there are available. This error should be caught and the second payment should still be processed. Note that for the input values we are using numbers with underscores which is possible because we imported the *NumericUnderscores* language extension.

5 Native tokens

To be able to work with native tokens and ADA we need the modules *Plutus.V1.Ledger.Ada* and *Plutus.V1.Ledger.Value*. First we look at the value module where the type *Value* is defined.

Constructors

Value

getValue :: Map CurrencySymbol (Map TokenName Integer)

Figure 12 - The Value type

Token name and currency symbol are just wrappers for the type *BuitlingByteString* that represents a byte string. These two byte strings define a native token or coin. The integer represents the amount of the currency. There is also another type called AssetClass (Figure 13).

Constructors

AssetClass

unAssetClass :: (CurrencySymbol, TokenName)

Figure 13 - Asset class

It defines a asset class which is a native token or coin. A Value just says how many units of an asset class are contained in it. Now let's look at some examples from the Repl.

```
ghci> import Plutus.V1.Ledger.Value
ghci> import Plutus.V1.Ledger.Ada
ghci> :set -XOverloadedStrings
ghci> :t adaSymbol
adaSymbol :: CurrencySymbol
ghci> adaSymbol

ghci> :t adaToken
adaToken :: TokenName
ghci> adaToken
"""

ghci> :t lovelaceValueOf
lovelaceValueOf :: Integer -> Value
ghci> lovelaceValueOf 123
Value (Map [(",Map [("",123)])])
```

We import the two modules and activate the language extension so we can enter byte strings as literal strings. Because both currency symbol and token name implement the *IsString* class we can enter both of them as literal strings. We see that the ADA currency symbol and token

name are just an empty byte string. We can create a value of 123 lovelace with the function *lovelaceValueOf*. Next we combine values and also create values that includes tokens.

```
ghci> lovelaceValueOf 123 <> lovelaceValueOf 10
Value (Map [(,Map [("",133)])])
ghci> :t singleton
singleton :: CurrencySymbol -> TokenName -> Integer -> Value
ghci> singleton "a8ff" "ABC" 7
Value (Map [(a8ff,Map [("ABC",7)])])
ghci> singleton "a8ff" "ABC" 7 <> lovelaceValueOf 42 <> singleton "a8ff" "XYZ" 100
Value (Map [(,Map [("",42)]),(a8ff,Map [("ABC",7),("XYZ",100)])])
ghci> let v = it
ghci> :t valueOf
valueOf :: Value -> CurrencySymbol -> TokenName -> Integer
ghci> valueOf v "a8ff" "XYZ"
100
ghci> valueOf v "a8ff" "abc"
0
ghci> :t flattenValue
flattenValue :: Value -> [(CurrencySymbol, TokenName, Integer)]
ghci> flattenValue v
[(,"",42),(a8ff,"XYZ",100),(a8ff,"ABC",7)]
```

Since *Value* is an instance of monoid and semigroup is a superclass of monoid we can use the *<>* operator for combining values. With the *singleton* function we can construct a value of just one asset class. For the currency symbol we can't use an arbitrary string. Instead we must use a hexadecimal value. We can also combine values that contain different native tokens and ADA. With the *it* keyword we get the previous result. The *valueOf* function extracts the given amount of lovelace or a token. The *flattenValue* function takes a value and returns a list of triples that contain information from the value parameter.

In general a transaction can't delete or create tokens. The inputs equal the outputs if we also take in account fees that need to be paid. Fees depend on the size of a transaction in bytes and on the scripts that need to be run to validate a transaction. The script takes more fees if it consumes more memory. The minting policies are responsible for managing tokens. The hexadecimal value that is the currency symbol represents the hash of the minting policy script. If we want to create or burn tokens the minting script has to be contained in the transaction which is then executed together with the validation script. The purpose of the minting script is to decide whether the given transaction is allowed to mint or burn tokens. And for ADA which has an empty string for the script hash, that means that there is no script which would allow minting or burning of ADA. All the ADA that exists comes from the genesis block and from monetary expansion. After each epoch rewards are paid and parts of these rewards come from monetary expansion where certain percentage of remaining reserves gets paid as rewards. The total amount of ADA in the system is fixed, it can never change.

5.1 Simple Minting Policy

So far when we looked at the `ScriptPurpose` type variable of the `ScriptContext` data type we were only concerned with the `Spending` constructor that takes as input a transaction reference. In the transaction information `TxInfo` data type we can specify a `Value` data type for the `txInfoMint` variable. Minting policies are triggered if this field contains a non-zero value. For each currency symbol defined in this field the corresponding minting policy is triggered. A minting policy has only two inputs, the redeemer and the context. The script purpose is then set to `Minting`. All of the minting policies contained in a transaction have to pass in order that the transaction passes, otherwise it fails. Let's look at a simple example now.

```
1 import           GHC.Generics          (Generic)
2 import           Plutus.Contract       as Contract
3 import           Plutus.Trace.Emulator as Emulator
4 import qualified PlutusTx
5 import           PlutusTx.Prelude     hiding (Semigroup(..), unless)
6 import           Ledger                 hiding (mint, singleton)
7 import           Ledger.Constraints   as Constraints
8 import qualified Ledger.Typed.Scripts as Scripts
9 import           Ledger.Value         as Value
10 import          Playground.Contract  (printJson, printSchemas, stage,
11                                         ensureKnownCurrencies, ToSchema)
12 import          Playground.TH        (mkKnownCurrencies,
13                                         mkSchemaDefinitions)
14 import          Playground.Types     (KnownCurrency(..))
15 import          Prelude               (IO, Show(..), String)
16 import          Text.Printf          (printf)
17 import          Wallet.Emulator.Wallet
18
19 {-# INLINABLE mkPolicy #-}
20 mkPolicy :: () -> ScriptContext -> Bool
21 mkPolicy () _ = True
22
23 policy :: Scripts.MintingPolicy
24 policy = mkMintingPolicyScript $$ (PlutusTx.compile
25                                     [|| Scripts.wrapMintingPolicy mkPolicy ||])
26
27 curSymbol :: CurrencySymbol
28 curSymbol = scriptCurrencySymbol policy
29
30 data MintParams = MintParams
31   { mpTokenName :: !TokenName
32   , mpAmount    :: !Integer
33   } deriving (Generic, ToJSON, FromJSON, ToSchema)
```

```

31
32 type FreeSchema = Endpoint "mint" MintParams
33
34 mint :: MintParams -> Contract w FreeSchema Text ()
35 mint mp = do
36     let val      = Value.singleton curSymbol (mpTokenName mp) (mpAmount mp)
37     lookups = Constraints.mintingPolicy policy
38     tx      = Constraints.mustMintValue val
39     ledgerTx <- submitTxConstraintsWith @Void lookups tx
40     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
41     Contract.logInfo @String $ printf "forged %s" (show val)
42
43 endpoints :: Contract () FreeSchema Text ()
44 endpoints = mint' >> endpoints
45 where
46     mint' = awaitPromise $ endpoint @"mint" mint
47
48 mkSchemaDefinitions ''FreeSchema
49
50 mkKnownCurrencies []
51
52 test :: IO ()
53 test = runEmulatorTraceIO $ do
54     let tn = "ABC"
55     h1 <- activateContractWallet (knownWallet 1) endpoints
56     h2 <- activateContractWallet (knownWallet 2) endpoints
57     callEndpoint @"mint" h1 $ MintParams
58         { mpTokenName = tn
59         , mpAmount   = 555
60         }
61     callEndpoint @"mint" h2 $ MintParams
62         { mpTokenName = tn
63         , mpAmount   = 444
64         }
65     void $ Emulator.waitNSlots 1
66     callEndpoint @"mint" h1 $ MintParams
67         { mpTokenName = tn
68         , mpAmount   = -222
69         }
70     void $ Emulator.waitNSlots 1

```

First we create the *mkPolicy* function that represents the policy and will also be compiled to Plutus Core (35-37). We use unit for the redeemer and return a bool, which is used in the typed version. Our minting policy will always return *True* no matter the context.

Then we compile the minting policy (39-40). We use the `mkMintingPolicyScript` function to create the minting policy script and with the `wrapMintingPolicy` function we convert our typed version to an un-typed version before we compile it. Because we use this function we also need to add the `INLINABLE` pragma before the `mkPolicy` function. Next we can get the currency symbol with the function `scriptCurrencySymbol` where we need to provide as input the compiled minting policy. The output is the hash of the script. Similar to how we parameterized validators we could also do this with minting policies. This completes the on-chain part.

For the off-chain part we first declare our minting parameters where we specify the token name and the amount (45-48). If the amount is a positive number we mint tokens and if it is a negative number we burn tokens. Then we define the schema with only one endpoint called `mint` (50). The `mint` contract takes minting parameters as input. In the body of the contract we first compute the `Value` that we want to forge (54). As lookups we specify the minting policy. The only constrain we put on our transaction is `mustMintValue` function that takes as input the previously defined `Value`. We can skip the redeemer since it has the value of unit. When we submit the transaction it will automatically transfer the minted value to the wallet if it is positive. If it is negative it will try to find sufficiently many tokens in the users wallet that will then be burned. After that we wait for confirmation and then log a message. The endpoint contract then follows the same pattern as in previous examples. In the end we define a emulator trace where we mint and burn some tokens in wallet 1 and 2 (70-88). One thing to notice is that when we try this out on the playground the UTXO that gets assigned the tokens will also get 2 ADA assigned which is the minimum UTXO value of ADA. For the actual Cardano blockchain this value can be different.

5.2 More Realistic Minting Policy

Now we will look at a minting policy that is parameterized by a payment public key hash. The minting or burning of the tokens will only be allowed if the owner of that key has signed the transaction. Here is the code from the `Signed.hs` example.

```
1  {-# INLINABLE mkPolicy #-}
2  mkPolicy :: PaymentPubKeyHash -> () -> ScriptContext -> Bool
3  mkPolicy pkh () ctx = txSignedBy (scriptContextTxInfo ctx) $
4                                unPaymentPubKeyHash pkh
5
6  policy :: PaymentPubKeyHash -> Scripts.MintingPolicy
7  policy pkh = mkMintingPolicyScript $
8      $$($PlutusTx.compile [|| Scripts.wrapMintingPolicy . mkPolicy ||])
9      `PlutusTx.applyCode`
10     PlutusTx.liftCode pkh
```

```

10
11 curSymbol :: PaymentPubKeyHash -> CurrencySymbol
12 curSymbol = scriptCurrencySymbol . policy
13
14 data MintParams = MintParams
15   { mpTokenName :: !TokenName
16     , mpAmount    :: !Integer
17   } deriving (Generic, ToJSON, FromJSON, ToSchema)
18
19 type FreeSchema = Endpoint "mint" MintParams
20
21 mint :: MintParams -> Contract w FreeSchema Text ()
22 mint mp = do
23   pkh <- Contract.ownPaymentPubKeyHash
24   let val      = Value.singleton (curSymbol pkh) (mpTokenName mp)
        (mpAmount mp)
25   lookups = Constraints.mintingPolicy $ policy pkh
26   tx      = Constraints.mustMintValue val
27   ledgerTx <- submitTxConstraintsWith @Void lookups tx
28   void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
29   Contract.logInfo @String $ printf "forged %s" (show val)

```

We add only the functions and parameters that changed compared to the previous *Free.hs* example. First we create our policy function that takes now the payment public key hash as an additional parameter (1-3). We use the *txSignedBy* function to check whether the transaction was signed with the given public key. Here is its type signature.

```
txSignedBy :: TxInfo -> PubKeyHash -> Bool
```

Next we compile the policy function where we use now the same pattern as in our *Parameterized.hs* code (5-9). Then we create the currency symbol which now takes in the public key as a parameter (11-12). For the off-chain code we declare the minting parameters and the schema the same way as in the *Free.hs* code. And for the mint function what changes is that we first look up our own payment public key hash for which we use the function from the *Contract* module to avoid collision for the same function from another module. Then we also change the value and the lookups declaration. When defining the value we add to the currency symbol our public payment key hash as input parameter and for the lookups we do the same for the policy parameter (24-25).



When writing Plutus code try to compile it as often as possible to avoid multiple errors at the end when running it in the playground.

If we run now the test trace we notice something interesting. When checking the balances of the wallets we see that the currency symbol for the token that wallet one and two got, is different. That was not the case with our first example. The token name is still the same. The reason for this is that the minting script is now longer a constant but is a function so the same is true for the hash of the script which represents the currency symbol. And because the input parameter is the wallet public payment key hash which is different for wallet 1 and wallet 2 also the currency symbol for those two wallets gets computed differently. So the wallets contain now two different tokens or asset classes.

5.3 NFT-s

Non fungible tokens can exist only once, which means there is only one token in existence. This means we have to put a constraint on our transaction and say that only one token is allowed to be minted and that the transaction that triggers the mint can be executed only once. In order to do that we need something on the Cardano blockchain that we can refer to in our minting policy and is unique i.e. it exists only in one transaction. And the trick is we use a UTXO, since it can exist only once and when consumed in a transaction it is never again available. If there is another UTXO sitting at the same address with the same value and datum it still has another ID. We identify an UTXO with the transaction ID that produced it and the index it got assigned, since a transaction can produce more than one UTXO. And transactions are unique, there can never be the same transaction again. They are unique because they spend fees which come from UTXOs and they themselves are unique. Which brings us to an induction. So the idea is that we name a specific UTXO as parameter to our minting policy and then we check that the transaction that does the minting consumes this UTXO. Let's look at the *NFT.hs* code.

```

1  -- import Playground.Contract (printJson, printSchemas,
2                                ensureKnownCurrencies, stage, ToSchema)
3  -- import Playground.TH      (mkKnownCurrencies, mkSchemaDefinitions)
4  -- import Playground.Types   (KnownCurrency(..))
5  import qualified Data.Map as Map
6  import           Prelude  (IO, Semigroup(..), Show(..), String)
7
8  {-# INLINABLE mkPolicy #-}
9  mkPolicy :: TxOutRef -> TokenName -> () -> ScriptContext -> Bool
10 mkPolicy oref tn () ctx = traceIfFalse
11                           "UTxO not consumed" hasUTxO &&
12                           traceIfFalse "wrong amount minted"
13                           checkMintedAmount
14
15 where
16   info :: TxInfo

```

```

13     info = scriptContextTxInfo ctx
14
15     hasUTxO :: Bool
16     hasUTxO = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs info
17
18     checkMintedAmount :: Bool
19     checkMintedAmount = case flattenValue (txInfoMint info) of
20         [(_, tn', amt)] -> tn' == tn && amt == 1
21         _ -> False
22
23     policy :: TxOutRef -> TokenName -> Scripts.MintingPolicy
24     policy oref tn = mkMintingPolicyScript $
25         $$($PlutusTx.compile [|| \oref' tn' -> Scripts.wrapMintingPolicy $ mkPolicy oref' tn' ||])
26         `PlutusTx.applyCode` PlutusTx.liftCode oref
27         `PlutusTx.applyCode` PlutusTx.liftCode tn
28
29
30     curSymbol :: TxOutRef -> TokenName -> CurrencySymbol
31     curSymbol oref tn = scriptCurrencySymbol $ policy oref tn
32
33
34     data NFTParams = NFTParams
35         { npToken    :: !TokenName
36         , npAddress :: !Address
37         } deriving (Generic, FromJSON, ToJSON, Show)
38
39     type NFTSchema = Endpoint "mint" NFTParams
40
41     mint :: NFTParams -> Contract w NFTSchema Text ()
42     mint np = do
43         utxos <- utxosAt $ npAddress np
44         case Map.keys utxos of
45             []      -> Contract.logError @String "no utxo found"
46             oref : _ -> do
47                 let tn      = npToken np
48                 let val    = Value.singleton (curSymbol oref tn) tn 1
49                 lookups = Constraints.mintingPolicy (policy oref tn) <>
50                         Constraints.unspentOutputs utxos
51                 tx      = Constraints.mustMintValue val <>
52                         Constraints.mustSpendPubKeyOutput oref
53                 ledgerTx <- submitTxConstraintsWith @Void lookups tx
54                 void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
55                 Contract.logInfo @String $ printf "Forged %s" (show val)

```

```

55 endpoints :: Contract () NFTSchema Text ()
56 endpoints = mint' >> endpoints
57 where
58     mint' = awaitPromise $ endpoint @"mint" mint
59
60 test :: IO ()
61 test = runEmulatorTraceIO $ do
62     let tn = "ABC"
63         w1 = knownWallet 1
64         w2 = knownWallet 2
65         h1 <- activateContractWallet w1 endpoints
66         h2 <- activateContractWallet w2 endpoints
67         callEndpoint @"mint" h1 $ NFTParams
68             { npToken    = tn
69             , npAddress = mockWalletAddress w1
70             }
71         callEndpoint @"mint" h2 $ NFTParams
72             { npToken    = tn
73             , npAddress = mockWalletAddress w2
74             }
75     void $ Emulator.waitNSlots 1

```

Compared to our previous code example we do not use the Playground modules and we add the Map module and add Semigroup to the Prelude module. In the *mkPolicy* function we have now 2 additional parameters, the transaction output reference and the token name (8). In the body of this function we check that the UTXO is consumed and that we mint only 1 token. For the helper functions we first get the script context info. Then for the *hasUTxO* parameter we use the *txInfoInputs* helper function of the *TxInfo* type that gives us a list of transaction inputs that are of type *TxInInfo* (15-16). This type has the following structure (Figure 14).



Figure 14 - Transaction input info

For the *checkMintedAmount* parameter we check that that the list we get from the minted info field contains only one triple and that the name and quantity match (18-21). Next we define our policy that has now the transaction output reference and token name as input parameters and same holds true for the currency symbol (23-32).

For the off-chain code we create our NFT parameters type variable where we define the token name and the contract address (34-37). In the schema we define only one endpoint called mint (39). In the body of the mint contract we first lookup all the UTXOs sitting at the given address (43). In the map that we get the keys that represent the *txOutRefs*. If there are none we log an error message and if there is at least one we take the 1st one (it could be any one). After that we get the token name and define the value, lookups and transaction. For the currency symbol and policy we have to provide our input parameters. For the lookups in the *unspentOutputs* function we could specify only the one UTXO that we need but we can also give a Map of UTXOs and the one we need just has to be contained in the Map (49). For the transaction we put a constrain that the given value must be minted and the transaction we are constructing spends the given output reference for our UTXO (50). This line of code ensures that the transaction can be performed only one, since we are referencing in a transaction a UTXO that will be unavailable after this transaction completes. After that we submit the transaction, wait for confirmation and log a message. Our endpoints definition follows the same pattern as before (55-58). In our emulator trace we define a token name and start the contract for wallet one and wallet two. Then we call the mint endpoint for the first and second wallet and wait for one slot that the transactions can be processed. The *mockWalletAddress* functions takes as input a wallet and produces an address. This works only for playground wallets. And again when we run the code we end up with two different currency symbols which means the NFTs that wallet 1 and 2 contain are unique.

5.4 Homework

For the first homework we want to implement a Marry era style monetary policy. What you are allowed to do is that you can specify signatures that have to be present in the minting transaction and specify the deadline that says minting can only happen before a certain deadline. So we want to create a minting policy that has 2 parameters, a public key hash and POSIX time. And the transaction should only succeed if it is signed by the corresponding signature and if the deadline has not passed. Let's look at the code *Solution1.hs*.

```

1  import Data.Default      (Default(..))
2  import Ledger.TimeSlot
3  import Prelude          (IO, Semigroup(..), Show(..), String)
4
5  {-# INLINABLE mkPolicy #-}
6  -- This policy should only allow minting (or burning) of tokens if the owner
   -- of the specified PaymentPubKeyHash has signed the transaction and if the
7  -- specified deadline has not passed.
8  mkPolicy :: PaymentPubKeyHash -> POSIXTime -> () -> ScriptContext -> Bool

```

```

9  mkPolicy pkh deadline () ctx =
10    traceIfFalse "signature missing" (txSignedBy info $ unPaymentPubKeyHash pkh) &&
11    traceIfFalse "deadline missed" (to deadline `contains` txInfoValidRange info)
12  where
13    info = scriptContextTxInfo ctx
14
15 policy :: PaymentPubKeyHash -> POSIXTime -> Scripts.MintingPolicy
16 policy pkh deadline = mkMintingPolicyScript $
17   $$($PlutusTx.compile [|| `pkh` deadline' -> Scripts.wrapMintingPolicy $ mkPolicy pkh` deadline' ||])
18   `PlutusTx.applyCode`
19   PlutusTx.liftCode pkh
20   `PlutusTx.applyCode`
21   PlutusTx.liftCode deadline
22
23 curSymbol :: PaymentPubKeyHash -> POSIXTime -> CurrencySymbol
24 curSymbol pkh deadline = scriptCurrencySymbol $ policy pkh deadline
25
26 data MintParams = MintParams
27   { mpTokenName :: !TokenName
28   , mpDeadline :: !POSIXTime
29   , mpAmount :: !Integer
30   } deriving (Generic, ToJSON, FromJSON, ToSchema)
31
32 type SignedSchema = Endpoint "mint" MintParams
33
34 mint :: MintParams -> Contract w SignedSchema Text ()
35 mint mp = do
36   pkh <- Contract.ownPaymentPubKeyHash
37   now <- Contract.currentTime
38   let deadline = mpDeadline mp
39   if now > deadline
40     then Contract.logError @String "deadline passed"
41   else do
42     let val      = Value.singleton (curSymbol pkh deadline)
43                  (mpTokenName mp) (mpAmount mp)
44     lookups = Constraints.mintingPolicy $ policy pkh deadline
45     tx      = Constraints.mustMintValue val <>
46               Constraints.mustValidateIn (to $ now + 60000)
47     ledgerTx <- submitTxConstraintsWith @Void lookups tx
48     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
49     Contract.logInfo @String $ printf "Forged %s" (show val)

```

```

49 endpoints :: Contract () SignedSchema Text ()
50 endpoints = mint' >> endpoints
51 where
52     mint' = awaitPromise $ endpoint @"mint" mint
53
54 mkSchemaDefinitions ''SignedSchema
55
56 mkKnownCurrencies []
57
58 test :: IO ()
59 test = runEmulatorTraceIO $ do
60     let tn      = "ABC"
61         deadline = slotToBeginPOSIXTime def 100
62     h <- activateContractWallet (knownWallet 1) endpoints
63     callEndpoint @"mint" h $ MintParams
64         { mpTokenName = tn
65         , mpDeadline = deadline
66         , mpAmount   = 555
67         }
68     void $ Emulator.waitNSlots 110
69     callEndpoint @"mint" h $ MintParams
70         { mpTokenName = tn
71         , mpDeadline = deadline
72         , mpAmount   = 555
73         }
74     void $ Emulator.waitNSlots 1

```

Compared to the *Signed.hs* code example we make two new imports and add the Semigroup type class to the Prelude import. Then we define our policy function where we take 2 input parameters: the payment public key hash and the deadline (8). Next we check that the correct signature is present and that the validity interval is before now (9-13). After that we compile the policy function and define the currency symbol (15-24). For the off-chain code we first define our minting parameters that include the token name, deadline and amount (26-30). For the schema we have only the mint endpoint. In the mint contract we get our own public key hash and the current POSIX time. If the current time is before the deadline we proceed with actions if not we just log a message. First we define the value where the currency symbol takes in our two parameters (42). Same is true for policy parameter in the lookups variable (43). When we construct the transaction we say which value should be minted and define the validity interval for the transaction (44). Then we submit the transaction, wait for confirmation and log a message. We define our endpoints contract as usual and create a schema definition and some currencies for the playground. In the emulator trace we define our token name and

deadline. Then we activate the contract for wallet one and call our first endpoint. After that we wait for 110 slots and then call again the endpoint. This second call should fail since the deadline will already pass and we should not be able to mint any tokens. An interesting observation we can mention is that the validitiy interval in the mint contract is hardcoded to 60 seconds. If we change the deadline in the emulator trace to 61 slots the minting fails, but for 62 slots it succeeds. For our second homework we want to create a similar minting policy as in the *NFT.hs* example where we fix the token name to an empty byte string. Let's look at the code example from *Solution2.hs*.

```

1 {-# INLINABLE tn #-}
2 tn :: TokenName
3 tn = TokenName emptyByteString
4
5 {-# INLINABLE mkPolicy #-}
6 -- Minting policy for an NFT, where the minting transaction must consume the
7 -- given UTxO as input and where the TokenName will be the empty ByteString.
8 mkPolicy :: TxOutRef -> () -> ScriptContext -> Bool
9 mkPolicy oref () ctx = traceIfFalse "UTx0 notconsumed" hasUTx0 &&
10                      traceIfFalse "wrong amount minted" checkMintedAmount
11 where
12   info :: TxInfo
13   info = scriptContextTxInfo ctx
14
15   hasUTx0 :: Bool
16   hasUTx0 = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs info
17
18   checkMintedAmount :: Bool
19   checkMintedAmount = case flattenValue (txInfoMint info) of
20     [(cs, tn', amt)] -> cs == ownCurrencySymbol ctx &&
21                           tn' == tn && amt == 1
22     _ -> False
23
24 policy :: TxOutRef -> Scripts.MintingPolicy
25 policy oref = mkMintingPolicyScript $
26   $$($PlutusTx.compile [|| Scripts.wrapMintingPolicy . mkPolicy ||])
27   `PlutusTx.applyCode`
28   PlutusTx.liftCode oref
29
30 curSymbol :: TxOutRef -> CurrencySymbol
31 curSymbol = scriptCurrencySymbol . policy
32
33 type NFTSchema = Endpoint "mint" Address
34 mint :: Address -> Contract w NFTSchema Text ()

```

```

35 mint addr = do
36     utxos <- utxosAt addr
37     case Map.keys utxos of
38         []      -> Contract.logError @String "no utxo found"
39         oref : _ -> do
40             let val      = Value.singleton (curSymbol oref) tn 1
41             lookups = Constraints.mintingPolicy (policy oref) <>
42                             Constraints.unspentOutputs utxos
43             tx      = Constraints.mustMintValue val <>
44                             Constraints.mustSpendPubKeyOutput oref
45             ledgerTx <- submitTxConstraintsWith @Void lookups tx
46             void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
47             Contract.logInfo @String $ printf "forged %s" (show val)
48
49 endpoints :: Contract () NFTSchema Text ()
50 endpoints = mint' >> endpoints
51 where
52     mint' = awaitPromise $ endpoint @"mint" mint
53
54 test :: IO ()
55 test = runEmulatorTraceIO $ do
56     let w1 = knownWallet 1
57         w2 = knownWallet 2
58     h1 <- activateContractWallet w1 endpoints
59     h2 <- activateContractWallet w2 endpoints
60     callEndpoint @"mint" h1 $ mockWalletAddress w1
61     callEndpoint @"mint" h2 $ mockWalletAddress w2
62     void $ Emulator.waitNSlots 1

```

First we define our token name which is an empty byte string. Then we define our policy function that takes now as input only the transaction output reference (8). The first helper function that checks the transaction output reference is the same as in the NFT example. The second helper function however changes. There we can now compare the currency symbol, the token name and the amount (18-20). The token name is now a fixed variable rather than an input parameter. We compile our policy where we have now only one parameter and define the currency symbol (23-30). Next comes the off-chain code. When defining the schema we take as input only the address which is also the input for the mint contract. The mint contract compared to the NFT example is very similar. What changes is the definition of the value and lookups variables where we provide for the currency symbol and the policy only one parameter (40-41). In the emulator trace we now do not need to define a token name but only pass in the address as input when calling the endpoint.

6 Deployment

In this lecture we will get to know the PAB - Plutus application backend. The PAB provides the components and an environment to help developers create and test DApps (decentralized applications), before deploying them to a live production environment. The PAB is a single Haskell library that makes it easier to write the off-chain infrastructure and the on-chain scripts [5]. We will see how to use the PAB to interact with contracts on the Cardano testnet and show an example how to mint native tokens on the testnet using the Cardano CLI and PAB.

6.1 The minting policy

In this chapter we will present the on-chain code found in the file week06/Token/OnChain.hs. The code is similar to the NFT example. Let's look at the code.

```
1 {-# LANGUAGE DataKinds      #-}
2 {-# LANGUAGE DeriveAnyClass #-}
3 {-# LANGUAGE DeriveGeneric   #-}
4 {-# LANGUAGE FlexibleContexts #-}
5 {-# LANGUAGE NoImplicitPrelude #-}
6 {-# LANGUAGE NumericUnderscores #-}
7 {-# LANGUAGE OverloadedStrings #-}
8 {-# LANGUAGE ScopedTypeVariables #-}
9 {-# LANGUAGE TemplateHaskell #-}
10 {-# LANGUAGE TypeApplications #-}
11 {-# LANGUAGE TypeFamilies   #-}
12 {-# LANGUAGE TypeOperators   #-}

13
14 module Week06.Token.OnChain
15     ( tokenPolicy
16     , tokenCurSymbol
17     ) where
18
19 import qualified PlutusTx
20 import           PlutusTx.Prelude          hiding (Semigroup(..), unless)
21 import           Ledger                   hiding (mint, singleton)
22 import qualified Ledger.Typed.Scripts    as Scripts
23 import           Ledger.Value            as Value
24
25 {-# INLINABLE mkTokenPolicy #-}
26 mkTokenPolicy :: TxOutRef -> TokenName -> Integer ->
27                  () -> ScriptContext -> Bool
28 mkTokenPolicy oref tn amt () ctx = traceIfFalse "UTxO not consumed" hasUTxO
29                               && traceIfFalse "wrong amount minted" checkMintedAmount
```

```

29   where
30     info :: TxInfo
31     info = scriptContextTxInfo ctx
32
33     hasUTxO :: Bool
34     hasUTxO = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs info
35
36     checkMintedAmount :: Bool
37     checkMintedAmount = case flattenValue (txInfoMint info) of
38       [(_, tn', amt')] -> tn' == tn && amt' == amt
39       _ -> False
40
41   tokenPolicy :: TxOutRef -> TokenName -> Integer -> Scripts.MintingPolicy
42   tokenPolicy oref tn amt = mkMintingPolicyScript $
43     $$($PlutusTx.compile [|| \oref' tn' amt' -> Scripts.wrapMintingPolicy $ mkTokenPolicy oref' tn' amt' ||])
44     `PlutusTx.applyCode`
45     PlutusTx.liftCode oref
46     `PlutusTx.applyCode`
47     PlutusTx.liftCode tn
48     `PlutusTx.applyCode`
49     PlutusTx.liftCode amt
50
51   tokenCurSymbol :: TxOutRef -> TokenName -> Integer -> CurrencySymbol
52   tokenCurSymbol oref tn = scriptCurrencySymbol . tokenPolicy oref tn

```

First we define our token policy that takes in three additional parameters: the transaction output reference, token name and the amount of tokens we want to create. In the NFT example the amount was not necessary since we wanted to create only one token. In the body of the function we check that the referenced UTXO is consumed and that the token name and quantity match (27-39). Then we compile our token policy function and create the currency symbol. All together this code represents the on-chain part.

6.2 Minting with the CLI

We will use now the command line interface to mint tokens. If we recall chapter 3 where we already used the CLI we remember that we needed the serialized script that will now be computed from our minting policy instead of a validator. If we check in the documentation for the type *MintingPolicy* we see it's just a wrapper around the *Script* type (Figure 15). Also the type *Validator* is a wrapper around the script type. We define various helper functions in the *Utils.hs* file. There we have the *writeMintingPolicy* function that will create our serialized script.

```
newtype MintingPolicy
```

MintingPolicy is a wrapper around Scripts which are used as validators for minting constraints.

Constructors

```
MintingPolicy
```

```
getMintingPolicy :: Script
```

Figure 15 - Minting policy

Let's have look now at the transaction output reference type (Figure 16). The picture shows us that a UTXO is specified by a reference to the transaction that created it which is represented by the TxId type and an index which gets assigned to each of the UTXOs a transaction has produced. The TxId type is just a newtype wrapper around the BuiltinByteString type.

```
data TxOutRef
```

Source

A reference to a transaction output. This is a pair of a transaction reference, and an index indicating which of the outputs of that transaction we are referring to.

Constructors

```
TxOutRef
```

```
txOutRefId :: TxId
```

```
txOutRefIdx :: Integer
```

Index into the referenced transaction's outputs

Figure 16 - Transaction output reference

The TxId type is also an instance of the IsString type class that implements the function *fromString* which you can use to convert between string and the type in question. Now we focus on the CLI. You can also use the week06/testnet folder where the configuration files are already included. The bash scripts we will use in this example you can find directly in the week06 folder. First start the cardano node and let it sync to the testnet.

```
$ ./start-testnet-node.sh
```

Next we go into the testnet/ folder and create a key and a payment addresses. The code is the same as in chapter 3.

```
$ cardano-cli address key-gen --verification-key-file 01.vkey --signing-key-file 01.skey
$ cardano-cli address build --payment-verification-key-file 01.vkey --testnet-magic
1097911063 -out-file 01.addr
```

You will again need the Faucet to sent some test ADA to your address. The script *env.sh* sets all important environment variables that we will use. Once you source it you can use other bash scripts. The address and wallet ID parameters will be used later in the PAB chapter.

```
$ . env.sh
```

With the *query-key1.sh* script we can get the list of UTXOs sitting at our address together with the amounts of test ADA.

```
$ ./query-key1.sh
```

TxHash	TxIx	Amount
80e62eea8e5079598e4f341c704ef8957fb4d33f9818939aeda78a959a42dbe1	0	989834279
lovelace + TxOutDatumNone		

As we said a UTXO is defined with the transaction hash followed by the # symbol and then the transaction index. We can use this string and convert it to a transaction output reference with the helper function *unsafeReadTxOutRef* that is in the Utils module. And with the code from the file *app/token-policy.hs* we can create the minting policy script in serialized form. This module is added to the cabal file so we can run in with the following command:

```
$ cabal exec token-policy -- policy.plutus <TxHash#TxIx> 123456 PPP
```

You will need to provide the transaction hash and index that you got from the query command in the filed <TxHash#TxIx>. We create an amount of 123456 tokens with name PPP. Now we can actually mint the token and we will use the *mint-token-cli.sh* script for that. It takes in 5 parameters: the transaction output reference, amount, token name, address file and the signing key file. Next it gets the protocol parameters for which we use the protocol-parameters command from the Cardano CLI. After that it creates the token policy file. Then we compute the policy id called pid and the token name in hexadecimal format which is a requirement from the Cardano CLI. The function that can convert a token name to the hexadecimal format is called *unsafeTokenNameToHex* and can be found in the Utils module. In the bash script we execute the *token-name.hs* script to get the token name in hexadecimal format. The parameter v represent the value we want to mint and the CLI uses the convention that we first specify the amount and then the pid and hex token name separated by a dot.

Next comes the actual transaction command. With the transaction build command we construct our transaction where we input various data as in the example we showed in chapter 3. The difference is that we now specify 3 mint parameters: the value, the script file and the redeemer file. For the tx-out parameter we specify some ADA which should be greater than the minimal amount of ADA required for a UTXO. What's important to notice here is that the

minimal amount is not a constant but is rather calculated from size of the output in bytes. In the end we specify where to write the unsigned balanced transaction to. Then we use the CLI commands to sign and submit the transaction. We mint the token with the following command.

```
$ ./mint-token-cli.sh <TxHash#TxIx> 123456 PPP testnet/01.addr testnet/01.skey
```

After we run the command we can wait for a few seconds and then run again the query-key1.sh script to check if the token got minted. We can also go to explorer.cardano-testnet.iohkdev.io which is a cardano blockchain explorer for the cardano testnet. There we have to input our transaction ID and we will see transaction information such as how many ADA was sent from which to which address and what tokens were minted. We could do the same procedure on the main net where we would replace the Magic ID with --mainnet.

6.3 Deployment Scenarios

Let's look at the deployment models for the Plutus application backend - PAB. There are two deployment models envisioned for the PAB: Hosted and in-browser. We will use the hosted model. It works by running a cardano node on a server, a cardano wallet backend which for example is used by the Daedalus wallet, a chain index and the PAB itself.

The chain index handles saving of the blockchain information in an SQL database. It can be used for instance to lookup a datum belonging to a datum hash, which the cardano node can't do. Our PAB should have access to our wallet so an external user of our dApp would then interact with our PAB through a user interface that has endpoints available (**Error! Reference source not found.**).

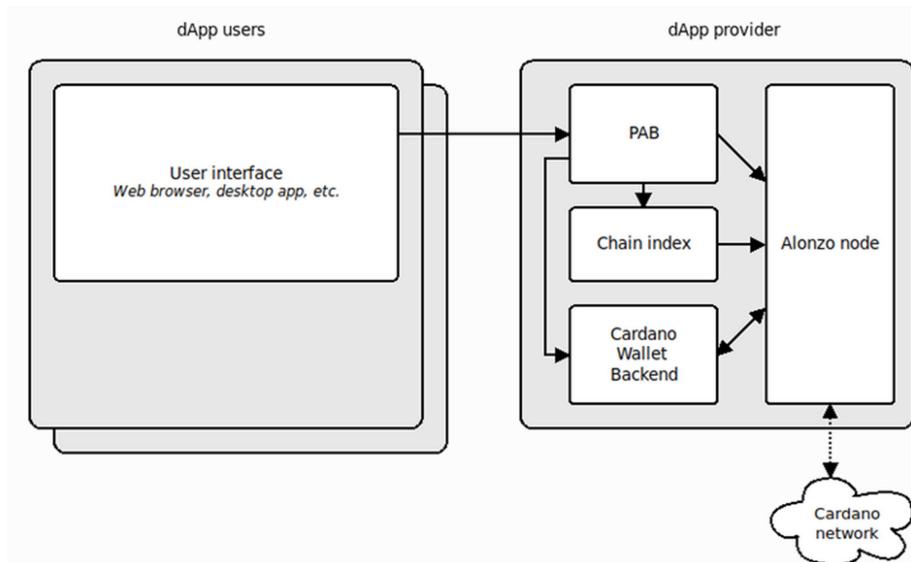


Figure 17 - dApp schema

6.4 The Contracts

We will now define some off-chain code written in a contract monad that the PAB can execute. First we look at the *getCredentials* helper function defined in the Utils module.

```
data Address  
  
Address with two kinds of credentials, normal and staking.  
  
Constructors  
  
Address  
  addressCredential :: Credential  
  addressStakingCredential :: Maybe StakingCredential
```

Figure 18 - Address type

It takes in a Plutus Address type (Figure 18). We can look at *Credential* type in Figure 19. It holds a public key hash and a validator hash.

```
data Credential # Source  
  
Credential required to unlock a transaction output  
  
Constructors  
  
PubKeyCredential PubKeyHash The transaction that spends this output must be signed by the private key  
ScriptCredential ValidatorHash The transaction that spends this output must include the validator script and be accepted by the validator.
```

Figure 19 - Credential type

Then we can look up the *StakingCredential* type (**Error! Reference source not found.**). It holds again a *Credential* type and a staking pointer that is represented by three integers.

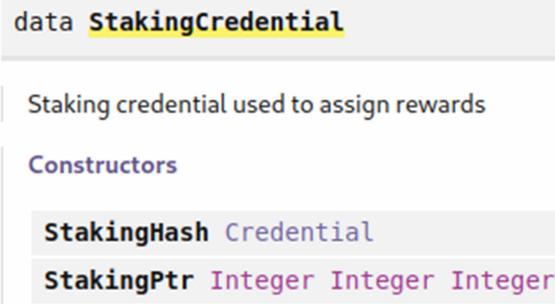


Figure 20 - Staking credential

So the `getCredentials` function returns a `Nothing` if it the input is a script address and a `Just` value if it is a public key address that contains a payment public key hash and maybe a stake public key hash. Let's look now at the off-chain code contained in the `OffChain.hs` file.

```

1  {-# LANGUAGE DataKinds          #-}
2  {-# LANGUAGE DeriveAnyClass     #-}
3  {-# LANGUAGE DeriveGeneric       #-}
4  {-# LANGUAGE FlexibleContexts   #-}
5  {-# LANGUAGE NoImplicitPrelude  #-}
6  {-# LANGUAGE OverloadedStrings   #-}
7  {-# LANGUAGE ScopedTypeVariables #-}
8  {-# LANGUAGE TypeApplications    #-}
9  {-# LANGUAGE TypeFamilies        #-}

10
11 module Week06.Token.OffChain
12   ( TokenParams(..)
13   , adjustAndSubmit, adjustAndSubmitWith
14   , mintToken
15   ) where
16
17 import           Control.Monad           hiding (fmap)
18 import           Data.Aeson             (FromJSON, ToJSON)
19 import qualified Data.Map               as Map
20 import           Data.Maybe             (fromJust)
21 import           Data.OpenApi.Schema   (ToSchema)
22 import           Data.Text              (Text, pack)
23 import           Data.Void              (Void)
24 import           GHC.Generics          (Generic)
25 import           Plutus.Contract       as Contract
26 import           Plutus.Contract.Wallet (getUnspentOutput)
27 import qualified PlutusTx
28 import           PlutusTx.Prelude     hiding (Semigroup(..), unless)
29 import           Ledger                  hiding (mint, singleton)

```

```

30 import           Ledger.Constraints      as Constraints
31 import qualified Ledger.Typed.Scripts    as Scripts
32 import           Ledger.Value           as Value
33 import           Prelude
34 import qualified Prelude
35 import           Text.Printf             (printf)
36
37 import           Week06.Token.OnChain
38 import           Week06.Utils          (getCredentials)
39
40 data TokenParams = TokenParams
41   { tpToken   :: !TokenName
42   , tpAmount  :: !Integer
43   , tpAddress :: !Address
44   } deriving (Prelude.Eq, Prelude.Ord, Generic, FromJSON, ToJSON,
45               ToSchema, Show)
46
47 adjustAndSubmitWith :: ( PlutusTx.FromData (Scripts.DatumType a)
48                         , PlutusTx.ToData (Scripts.RedemerType a)
49                         , PlutusTx.ToData (Scripts.DatumType a)
50                         , AsContractError e
51                         )
52           => ScriptLookups a
53           -> TxConstraints (Scripts.RedemerType a)
54           -> (Scripts.DatumType a)
55           -> Contract w s e CardanoTx
56
57 adjustAndSubmitWith lookups constraints = do
58   unbalanced <- adjustUnbalancedTx <$> mkTxConstraints lookups constraints
59   Contract.logDebug @String $ printf "unbalanced: %s" $ show unbalanced
60   unsigned <- balanceTx unbalanced
61   Contract.logDebug @String $ printf "balanced: %s" $ show unsigned
62   signed <- submitBalancedTx unsigned
63   Contract.logDebug @String $ printf "signed: %s" $ show signed
64   return signed
65
66
67 adjustAndSubmit :: ( PlutusTx.FromData (Scripts.DatumType a)
68                     , PlutusTx.ToData (Scripts.RedemerType a)
69                     , PlutusTx.ToData (Scripts.DatumType a)
70                     , AsContractError e
71                     )
72           => Scripts.TypedValidator a
73           -> TxConstraints (Scripts.RedemerType a)
74           -> (Scripts.DatumType a)
75           -> Contract w s e CardanoTx

```

```

71 adjustAndSubmit inst = adjustAndSubmitWith $
    Constraints.typedValidatorLookups inst
72
73 mintToken :: TokenParams -> Contract w s Text CurrencySymbol
74 mintToken tp = do
75     Contract.logDebug @String $ printf "started minting: %s" $ show tp
76     let addr = tpAddress tp
77     case getCredentials addr of
78         Nothing      -> Contract.throwError $ pack $ printf
79                         "expected pubkey address, but got %s" $ show addr
80         Just (x, my) -> do
81             oref <- getUnspentOutput
82             o     <- fromJust <$> Contract.txOutFromRef oref
83             Contract.logDebug @String $ printf "picked UTXO at %s with value
84                         %s" (show oref) (show $ _ciTxOutValue o)
85
86             let tn          = tpToken tp
87                 amt        = tpAmount tp
88                 cs          = tokenCurSymbol oref tn amt
89                 val        = Value.singleton cs tn amt
90                 c           = case my of
91                     Nothing   -> Constraints.mustPayToPubKey x val
92                     Just y    -> Constraints.mustPayToPubKeyAddress x y val
93             lookups      = Constraints.mintingPolicy
94                           (tokenPolicy oref tn amt) <>
95                           Constraints.unspentOutputs
96                           (Map.singleton oref o)
97             constraints = Constraints.mustMintValue val           <>
98                           Constraints.mustSpendPubKeyOutput oref <> c

96 void $ adjustAndSubmitWith @Void lookups constraints
97 Contract.logInfo @String $ printf "minted %s" (show val)
98 return cs

```

First we define our token parameter type that holds the token name, the amount and the address. This is not the change address but the address where the tokens should be sent after the minting. The change address will then be picked by the wallet.

If we look at the *mintToken* function we see it takes in the token parameters and returns a contract parameterized with the currency symbol. We won't really need this since this was used in an oracle example from a previous lecture of the pluto pioneer program. First we log that we are starting the mint (75) and then we define the address. Then we use the *getCredentials* helper function (77). If it returns Nothings which means we got a script address we raise an

error. If it's not nothing we extract the payment public key hash and optionally the staking key hash. Next we use the helper function `getUnspentOutput` that looks for an unspent UTXO and returns its output reference (80). With the function `txOutFromRef` we get the actual output for which we can be sure at this moment that it exists (81). Then we log some information about the output and its value. After that come several definitions. First we define the token name and the amount we want to mint. Then we compute the currency symbol (86) and the value (87) we want to mint. Depending on whether we do have staking involved we compute two different constraints (88-90). For the lookups we specify the minting policy which we can get because we imported the module from the `OnChain.hs` file and the unspent outputs. Because there is only one output we can use the `Map.singleton` function. The last definitions are our constraints where we constrain the minting value, to which address it should be paid and of course which UTXO we want to spend. This makes sure there can be only one minting transaction for the given currency symbol. Now we want to submit the transaction. For that we use the helper function `adjustAndSubmitWith`. The reason for this is that we want some more extensive logging and that we want to take care of the minimal ADA requirement. In the end we log some minimal information and return the currency symbol.

There is also an emulator trace where this contract can be tested. It can be found in the `Trace.hs` file. In there we activate the contract for wallet 1 with some input parameters. And if we run it we get the newly minted token with the specified amount. The `mintToken` contract does not return any information to the user and it does not have any endpoints. Let's look now at a second contract from the `week06/Monitor.hs` example. It's a long running contract that monitors an address that is in contrast to the previous example where the mint immediately returns the result. Let's have a look at the code.

```

1  {-# LANGUAGE DataKinds          #-}
2  {-# LANGUAGE FlexibleContexts  #-}
3  {-# LANGUAGE OverloadedStrings #-}
4  {-# LANGUAGE ScopedTypeVariables #-}
5  {-# LANGUAGE TypeApplications   #-}
6  {-# LANGUAGE TypeFamilies      #-}

7
8  module Week06.Monitor
9    ( monitor
10  ) where
11
12 import           Data.Functor      (void)
13 import qualified Data.Map          as Map
14 import           Data.Monoid       (Last(..))
15 import           Data.Text         (Text)
16 import           Plutus.Contract as Contract

```

```

17 import           Ledger
18 import           Text.Printf      (printf)
19
20 monitor :: Address -> Contract (Last Value) Empty Text a
21 monitor addr = do
22     Contract.logInfo @String $ printf "started monitoring address %s" $
23             show addr
24     go
25     where
26         go = do
27             utxos <- utxosAt addr
28             let v = Map.foldl' (\w o -> w <> _ciTxOutValue o) mempty utxos
29             tell $ Last $ Just v
30             void $ waitNSlots 1
31         go

```

It takes an address as its only argument and then it queries the blockchain and reports the value sitting at that address. We report this value with the use of the w parameter. For it we choose the *Last Value* type that is an instance of monoid type class.

```

Prelude> import Data.Monoid
Prelude Data.Monoid> :i Last
type Last :: * -> *
newtype Last a = Last {getLast :: Maybe a}

```

We see that Last is just a newtype wrapper around a Maybe. It changes the monoid instance of Maybe a and it always keeps the last Just.

```

Prelude Data.Monoid> mempty :: Last Char
Last {getLast = Nothing}
Prelude Data.Monoid> Last (Just 'a') <> Last (Just 'b')
Last {getLast = Just 'b'}
Prelude Data.Monoid> Last (Just 'a') <> Last (Just 'b') <> Last Nothing
Last {getLast = Just 'b'}

```

What we want to do in this code example is to write the last Value found at the given address in regular intervals. The go function calls itself recursively so it goes on forever (25-30). In each iteration we first find all UTXOs at the given address. We want to find the value contained at this address so we fold over the UTXOs to add the values from them to a single parameter. We use the accumulator w in the lambda function and add to it the value from a single UTXO (27). Then we tell the value and wait for one slot before repeating the process. So every slot this will update the observable state contained in the w parameter value.

6.5 Minting with the PAB

We will use now the code from the `week06/PAB.hs` file to hook up our minting and monitor contracts to the PAB. Let's look at the code.

```
1 {-# LANGUAGE DeriveAnyClass      #-}
2 {-# LANGUAGE DeriveGeneric       #-}
3 {-# LANGUAGE OverloadedStrings   #-}
4 {-# LANGUAGE TypeApplications    #-}
5
6 module Week06.PAB
7   ( Address
8     , TokenContracts(..)
9   ) where
10
11 import           Data.Aeson          (FromJSON, ToJSON)
12 import           Data.OpenApi.Schema (ToSchema)
13 import           GHC.Generics        (Generic)
14 import           Ledger              (Address)
15 import           Plutus.PAB.Effects.Contract.Builtin (Empty, HasDefinitions
16                                                       (..), SomeBuiltin (..), endpointsToSchemas)
17 import           Prettyprinter        (Pretty (..), viaShow)
18 import           Wallet.Emulator.Wallet (knownWallet,
19                                         mockWalletAddress)
20
21
22 data TokenContracts = Mint Token.TokenParams | Monitor Address
23   deriving (Eq, Ord, Show, Generic, FromJSON, ToJSON, ToSchema)
24
25 instance Pretty TokenContracts where
26   pretty = viaShow
27
28 instance HasDefinitions TokenContracts where
29
30   getDefinitions      = [Mint exampleTP, Monitor exampleAddr]
31
32   getContract (Mint tp)      = SomeBuiltin $ Token.mintToken @() @Empty tp
33   getContract (Monitor addr) = SomeBuiltin $ Monitor.monitor addr
34
35   getSchema = const $ endpointsToSchemas @Empty
36
37 exampleAddr :: Address
38 exampleAddr = mockWalletAddress $ knownWallet 1
```

```

39
40 exampleTP :: Token.TokenParams
41 exampleTP = Token.TokenParams
42     { Token.tpAddress = exampleAddr
43     , Token.tpAmount = 123456
44     , Token.tpToken = "PPP"
45 }
```

First we import our monitor and minting examples (19-20). In the beginning we define a data type called *TokenContracts* that represents everything a PAB can do which means it will define which contracts the PAB will expose (22). From it we derive a bunch of type classes (23). We need to define the *Pretty* instance for our token contracts and use the *viaShow* function. Then we need to write an instance for the class *HasDefinitions* that has 3 methods. One is called *getDefinitions* that is a list of values of the token contract type. For the list we use some sample values that we define at the end of our code (37-45). The *getContract* method tells us which contract should run given a value of the token contracts type (32-33). We have to wrap our contract in the *SomeBuiltIn* constructor. Finally we provide the empty schema for both contracts where we use the *Empty* type (35). Now we can write an application that starts the PAB with our contracts. The code can be found in *app/token-pab.hs*.

```

1 {-# LANGUAGE DataKinds      #-}
2 {-# LANGUAGE DerivingStrategies #-}
3 {-# LANGUAGE FlexibleContexts #-}
4 {-# LANGUAGE OverloadedStrings #-}
5 {-# LANGUAGE RankNTypes      #-}
6 {-# LANGUAGE TypeApplications #-}
7 {-# LANGUAGE TypeFamilies    #-}
8
9 module Main
10   ( main
11   ) where
12
13 import qualified Plutus.PAB.Effects.Contract.Builtin as Builtin
14 import           Plutus.PAB.Run                      (runWith)
15
16 import           Week06.PAB                         (TokenContracts)
17
18 main :: IO ()
19 main = do
20   runWith (Builtin.handleBuiltin @TokenContracts)
```

It's quite simple. All it does it calls the *runWith* function that takes in the token contracts parameter. This code represents the executable that will start the PAB and it will be specialized to the provided contracts. But for this to work we have to start some other applications as well. That are the applications shown in the dApp schema (**Error! Reference source not found.**). We already know how to set up a cardano node. What we also need is a wallet and the chain index. In order to set that up we can follow the instructions for the PAB in the `plutus-apps` git repository:

<https://github.com/input-output-hk/plutus-apps/blob/main/plutus-pab/test-node/README.md>

We will look now at these instructions step by step. Our node should already be running. The next step is the wallet. When we are in the nix shell `cardano-wallet` is available as a command. We can use the bash script `start-testnet-wallet.sh` to start the wallet backend from the nix shell. The next step is to create a wallet. For that we can use the `create-wallet.sh` bash script. The script takes following input parameters: the name of the wallet, the passphrase and the file name where to write it. A file gets generated with the recovery phrase. We can now use the recovery phrase to import our wallet into the Daedalus wallet. After that we need to fund that wallet by either sending funds from an existing wallet to the new one or we use the testnet Faucet. Now we also have to inform the wallet backend about our new wallet. We can do this with the bash script `load-wallet.sh`. The request body in the curl command is the json file we just created. When we run this script we get a json object as response that contains the wallet ID which we will later need in the PAB, so we save it to the `env.sh` file. Next we can start the chain index with the script `start-testnet-chain-index.sh`. The chain index takes a long time to synchronize; it could be longer than the node itself. We can start the PAB with the bash script `start-testnet-pab.sh`. There we have to use the passphrase that we choose when we created the wallet. Before we do this we need to do some more configurations. In the `pab-config.yml` configuration file we provide in the command `developmentOptions` a parameter that allows us to specify from which block we want to start synchronizing the PAB, which can speed up the synchronization process. We can get the block id from the node where it's logged after the "new tip:" keyword every time a block gets created and the slot after the "at slot" keyword. So we can take just the last entry. Because we make a fresh start there is no database yet and we need to migrate it before we start the PAB. We do this with the bash script `migrate-pab.sh`. We run this script and then we can run the `start-testnet-pab.sh` script. From the output we will see that the PAB is available at port 9080 on the localhost. To get a nice interface we can go to `localhost:9080/swagger/swagger-ui`. There we have various HTTP endpoints available that we can execute and look at the results directly in this UI. With the `/api/contract/activate` endpoint we can start the contract on the PAB to do the minting or the monitoring. The cool thing about the UI is when we run an endpoint we also get the associated curl command displayed which

we can copy into a bash script and run it with our parameters. You can find the minting command in the bash script *mint-token-curl.sh*. For this script we need the address of our wallet, which we can get if we use Yoroi or Daedalus wallet where we go to the receive tab and look at our receiving address. Or we can use the wallet backend directly which is demonstrated in the script *get-address.sh*. The script generates many wallet addresses from which we can pick one and write it to our *env.hs* file. But we need to provide our addresses in Plutus format which separates the payment public key hash and the staking public key hash. We get the pkh and skh with use of the helper functions *payment-key-hash* and *stake-key-hash* specified at the beginning of the bash file. If we run now the *mint-token-curl.sh* script we can look at our wallet under Assets and our PPP token should appear. There is also the option to mint the token directly from a Haskell script. We can look at the *app/mint-token.hs* file.

```

1 {-# LANGUAGE OverloadedStrings #-}
2
3 module Main
4   ( main
5   ) where
6
7 import Control.Exception          (throwIO)
8 import Data.String                (IsString(..))
9 import Network.HTTP.Req
10 import System.Environment         (getArgs)
11 import Text.Printf               (printf)
12 import Wallet.Emulator.Wallet  (WalletId(..))
13 import Wallet.Types              (ContractInstanceId(..))
14 import Week06.PAB                (TokenContracts(..))
15 import Week06.Token.OffChain    (TokenParams(..))
16 import Week06.Utils              (contractActivationArgs, unsafeReadAddress,
17                                   unsafeReadWalletId)
18
19 main :: IO ()
20 main = do
21   [amt', tn', wid', addr'] <- getArgs
22   let wid = unsafeReadWalletId wid'
23   tp = TokenParams
24     { tpToken  = fromString tn'
25     , tpAmount = read amt'
26     , tpAddress = unsafeReadAddress addr'
27     }
28   printf "minting token for wallet id %s with parameters %s\n"
29         (show wid) $ show tp
30   cid <- mintToken wid tp
31   printf "minted tokens, contract instance id: %s\n" $ show cid

```

```

30
31 mintToken :: WalletId -> TokenParams -> IO ContractInstanceId
32 mintToken wid tp = do
33     v <- runReq defaultHttpConfig $ req
34     POST
35     (http "127.0.0.1" /: "api" /: "contract" /: "activate")
36     (ReqBodyJson $ contractActivationArgs wid $ Mint tp)
37     jsonResponse
38     (port 9080)
39     let c = responseStatusCode v
40     if c == 200
41         then return $ responseBody v
42         else throwError $ userError $ printf "ERROR: %d\n" c

```

We recall that there were quite few steps before we got to the JSON body of our curl command in the *mint-token-curl.sh* script. In Haskell that's much easier. Our program takes in 4 parameters: the amount, the token name, the wallet ID and the address (20). Then we pass them into appropriate types (21-26). From the Utils module we can use the function *unsafeReadWalletId* that converts a string into a real wallet ID and similar *unsafeReadAddress* converts a string into a real address. Then we call the *mintToken* function (28) which takes a wallet ID and token parameters as input and makes a HTTP request. If the response code is 200 we return the response body which will be of type *ContractInstanceId*. Else we log an error. With the bash script *mint-token-haskell.sh* we can now mint the token by using our Haskell file. As command line parameters it takes in the amount and token name. Now we can also look at the Haskell code for monitoring found in *app/monitor.hs*.

```

1 {-# LANGUAGE NumericUnderscores #-}
2 {-# LANGUAGE OverloadedStrings #-}
3
4 module Main
5     ( main
6     ) where
7
8 import Control.Concurrent          (threadDelay)
9 import Control.Exception           (throwIO)
10 import Control.Monad              (when)
11 import Data.Aeson                  (FromJSON(..))
12 import Data.Aeson.Types            (parseMaybe)
13 import Data.Maybe                  (fromMaybe)
14 import Data.Monoid                 (Last(..))
15 import Data.Text                   (pack)
16 import Network.HTTP.Req
17 import Plutus.PAB.Events.ContractInstanceState (PartiallyDecodedResponse)

```

```

18 import Plutus.PAB.Webserver.Types          (..))
19 import Plutus.V1.Ledger.Value             (ContractInstanceState
20 import System.Environment                (..))
21 import Text.Printf                      (Value, flattenValue)
22 import Wallet.Emulator.Wallet           (getArgs)
23 import Week06.PAB                      (printf)
24 import Wallet.Types                     (WalletId(..))
25 import Week06.Utils                     (Address, TokenContracts(..))
26                                         (ContractInstanceId(..))
27                                         (cidToString,
28                                         contractActivationArgs,
29                                         unsafeReadAddress,
30                                         unsafeReadWalletId)
31                                         (..))

32 main :: IO ()                          (..))
33 main = do
34     [wid', addr'] <- getArgs
35     let wid  = unsafeReadWalletId wid'
36     addr = unsafeReadAddress addr'
37     printf "monitoring address %s on wallet %s\n" (show addr) $ show wid
38     cid <- startMonitor wid addr
39     printf "started monitor-process with contract id %s\n\n" $
40             cidToString cid
41     go cid mempty
42 where
43     go :: ContractInstanceId -> Value -> IO a
44     go cid v = do
45         cic <- getMonitorState cid
46         let v' = fromMaybe v $ observedValue cic
47         when (v' /= v) $
48             printf "%s\n\n" $ show $ flattenValue v'
49         threadDelay 1_000_000
50         go cid v'
51
52 startMonitor :: WalletId -> Address -> IO ContractInstanceId
53 startMonitor wid addr = do
54     v <- runReq defaultHttpConfig $ req
55     POST
56     (http "127.0.0.1" /: "api" /: "contract" /: "activate")
57     (ReqBodyJson $ contractActivationArgs wid $ Monitor addr)
58     jsonResponse
59     (port 9080)
60     let c = responseStatusCode v
61     when (c /= 200) $
62         throwError $ userError $ printf "ERROR: %d\n" c

```

```

57     return $ responseBody v
58
59 getMonitorState :: ContractInstanceId ->
60             IO (ContractInstanceClientState TokenContracts)
60 getMonitorState cid = do
61     v <- runReq defaultHttpConfig $ req
62         GET
63         (http "127.0.0.1" /: "api" /: "contract" /: "instance" /:
64             pack (cidToString cid) /: "status")
64     NoReqBody
65     jsonResponse
66     (port 9080)
67     let c = responseStatusCode v
68     when (c /= 200) $
69         throwIO $ userError $ printf "ERROR: %d\n" c
70     return $ responseBody v
71
72 observedValue :: ContractInstanceClientState TokenContracts -> Maybe Value
73 observedValue cic = do
74     Last mv <- parseMaybe parseJSON $ observableState $ cicCurrentState cic
75     mv

```

The body of the main function is similar to how we did the minting. From it we call now two helper functions the *startMonitor* function that starts the monitoring and the *getMonitorState* function that gets the state of the monitor contract and some other information as the tokens we have in our wallet. We can use the *monitor.sh* script to start our code. There we input the environment variables wallet id and address.

Altogether we presented now 3 different ways how to mint our tokens: using the client, using the PAB and using Haskell functions directly. We have to note that these operations are quite resource hungry. If you have around 12GB of RAM it can still happen you will need to increase the size of your SWAP partition to 50GB to get the chain index to actually sync.

7 State Machines

In this chapter we will look at state machines (SM). They can be useful to write shorter and more concise code for both the on-chain and off-chain part. There is support for state machines in the Plutus libraries that is higher level and builds on top of the lower level mechanisms we have seen so far. But what we do have to mention is that at the current time of writing there is a certain overhead using state machines. If you write a contract with state machines it will require more resources to run as if you wrote the same contract without state machines. Because of that SM have not seen much use in practice yet. However the Plutus team is permanently working on improving performance and optimizing the compiler and interpreter so we can expect state machines to be really useful in the near future.

7.1 Commit schemes

Let's imagine a game played between Alice and Bob. It's similar to rock-paper-scissors but with only two options. Instead we have one gesture for 0 and one gesture for 1. If they both raise the same gesture then Alice wins and if they raise a different gesture then Bob wins. Let's say that Alice and Bob can't meet in person but rather play the game via email. Now we come to the problem how to make sure when Alice sends her choice to Bob that Bob does not read the email before making his choice to get an unfair advantage. There is a trick that is used in cryptographic protocols and it's about commit schemes. The idea is that Alice does not reveal her choice to Bob but rather commits to it so she later cannot change her mind. One way to make that work is using hash functions. Hashes are one way functions because given a hash it's impossible to construct the original text or byte string from which the hash was computed. The problem we have now is that Bob will soon figure out which hash belongs to which number since there are only two choices. So what Alice can do is that she first concatenates her choice number with some arbitrary byte string (that we call a nonce) before hashing it and then sends the hash to Bob. And when they check their choices Alice has to send to Bob her choice and the nonce in plain text so Bob can compute the hash on his own and make sure Alice did not cheat. We will try to implement such an example in Plutus together with the code we have seen so far. The idea is Alice and Bob put down a certain amount of money that then gets processed accordingly to their choices. We also implement the possibility when Alice opens the game by posting her hash and if Bob does not reply, she can retrieve her funds after a certain amount of time. Or if Bob replies and Alice sees she has lost and does not reply, Bob can retrieve his funds after a certain amount of time has passed.

7.2 Implementation without State Machines

We will first look at the example how to implement the game in Plutus from the previous chapter without using state machines. Let's first look at the *EvenOdd.hs* file.

```
1 {-# LANGUAGE DataKinds          #-}
2 {-# LANGUAGE DeriveAnyClass     #-}
3 {-# LANGUAGE DeriveGeneric      #-}
4 {-# LANGUAGE FlexibleContexts   #-}
5 {-# LANGUAGE MultiParamTypeClasses #-}
6 {-# LANGUAGE NoImplicitPrelude   #-}
7 {-# LANGUAGE OverloadedStrings   #-}
8 {-# LANGUAGE ScopedTypeVariables #-}
9 {-# LANGUAGE TemplateHaskell     #-}
10 {-# LANGUAGE TypeApplications    #-}
11 {-# LANGUAGE TypeFamilies       #-}
12 {-# LANGUAGE TypeOperators      #-}
13
14 module Week07.EvenOdd
15   ( Game(..)
16   , GameChoice(..)
17   , FirstParams(..)
18   , SecondParams(..)
19   , GameSchema
20   , endpoints
21   ) where
22
23 import           Control.Monad      hiding (fmap)
24 import           Data.Aeson        (FromJSON, ToJSON)
25 import qualified Data.Map          as Map
26 import           Data.Text          (Text)
27 import           GHC.Generics     (Generic)
28 import           Ledger            hiding (singleton)
29 import           Ledger.Constraints as Constraints
30 import qualified Ledger.Typed.Scripts as Scripts
31 import           Ledger.Ada         as Ada
32 import           Ledger.Value
33 import           Playground.Contract (ToSchema)
34 import           Plutus.Contract    as Contract
35 import qualified PlutusTx
36 import           PlutusTx.Prelude  hiding (Semigroup(..), unless)
37 import           Prelude            (Semigroup(..), Show(..), String)
38 import qualified Prelude
39
40 data Game = Game
```

```

41     { gFirst          :: !PaymentPubKeyHash
42     , gSecond         :: !PaymentPubKeyHash
43     , gStake          :: !Integer
44     , gPlayDeadline   :: !POSIXTime
45     , gRevealDeadline :: !POSIXTime
46     , gToken          :: !AssetClass
47   } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq, Prelude.Ord)
48
49 PlutusTx.makeLift ''Game
50
51 data GameChoice = Zero | One
52     deriving (Show, Generic, FromJSON, ToJSON, ToSchema, Prelude.Eq,
53               Prelude.Ord)
54
55 instance Eq GameChoice where
56     {-# INLINABLE (==) #-}
57     Zero == Zero = True
58     One == One = True
59     _ == _ = False
60
61 PlutusTx.unstableMakeIsData ''GameChoice
62
63 data GameDatum = GameDatum BuiltinByteString (Maybe GameChoice)
64     deriving Show
65
66 instance Eq GameDatum where
67     {-# INLINABLE (==) #-}
68     GameDatum bs mc == GameDatum bs' mc' = (bs == bs') && (mc == mc')
69
70 PlutusTx.unstableMakeIsData ''GameDatum
71
72 data GameRedeemer = Play GameChoice | Reveal BuiltinByteString |
73                         ClaimFirst | ClaimSecond
74     deriving Show
75
76 PlutusTx.unstableMakeIsData ''GameRedeemer
77
78 {-# INLINABLE lovelaces #-}
79 lovelaces :: Value -> Integer
80 lovelaces = Ada.getLovelace . Ada.fromValue
81
82 {-# INLINABLE gameDatum #-}
83 gameDatum :: Maybe Datum -> Maybe GameDatum
84 gameDatum md = do
85     Datum d <- md

```

```

84     PlutusTx.fromBuiltinData d
85
86 {-# INLINABLE mkGameValidator #-}
87 mkGameValidator :: Game -> BuiltinByteString -> BuiltinByteString ->
88             GameDatum -> GameRedeemer -> ScriptContext -> Bool
89 mkGameValidator game bsZero' bsOne' dat red ctx =
90     traceIfFalse "token missing from input" (assetClassValueOf
91                 (txOutValue ownInput) (gToken game) == 1) &&
92     case (dat, red) of
93         (GameDatum bs Nothing, Play c) ->
94             traceIfFalse "not signed by second player"
95                 (txSignedBy info (unPaymentPubKeyHash $ gSecond game)) &&
96             traceIfFalse "first player's stake missing" (lovelaces
97                 (txOutValue ownInput) == gStake game) &&
98             traceIfFalse "second player's stake missing" (lovelaces
99                 (txOutValue ownOutput) == (2 * gStake game)) &&
100            traceIfFalse "wrong output datum"
101            (outputDatum == GameDatum bs (Just c)) &&
102            traceIfFalse "missed deadline"
103            (to (gPlayDeadline game) `contains` txInfoValidRange info) &&
104            traceIfFalse "token missing from output"
105            (assetClassValueOf (txOutValue ownOutput) (gToken game) == 1)
106
106     (GameDatum bs (Just c), Reveal nonce) ->
107         traceIfFalse "not signed by first player"
108         (txSignedBy info (unPaymentPubKeyHash $ gFirst game)) &&
109         traceIfFalse "commit mismatch"
110         (checkNonce bs nonce c) &&
111         traceIfFalse "missed deadline"
112         (to (gRevealDeadline game) `contains` txInfoValidRange info) &&
113         traceIfFalse "wrong stake"
114         (lovelaces (txOutValue ownInput) == (2 * gStake game)) &&
115         traceIfFalse "NFT must go to first player" nftToFirst
116
116     (GameDatum _ Nothing, ClaimFirst) ->
117         traceIfFalse "not signed by first player"
118         (txSignedBy info (unPaymentPubKeyHash $ gFirst game)) &&
119         traceIfFalse "too early"
120         (from (1 + gPlayDeadline game) `contains` txInfoValidRange info)
121         &&
122         traceIfFalse "first player's stake missing"
123         (lovelaces (txOutValue ownInput) == gStake game) &&
124         traceIfFalse "NFT must go to first player" nftToFirst
125
125     (GameDatum _ (Just _), ClaimSecond) ->

```

```

113         traceIfFalse "not signed by second player"
114             (txSignedBy info (unPaymentPubKeyHash $ gSecond game)) &&
115             traceIfFalse "too early"
116                 (from (1 + gRevealDeadline game) `contains`
117                     txInfoValidRange info) &&
118                     traceIfFalse "wrong stake"
119                         (lovelaces (txOutValue ownInput) == (2 * gStake game)) &&
120                             traceIfFalse "NFT must go to first player"    nftToFirst
121
122             _ -> False
123
124     where
125         info :: TxInfo
126         info = scriptContextTxInfo ctx
127
128         ownInput :: TxOut
129         ownInput = case findOwnInput ctx of
130             Nothing -> traceError "game input missing"
131             Just i   -> txInInfoResolved i
132
133         ownOutput :: TxOut
134         ownOutput = case getContinuingOutputs ctx of
135             [o] -> o
136             _     -> traceError "expected exactly one game output"
137
138         outputDatum :: GameDatum
139         outputDatum = case gameDatum $ txOutDatumHash ownOutput >=
140                         flip findDatum info of
141                         Nothing -> traceError "game output datum not found"
142                         Just d   -> d
143
144         checkNonce :: BuiltinByteString -> BuiltinByteString ->
145             GameChoice -> Bool
146         checkNonce bs nonce cSecond = sha2_256
147             (nonce `appendByteString` cFirst) == bs
148
149     where
150         cFirst :: BuiltinByteString
151         cFirst = case cSecond of
152             Zero -> bsZero'
153             One  -> bsOne'
154
155         nftToFirst :: Bool
156         nftToFirst = assetClassValueOf (valuePaidTo info $ unPaymentPubKeyHash $ gFirst game) (gToken game) == 1
157
158     data Gaming

```

```

150 instance Scripts.ValidatorTypes Gaming where
151     type instance DatumType Gaming = GameDatum
152     type instance RedeemerType Gaming = GameRedeemer
153
154     bsZero, bsOne :: BuiltinByteString
155     bsZero = "0"
156     bsOne = "1"
157
158     typedGameValidator :: Game -> Scripts.TypedValidator Gaming
159     typedGameValidator game = Scripts.mkTypedValidator @Gaming
160         ($$(PlutusTx.compile [|| mkGameValidator ||])
161             `PlutusTx.applyCode` PlutusTx.liftCode game
162             `PlutusTx.applyCode` PlutusTx.liftCode bsZero
163             `PlutusTx.applyCode` PlutusTx.liftCode bsOne)
164         $$$(PlutusTx.compile [|| wrap ||])
165     where
166         wrap = Scripts.wrapValidator @GameDatum @GameRedeemer
167
168     gameValidator :: Game -> Validator
169     gameValidator = Scripts.validatorScript . typedGameValidator
170
171     gameAddress :: Game -> Ledger.Address
172     gameAddress = scriptAddress . gameValidator
173
174     findGameOutput :: Game -> Contract w s Text (Maybe (TxOutRef,
175                                                 ChainIndexTxOut, GameDatum))
175     findGameOutput game = do
176         utxos <- utxosAt $ gameAddress game
177         return $ do
178             (oref, o) <- find f $ Map.toList utxos
179             dat       <- gameDatum $ either (const Nothing) Just $
180                             _ciTxOutDatum o
181             return (oref, o, dat)
182     where
183         f :: (TxOutRef, ChainIndexTxOut) -> Bool
184         f (_, o) = assetClassValueOf (_ciTxOutValue o) (gToken game) == 1
185
185     waitUntilTimeHasPassed :: AsContractError e => POSIXTime ->
186                                     Contract w s e ()
186     waitUntilTimeHasPassed t = do
187         s1 <- currentSlot
188         logInfo @String $ "current slot: " ++ show s1 ++
189                         ", waiting until " ++ show t
190         void $ awaitTime t >> waitNSlots 1
190         s2 <- currentSlot

```

```

191     logInfo @String $ "waited until: " ++ show s2
192
193 data FirstParams = FirstParams
194     { fpSecond      :: !PaymentPubKeyHash
195     , fpStake       :: !Integer
196     , fpPlayDeadline :: !POSIXTime
197     , fpRevealDeadline :: !POSIXTime
198     , fpNonce        :: !BuiltinByteString
199     , fpCurrency     :: !CurrencySymbol
200     , fpTokenName   :: !TokenName
201     , fpChoice       :: !GameChoice
202   } deriving (Show, Generic, FromJSON, ToJSON, ToSchema)
203
204 firstGame :: forall w s. FirstParams -> Contract w s Text ()
205 firstGame fp = do
206     pkh <- Contract.ownPaymentPubKeyHash
207     let game = Game
208         { gFirst      = pkh
209         , gSecond     = fpSecond fp
210         , gStake      = fpStake fp
211         , gPlayDeadline = fpPlayDeadline fp
212         , gRevealDeadline = fpRevealDeadline fp
213         , gToken      = AssetClass (fpCurrency fp, fpTokenName fp)
214     }
215     v = lovelaceValueOf (fpStake fp) <> assetClassValue (gToken game) 1
216     c = fpChoice fp
217     bs = sha2_256 $ fpNonce fp `appendByteString` if c == Zero then
218             bsZero else bsOne
219     tx = Constraints.mustPayToTheScript (GameDatum bs Nothing) v
220     ledgerTx <- submitTxConstraints (typedGameValidator game) tx
221     void $ awaitTxConfirmed $ getCardanoTxId ledgerTx
222     logInfo @String $ "made first move: " ++ show (fpChoice fp)
223
224     waitUntilTimeHasPassed $ fpPlayDeadline fp
225
226     m <- findGameOutput game
227     now <- currentTime
228     case m of
229         Nothing          -> throwError "game output not found"
230         Just (oref, o, dat) -> case dat of
231             GameDatum _ Nothing -> do
232                 logInfo @String "second player did not play"
233                 let lookups = Constraints.unspentOutputs
234                     (Map.singleton oref o) <>
235                     Constraints.otherScript (gameValidator game)

```

```

234          tx'      = Constraints.mustSpendScriptOutput oref
235                  (Redeemer $ PlutusTx.toBuiltinData ClaimFirst)
236                  <> Constraints.mustValidateIn (from now)
237          ledgerTx' <- submitTxConstraintsWith @Gaming lookups tx'
238          void $ awaitTxConfirmed $ getCardanoTxId ledgerTx'
239          logInfo @String "reclaimed stake"
240
241          GameDatum _ (Just c') | c' == c -> do
242
243              logInfo @String "second player played and lost"
244              let lookups = Constraints.unspentOutputs
245                      (Map.singleton oref o) <>
246                      Constraints.otherScript (gameValidator game)
247              tx'      = Constraints.mustSpendScriptOutput oref
248                      (Redeemer $ PlutusTx.toBuiltinData $ Reveal $
249                          fpNonce fp) <>
250                          Constraints.mustValidateIn (to $ now + 1000)
251              ledgerTx' <- submitTxConstraintsWith @Gaming lookups tx'
252              void $ awaitTxConfirmed $ getCardanoTxId ledgerTx'
253              logInfo @String "victory"
254
255          _ -> logInfo @String "second player played and won"
256
257 data SecondParams = SecondParams
258     { spFirst        :: !PaymentPubKeyHash
259     , spStake       :: !Integer
260     , spPlayDeadline :: !POSIXTime
261     , spRevealDeadline :: !POSIXTime
262     , spCurrency    :: !CurrencySymbol
263     , spTokenName   :: !TokenName
264     , spChoice      :: !GameChoice
265     } deriving (Show, Generic, FromJSON, ToJSON, ToSchema)
266
267 secondGame :: forall w s. SecondParams -> Contract w s Text ()
268 secondGame sp = do
269     pkh <- Contract.ownPaymentPubKeyHash
270     let game = Game
271         { gFirst        = spFirst sp
272         , gSecond       = pkh
273         , gStake        = spStake sp
274         , gPlayDeadline = spPlayDeadline sp
275         , gRevealDeadline = spRevealDeadline sp
276         , gToken        = AssetClass (spCurrency sp, spTokenName sp)
277         }
278     m <- findGameOutput game

```

```

275   case m of
276     Just (oref, o, GameDatum bs Nothing) -> do
277       logInfo @String "running game found"
278       now <- currentTime
279       let token    = assetClassValue (gToken game) 1
280       let v        = let x = lovelaceValueOf (spStake sp) in x <> x
281                     <> token
282       c          = spChoice sp
283       lookups = Constraints.unspentOutputs
284                     (Map.singleton oref o) <>
285                     Constraints.otherScript (gameValidator game) <>
286                     Constraints.typedValidatorLookups
287                     (typedGameValidator game)
288       tx        = Constraints.mustSpendScriptOutput oref (Redeemer $ PlutusTx.toBuiltinData $ Play c) <>
289                     Constraints.mustPayToTheScript
290                     (GameDatum bs $ Just c) v <>
291                     Constraints.mustValidateIn (to now)
292       ledgerTx <- submitTxConstraintsWith @Gaming lookups tx
293       let tid = getCardanoTxId ledgerTx
294       void $ awaitTxConfirmed tid
295       logInfo @String $ "made second move: " ++ show (spChoice sp)
296
297       waitUntilTimeHasPassed $ spRevealDeadline sp
298
299       m'    <- findGameOutput game
300       now'  <- currentTime
301       case m' of
302         Nothing           -> logInfo @String "first player won"
303         Just (oref', o', _) -> do
304           logInfo @String "first player didn't reveal"
305           let lookups' = Constraints.unspentOutputs
306             (Map.singleton oref' o') <>
307             Constraints.otherScript
308               (gameValidator game)
309           tx'      = Constraints.mustSpendScriptOutput oref'
310             (Redeemer $ PlutusTx.toBuiltinData
311               ClaimSecond) <>
312             Constraints.mustValidateIn (from now') <>
313             Constraints.mustPayToPubKey (spFirst sp)
314               (token <> adaValueOf
315                 (getAda minAdaTxOut))
316           ledgerTx' <- submitTxConstraintsWith @Gaming
317                         lookups' tx'
318           void $ awaitTxConfirmed $ getCardanoTxId ledgerTx'

```

```

308             logInfo @String "second player won"
309
310         _ -> logInfo @String "no running game found"
311
312 type GameSchema = Endpoint "first" FirstParams .\|
313                 Endpoint "second" SecondParams
314
314 endpoints :: Contract () GameSchema Text ()
315 endpoints = awaitPromise (first `select` second) >> endpoints
316   where
317     first  = endpoint @"first" firstGame
318     second = endpoint @"second" secondGame

```

We call this code EvenOdd because if the sum of the choices is even the first player wins and if it is odd the second player wins. First we create the data type game that will be used as a parameter for the contract (40-47). There we define the first and second player with their public key hashes. Then we also define their stake, the playing deadline and the revealing deadline. And in the end we define a NFT token. Since the game contains some state that is changing and UTXOs together with their datums are immutable we need to create a new UTXO every time the state of the game is changing. And to be able to connect the old UTXO with the new one we can use an NFT that exists only once and gets assigned to the new UTXO every time the state changes. We then call this token a stake token. Another reason we need this NFT is that somebody could create a UTXO at the same address with the same datum and would try to disturb the game. So in order to uniquely be able to identify our UTXO with which we start the game we need an NFT that exists only once. The type game choice defines the two moves the players can make (51-52). Then we derive the Plutus equality for the game choice type (54-58). For this to work with template Haskell we need to add the inalienable pragma. We will use the game datum as state information for the contract (62-63). The byte string there is the hash that the first player submits and maybe game choice is the move of the second player. It's a maybe because in the beginning the second player has not yet moved. We implement also the Plutus equality for the game datum (65-67). Next we implement the game redeemer with the options that corresponds to our player actions (71-72). Play means when the second player moves and makes a game choice, reveal is for the case that first player reveals his nonce which is the byte string argument, claim first is the case when the second player does not move and the first player claims back his stake and claim second is for the case if the first player does not reveal his nonce and the deadline passes for the second player to collect his earnings. Then we define two helper functions. The function *lovelaces* when given a value extracts the amount of lovelaces (76-78). And the function *gameDatum* given a maybe datum tries to deserialize that if it's a Just to a maybe game datum (80-84).

Now we write our game validator function. The first parameter is the game parameter that we defined in the beginning. The second and third parameters are the byte strings with the digit zero and the digit one that represents the choice. Then we take in the datum, redeemer and context. Let's look first at the helper function before we come to the main body. For the *ownInput* function we use the function *findOwnInput* (Figure 21) (123-126).

```
findOwnInput :: ScriptContext -> Maybe TxInInfo
```

Find the input currently being validated.

Figure 21 - *findOwnInput* function

It takes in a script context and produces a maybe transaction input info. If a script would be used for minting this function would return Nothing.

```
data TxInInfo
```

An input of a pending transaction.

Constructors

```
TxInInfo
```

```
txInInfoOutRef :: TxOutRef
```

```
txInInfoResolved :: TxOut
```

Figure 22 - Transaction input info

What we are interested in inside the *TxInInfo* type is the transaction output. The idea of our game is that with each state change we consume an UTXO and produce another one at the same address. For the *ownOutput* function we use the *getContinuingOutputs* function.

```
getContinuingOutputs :: ScriptContext -> [TxOut]
```

It takes a script context and returns a list of transaction outputs. And in our case we expect there is exactly one output sitting at our address (128-131). The output datum function should give us a game datum of our own output (133-136). We use the function *findDatum* that takes in a datum hash and a transaction info and then returns a maybe datum (Figure 23). A maybe datum because we said that the producing transaction can optionally include the datum in the output. It is required only to include the hash of the datum.

```
findDatum :: DatumHash -> TxInfo -> Maybe Datum
```

| Find the data corresponding to a data hash, if there is one

Figure 23 - Find datum function

The check nonce function is for the case that the first player has won and wants to prove it by revealing his nonce and proving that the hash submitted at the beginning of the game fits this nonce (138-144). The first argument is the hash he submitted, the second is the nonce and the third is the move that the second player made. To compute the hash we take the nonce concatenate it with the byte string and apply the SHA 256 hash function. At the end we define the *nftToFirst* function (146-147). The idea is after the game finishes the 1st player gets back his NFT no matter who won the game.

One thing to notice about our code at this point is we haven't spent much consideration about the staking address. In general a user could use an address where the payment part goes to the winner but the staking part goes to the looser, since these 2 addresses could be different. So one has to be careful when specifying the input data and take this consideration into account. Let's now look at the conditions in the body of the validator function. There is one condition that applies to all cases simultaneously. That is the input we are validating has to contain the state token (89). Then the rules after that depend on the situation. The first situation is when the 1st player has moved and 2nd player is moving now and chooses the move c (91-97). First we check that the second player actually signs the transaction. Then we check that the first player has put down his stake. In the third condition we check that the second player added in this transaction his own stake. After that we check the datum from the transaction context. Then we check that the move has to happen before the first deadline. And in the end we check that the NFT is passed to the transaction output UTXO. The second situation is both players have moved and the 1st player has won (99-104). And to prove it and collect his winnings he has to reveal his nonce. So first we check that the transaction is signed by the first player. Then the nonce must agree with the hash we submitted earlier. He must do this before the reveal deadline. The input must contain the stakes of both players. And finally the NFT must go back to the first player. The third situation is that the 2nd player does not move and the 1st player wants his stake back (106-110). So the transaction must be signed by the first player has to have a validity interval after the deadline has passed. The first player has provided his stake and he must get back his NFT token. The last situation is both players have moved but the 1st player has lost and did not reveal his nonce (112-116). So the transaction must be signed by the 2nd player. The reveal deadline should pass. The input must contain the stakes of both players. And the token has to go back to the first player. In all other cases we fail validation.

Let's look at the rest of the on-chain code. First we define the helper that bundles information what the datum and the redeemer types are (149-152). Then we define the byte strings that we use for choice 1 and choice 2. Then we define our typed validator and compile the code with our parameters (158-166). After that we define the game validator and game address.

For the off-chain code we first define two helper functions. The *findGameOutput* function takes as input a game type parameter and then in the contract monad tries to find the UTXO (174-180). And because it could fail we are returning a maybe type. We return the transaction output reference, the transaction output chain index and the game datum. We use the find function that has following type signature:

```
Prelude> import Data.List
Prelude Data.List> :i find
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
Prelude Data.List> find even [1 :: Int, 3, 4, 5, 6]
Just 4
```

It works like this: if it finds an element in the container that satisfies the initial condition then it returns a Just of that element. Our helper function *f* checks weather the output contains the token. With the second function *waitForTimeHasPassed* we take in a posix time and wait until that time has passed and then we wait for 1 more slot (185-191). Then come the two contracts for the two players with the corresponding input parameters for each contract. In the first contract we first get our public key. Then we define several parameters (207-218). First the value of the game type, second the amount of lovelace we put in as our stake plus the NFT we put in, then *c* is our choice, next we compute the hash of our nonce with the appended choice and in the end we define our transactions. Then we submit, wait for the transaction to process and log a message. Now the second player has a chance to move but it must happen before the play deadline so as first player we wait until the deadline has passed (223). And then there are several cases. First we check if we find the UTXO containing the NFT and if yes we check the datum. One case is that it is Nothing so the 2nd player did not moved (230). In that case as constraints we say we must spend this UTXO we found with this redeemer and as lookups we need to provide the UTXO and the validator. Then the second case is that the second player did move and he chooses the same move as we did so we won (240). So we have to reveal our nonce. We use the Reveal nonce game redeemer. And we need to submit this transaction before the deadline for revealing has passed. The third case is the second player won and in that case we don't do anything.

Now for the second contract for player 2 the input parameters are very similar (253-261). We do not need to provide the second players public key hash because we look up our own public key hash. Then we define the game value (266-273). Next we try to find the UTXO that contains

the NFT. If we find it we continue by defining several parameters (280-287). We define the token, v represents the output which is twice the stake and the NFT and c is our choice. For the transaction we put the constraints that we must spend the existing UTXO with the redeemer `Play` that holds our choice. Then we create a new UTXO with the updated datum and we must do this before the deadline passes. For the lookups we provide the UTXO. Because we are consuming the script output we need the validator and because we are also producing we need the script instance. After we defined our parameters we submit the transaction, wait for confirmation and log a message. Then we wait until the reveal deadline has passed so the first player can make his move. Next we try again to find the UTXO which could be now a different one. If we do not find it the first player made his reveal and had won, so we do nothing. If we do find it we must spend the UTXO that we have found, we must do this after the deadline has passed and return the NFT to the first player. Because every UTXO has to contain some ADA we must add some ADA when returning the NFT. Then we submit the transaction and log that the 2nd player has won. In the end we define the schema and define the endpoints contract. We can test this now with the emulator trace defined in the `TestEvenOdd.hs` file.

```

1 {-# LANGUAGE DataKinds          #-}
2 {-# LANGUAGE FlexibleContexts   #-}
3 {-# LANGUAGE MultiParamTypeClasses #-}
4 {-# LANGUAGE NoImplicitPrelude   #-}
5 {-# LANGUAGE NumericUnderscores #-}
6 {-# LANGUAGE OverloadedStrings   #-}
7 {-# LANGUAGE ScopedTypeVariables #-}
8 {-# LANGUAGE TypeApplications    #-}
9 {-# LANGUAGE TypeFamilies        #-}

10
11 module Week07.TestEvenOdd
12   ( test
13   , test'
14   , GameChoice(..)
15   ) where
16
17 import           Control.Monad          hiding (fmap)
18 import           Control.Monad.Freer.Extras as Extras
19 import           Data.Default           (Default(..))
20 import qualified Data.Map               as Map
21 import           Ledger
22 import           Ledger.TimeSlot
23 import           Ledger.Value
24 import           Ledger.Ada            as Ada
25 import           Plutus.Trace.Emulator as Emulator
26 import           PlutusTx.Prelude

```

```

27 import Prelude (IO, Show(..))
28 import Wallet.Emulator.Wallet
29
30 import Week07.EvenOdd
31
32 test :: IO ()
33 test = do
34     test' Zero Zero
35     test' Zero One
36     test' One Zero
37     test' One One
38
39 w1, w2 :: Wallet
40 w1 = knownWallet 1
41 w2 = knownWallet 2
42
43 test' :: GameChoice -> GameChoice -> IO ()
44 test' c1 c2 = runEmulatorTraceIO' def emCfg $ myTrace c1 c2
45 where
46     emCfg :: EmulatorConfig
47     emCfg = def { _initialChainState = Left $ Map.fromList
48                  [ (w1, v <>> assetClassValue (AssetClass
49                                gameTokenCurrency, gameTokenName)) 1)
50                  , (w2, v)
51                  ]
52
53     v :: Value
54     v = Ada.lovelaceValueOf 1_000_000_000
55
56 gameTokenCurrency :: CurrencySymbol
57 gameTokenCurrency = "ff"
58
59 gameTokenName :: TokenName
60 gameTokenName = "STATE TOKEN"
61
62 myTrace :: GameChoice -> GameChoice -> EmulatorTrace ()
63 myTrace c1 c2 = do
64     Extras.logInfo $ "first move: " ++ show c1 ++
65                 ", second move: " ++ show c2
66
67     h1 <- activateContractWallet w1 endpoints
68     h2 <- activateContractWallet w2 endpoints
69
70     let pkh1      = mockWalletPaymentPubKeyHash w1

```

```

70      pkh2      = mockWalletPaymentPubKeyHash w2
71      stake     = 100_000_000
72      deadline1 = slotToBeginPOSIXTime def 5
73      deadline2 = slotToBeginPOSIXTime def 10
74
75      fp = FirstParams
76          { fpSecond      = pkh2
77            , fpStake       = stake
78            , fpPlayDeadline = deadline1
79            , fpRevealDeadline = deadline2
80            , fpNonce        = "SECRETNONCE"
81            , fpCurrency    = gameTokenCurrency
82            , fpTokenName   = gameTokenName
83            , fpChoice       = c1
84          }
85      sp = SecondParams
86          { spFirst       = pkh1
87            , spStake      = stake
88            , spPlayDeadline = deadline1
89            , spRevealDeadline = deadline2
90            , spCurrency    = gameTokenCurrency
91            , spTokenName   = gameTokenName
92            , spChoice      = c2
93          }
94
95      callEndpoint @"first" h1 fp
96
97      void $ Emulator.waitNSlots 3
98
99      callEndpoint @"second" h2 sp
100
101     void $ Emulator.waitNSlots 10

```

The idea is that you can test our code for all possible choices. So we define our *test* IO action such that all possible game choices are covered (32-37). We define the two wallets (39-41). The *test'* function takes in 2 choices and runs the emulator trace where it assigns 1000 ADA to both wallets and the token to wallet 1 (43-54). Then we define our trace that also takes as input our two choices. We start the contract for both wallets and save the returned handle (66-67). We look up the payment public key hashes and use the stake of 100 ADA. Then we define the deadlines which are a bit short in our case but for the example it will be fine (72-73). Next we define the parameters for the first and second player. Now we can call the endpoint for the first player, wait for 3 slots, call the endpoint for the second player and wait for 10 slots.

7.3 State Machines

A state machine (SM) is a system that starts in one state and normally there are available transitions to other states. And there can be also final states from which there are no transitions left. For our game we can also draw a state diagram where all the nodes represent a state and all the arrows represent transitions (Figure 24).

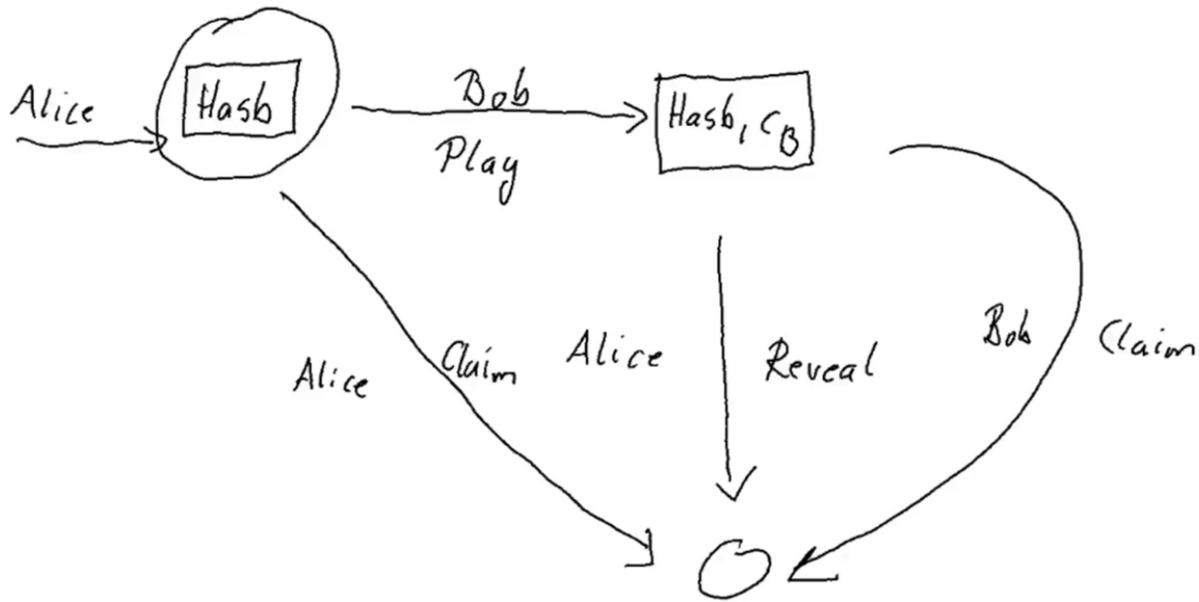


Figure 24 - Game state diagram

In the blockchain the states will be represented by UTXO sitting at the state machine script address. The state of the state machine will be the datum of the UTXO and a transition is represented by a transaction. There is special support in the Plutus libraries to implement such state machines which makes our code shorter compared to the case without using SM. The module `Plutus.Contract.StateMachine` contains all the code for using SM. Let's first look at the definition of a `StateMachine` type (Figure 25). It takes two parameters that are state and input which correspond to datum and redeemer. It is a record type with four fields. The `smTransition` function defines from which state using which transition you can define another state. The `State` type is defined with the state data and a state value of type `Value`. If the transition is not allowed we return a `Nothing` in the function, else we return a tuple from which the second component is the new state. And the `TxConstraints` define additional constraints that the transaction that does the transition must have. The `smFinal` function tells us whether we are transitioning to a final state. If yes then we do not produce a new UTXO and there is no value attached to the transition. The `smCheck` function takes in the datum, redeemer and context and basically makes an additional check that can't be expressed with the transaction constraints.

<pre>data StateMachine s i</pre>	# 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 <table border="1"> <tr> <td>StateMachine</td><td></td></tr> <tr> <td><code>smTransition :: State s -> i -> Maybe (TxConstraints Void Void, State s)</code></td><td>The transition function of the state machine. <code>Nothing</code> indicates an invalid transition from the current state.</td></tr> <tr> <td><code>smFinal :: s -> Bool</code></td><td>Check whether a state is the final state</td></tr> <tr> <td><code>smCheck :: s -> i -> ScriptContext -> Bool</code></td><td>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.</td></tr> <tr> <td><code>smThreadToken :: Maybe ThreadToken</code></td><td>The ThreadToken that identifies the contract instance. Make one with <code>getThreadToken</code> and pass it on to <code>mkStateMachine</code>. Initialising the machine will then mint a thread token value.</td></tr> </table>		StateMachine		<code>smTransition :: State s -> i -> 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 -> Bool</code>	Check whether a state is the final state	<code>smCheck :: s -> i -> ScriptContext -> 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.	<code>smThreadToken :: Maybe ThreadToken</code>	The ThreadToken that identifies the contract instance. Make one with <code>getThreadToken</code> and pass it on to <code>mkStateMachine</code> . Initialising the machine will then mint a thread token value.
StateMachine											
<code>smTransition :: State s -> i -> 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 -> Bool</code>	Check whether a state is the final state										
<code>smCheck :: s -> i -> ScriptContext -> 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.										
<code>smThreadToken :: Maybe ThreadToken</code>	The ThreadToken that identifies the contract instance. Make one with <code>getThreadToken</code> and pass it on to <code>mkStateMachine</code> . Initialising the machine will then mint a thread token value.										

Figure 25 - State machine type

<pre>data State s</pre>
Constructors

State

<code>stateData :: s</code>
<code>stateValue :: Value</code>

Figure 26 - State type

The `smThreadToken` serves the same purpose as the NFT we were using in the previous chapter to identify our UTXO for the game (Figure 27).

<pre>data ThreadToken</pre>
Constructors

<code>ThreadToken TxOutRef CurrencySymbol</code>
--

Figure 27 - Thread token

If we look at the thread token it's a reference to a UTXO and a currency symbol. And the UTXO in our case will be the one that uniquely identifies the minting transaction of the NFT. So we don't have to worry about the NFT, it will automatically be taken care of. So let's look now at the code in *StateMachine.hs* that uses now state machines.

```

1  {-# LANGUAGE DataKinds          #-}
2  {-# LANGUAGE DeriveAnyClass     #-}
3  {-# LANGUAGE DeriveGeneric      #-}
4  {-# LANGUAGE FlexibleContexts   #-}
5  {-# LANGUAGE MultiParamTypeClasses #-}
6  {-# LANGUAGE NoImplicitPrelude   #-}
7  {-# LANGUAGE OverloadedStrings   #-}
8  {-# LANGUAGE ScopedTypeVariables #-}
9  {-# LANGUAGE TemplateHaskell     #-}
10 {-# LANGUAGE TypeApplications    #-}
11 {-# LANGUAGE TypeFamilies       #-}
12 {-# LANGUAGE TypeOperators      #-}
13
14 module Week07.StateMachine
15   ( Game(..)
16   , GameChoice(..)
17   , FirstParams(..)
18   , SecondParams(..)
19   , GameSchema
20   , Last(..)
21   , ThreadToken
22   , Text
23   , endpoints
24   ) where
25
26 import           Control.Monad           hiding (fmap)
27 import           Data.Aeson             (FromJSON, ToJSON)
28 import           Data.Monoid            (Last(..))
29 import           Data.Text              (Text, pack)
30 import           GHC.Generics          (Generic)
31 import           Ledger                hiding (singleton)
32 import           Ledger.Ada             as Ada
33 import           Ledger.Constraints    as Constraints
34 import           Ledger.Typed.Tx
35 import qualified Ledger.Typed.Scripts  as Scripts
36 import           Plutus.Contract        as Contract
37 import           Plutus.Contract.StateMachine
38 import qualified PlutusTx
39 import           PlutusTx.Prelude       hiding (Semigroup(..), check,
unless)
```

```

40 import           Playground.Contract          (ToSchema)
41 import           Prelude                      (Semigroup(..), Show(..),
42                                         String)
43
44 data Game = Game
45   { gFirst        :: !PaymentPubKeyHash
46   , gSecond       :: !PaymentPubKeyHash
47   , gStake        :: !Integer
48   , gPlayDeadline :: !POSIXTime
49   , gRevealDeadline :: !POSIXTime
50   , gToken        :: !ThreadToken
51 } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq)
52
53 PlutusTx.makeLift ''Game
54
55 data GameChoice = Zero | One
56   deriving (Show, Generic, FromJSON, ToJSON, ToSchema, Prelude.Eq,
57             Prelude.Ord)
58
59 instance Eq GameChoice where
60   {-# INLINABLE (==) #-}
61   Zero == Zero = True
62   One == One = True
63   _ == _ = False
64
65 PlutusTx.unstableMakeIsData ''GameChoice
66
67 data GameDatum = GameDatum BuiltinByteString (Maybe GameChoice) | Finished
68   deriving Show
69
70 instance Eq GameDatum where
71   {-# INLINABLE (==) #-}
72   GameDatum bs mc == GameDatum bs' mc' = (bs == bs') && (mc == mc')
73   Finished == Finished = True
74   _ == _ = False
75
76 PlutusTx.unstableMakeIsData ''GameDatum
77
78 data GameRedeemer = Play GameChoice | Reveal BuiltinByteString |
79                           ClaimFirst | ClaimSecond
80   deriving Show
81
82 PlutusTx.unstableMakeIsData ''GameRedeemer

```

```

82 {-# INLINABLE lovelaces #-}
83 lovelaces :: Value -> Integer
84 lovelaces = Ada.getLovelace . Ada.fromValue
85
86 {-# INLINABLE gameDatum #-}
87 gameDatum :: TxOut -> (DatumHash -> Maybe Datum) -> Maybe GameDatum
88 gameDatum o f = do
89     dh      <- txOutDatum o
90     Datum d <- f dh
91     PlutusTx.fromBuiltinData d
92
93 {-# INLINABLE transition #-}
94 transition :: Game -> State GameDatum -> GameRedeemer -> Maybe
95             (TxConstraints Void Void, State GameDatum)
96 transition game s r = case (stateValue s, stateData s, r) of
97     (v, GameDatum bs Nothing, Play c)
98     | lovelaces v == gStake game      ->
99         Just ( Constraints.mustBeSignedBy (gSecond game) <>
100            Constraints.mustValidateIn (to $ gPlayDeadline game)
101            , State (GameDatum bs $ Just c) (lovelaceValueOf $
102                2 * gStake game) )
103
104     (v, GameDatum _ (Just _), Reveal _)
105     | lovelaces v == (2 * gStake game) ->
106         Just ( Constraints.mustBeSignedBy (gFirst game) <>
107            Constraints.mustValidateIn (to $ gRevealDeadline game)
108            , State Finished mempty )
109
110     (v, GameDatum _ Nothing, ClaimFirst)
111     | lovelaces v == gStake game      ->
112         Just ( Constraints.mustBeSignedBy (gFirst game) <>
113            Constraints.mustValidateIn (from $ 1 + gPlayDeadline game)
114            , State Finished mempty )
115
116     -                                         -> Nothing
117
118 {-# INLINABLE final #-}
119 final :: GameDatum -> Bool

```

```

120 final Finished = True
121 final _        = False
122
123 {-# INLINABLE check #-}
124 check :: BuiltinByteString -> BuiltinByteString -> GameDatum ->
     GameRedeemer -> ScriptContext -> Bool
125 check bsZero' bsOne' (GameDatum bs (Just c)) (Reveal nonce) _ =
126     sha2_256 (nonce `appendByteString` if c == Zero then bsZero' else
                  bsOne') == bs
127 check _          _          _          _          _          _ = True
128
129 {-# INLINABLE gameStateMachine #-}
130 gameStateMachine :: Game -> BuiltinByteString -> BuiltinByteString ->
     StateMachine GameDatum GameRedeemer
131 gameStateMachine game bsZero' bsOne' = StateMachine
132   { smTransition = transition game
133   , smFinal      = final
134   , smCheck       = check bsZero' bsOne'
135   , smThreadToken = Just $ gToken game
136   }
137
138 {-# INLINABLE mkGameValidator #-}
139 mkGameValidator :: Game -> BuiltinByteString -> BuiltinByteString ->
     GameDatum -> GameRedeemer -> ScriptContext -> Bool
140 mkGameValidator game bsZero' bsOne' = mkValidator $ gameStateMachine
                                         game bsZero' bsOne'
141
142 type Gaming = StateMachine GameDatum GameRedeemer
143
144 bsZero, bsOne :: BuiltinByteString
145 bsZero = "0"
146 bsOne  = "1"
147
148 gameStateMachine' :: Game -> StateMachine GameDatum GameRedeemer
149 gameStateMachine' game = gameStateMachine game bsZero bsOne
150
151 typedGameValidator :: Game -> Scripts.TypedValidator Gaming
152 typedGameValidator game = Scripts.mkTypedValidator @Gaming
153   ($$($PlutusTx.compile [|| mkGameValidator ||])
154     `PlutusTx.applyCode` PlutusTx.liftCode game
155     `PlutusTx.applyCode` PlutusTx.liftCode bsZero
156     `PlutusTx.applyCode` PlutusTx.liftCode bsOne)
157   $$($PlutusTx.compile [|| wrap ||])
158   where
159     wrap = Scripts.wrapValidator @GameDatum @GameRedeemer

```

```

160
161 gameValidator :: Game -> Validator
162 gameValidator = Scripts.validatorScript . typedGameValidator
163
164 gameAddress :: Game -> Ledger.Address
165 gameAddress = scriptAddress . gameValidator
166
167 gameClient :: Game -> StateMachineClient GameDatum GameRedeemer
168 gameClient game = mkStateMachineClient $ StateMachineInstance
    (gameStateMachine' game) (typedGameValidator game)
169
170 data FirstParams = FirstParams
171     { fpSecond      :: !PaymentPubKeyHash
172     , fpStake       :: !Integer
173     , fpPlayDeadline :: !POSIXTime
174     , fpRevealDeadline :: !POSIXTime
175     , fpNonce        :: !BuiltinByteString
176     , fpChoice       :: !GameChoice
177   } deriving (Show, Generic, FromJSON, ToJSON, ToSchema)
178
179 mapError' :: Contract w s SMContractError a -> Contract w s Text a
180 mapError' = mapError $ pack . show
181
182 waitUntilTimeHasPassed :: AsContractError e => POSIXTime ->
    Contract w s e ()
183 waitUntilTimeHasPassed t = void $ awaitTime t >> waitNSlots 1
184
185 firstGame :: forall s. FirstParams -> Contract (Last ThreadToken) s Text ()
186 firstGame fp = do
187     pkh <- Contract.ownPaymentPubKeyHash
188     tt  <- mapError' getThreadToken
189     let game  = Game
190         { gFirst      = pkh
191         , gSecond     = fpSecond fp
192         , gStake      = fpStake fp
193         , gPlayDeadline = fpPlayDeadline fp
194         , gRevealDeadline = fpRevealDeadline fp
195         , gToken      = tt
196     }
197     client = gameClient game
198     v      = lovelaceValueOf (fpStake fp)
199     c      = fpChoice fp
200     bs    = sha2_256 $ fpNonce fp `appendByteString` if c == Zero then
                bsZero else bsOne
201     void $ mapError' $ runInitialise client (GameDatum bs Nothing) v

```

```

202     logInfo @String $ "made first move: " ++ show (fpChoice fp)
203     tell $ Last $ Just tt
204
205     waitUntilTimeHasPassed $ fpPlayDeadline fp
206
207     m <- mapError' $ getOnChainState client
208     case m of
209         Nothing    -> throwError "game output not found"
210         Just (o, _) -> case tyTxOutData $ ocsTxOut o of
211
212             GameDatum _ Nothing -> do
213                 logInfo @String "second player did not play"
214                 void $ mapError' $ runStep client ClaimFirst
215                 logInfo @String "first player reclaimed stake"
216
217             GameDatum _ (Just c') | c' == c -> do
218                 logInfo @String "second player played and lost"
219                 void $ mapError' $ runStep client $ Reveal $ fpNonce fp
220                 logInfo @String "first player revealed and won"
221
222             _ -> logInfo @String "second player played and won"
223
224 data SecondParams = SecondParams
225   { spFirst        :: !PaymentPubKeyHash
226   , spStake       :: !Integer
227   , spPlayDeadline :: !POSIXTime
228   , spRevealDeadline :: !POSIXTime
229   , spChoice      :: !GameChoice
230   , spToken       :: !ThreadToken
231   } deriving (Show, Generic, FromJSON, ToJSON)
232
233 secondGame :: forall w s. SecondParams -> Contract w s Text ()
234 secondGame sp = do
235     pkh <- Contract.ownPaymentPubKeyHash
236     let game   = Game
237         { gFirst        = spFirst sp
238         , gSecond       = pkh
239         , gStake        = spStake sp
240         , gPlayDeadline = spPlayDeadline sp
241         , gRevealDeadline = spRevealDeadline sp
242         , gToken        = spToken sp
243         }
244     client = gameClient game
245     m <- mapError' $ getOnChainState client
246     case m of

```

```

247      Nothing      -> logInfo @String "no running game found"
248      Just (o, _) -> case tyTxOutData $ ocsTxOut o of
249          GameDatum _ Nothing -> do
250              logInfo @String "running game found"
251              void $ mapError' $ runStep client $ Play $ spChoice sp
252              logInfo @String $ "made second move: " ++ show (spChoice sp)
253
254              waitUntilTimeHasPassed $ spRevealDeadline sp
255
256              m' <- mapError' $ getOnChainState client
257              case m' of
258                  Nothing -> logInfo @String "first player won"
259                  Just _ -> do
260                      logInfo @String "first player didn't reveal"
261                      void $ mapError' $ runStep client ClaimSecond
262                      logInfo @String "second player won"
263
264                  _ -> throwError "unexpected datum"
265
266 type GameSchema = Endpoint "first" FirstParams ./\
267                   Endpoint "second" SecondParams
268
269 endpoints :: Contract (Last ThreadToken) GameSchema Text ()
270 endpoints = awaitPromise (first `select` second) >> endpoints
271   where
272       first  = endpoint @"first"  firstGame
273       second = endpoint @"second" secondGame

```

In the beginning we again define our game type (44-51). The difference to the previous example is that the token was of type asset class and now it's of type thread token. The game choice stays the same. What changes is the game datum where we add the second constructor called *Finished* that we did not need before (66-67). We need it for state machine mechanism to work. And when we make the Eq instance for the game datum we take this second constructor in account. The redeemer and the two helper functions are the same as before (77-91). Then we write the transition function for the state machine. It takes the game parameter and then the state datum and redeemer. And then we return a Nothing if the transition is not allowed or just a pair of new state and constraints on the transaction. First we again have the case where the 1st player has already moved and the 2nd player wants to make his move with choice *c*. The parameter *s* holds the combination of datum and value. The *stateValue* parameter gives us the value of the UTXO we are consuming and *stateData* gives us the datum. So first we check that the value is the stake of the game (97). If the check passes we return a Just where we put the constraints on the transaction for the signature and validity interval and include the new state.

Compared to our previous code where we did not use SM all conditions from the first case are also satisfied in our first SM case. The second case is when the 2nd player has moved and the 1st player realizes he has won and wants to reveal his nonce. Compared to our previous code the condition for revealing the nonce is missing here because we cannot formulate it as a constraint and we will take care of it later. We also do not return the token because it gets burned when we reached our final state. The third case is where the 2nd player does not move so the 1st player wants to reclaim his stake. Again if we compare the conditions we see we satisfy all as in the non SM example. The last case is that the 2nd player has moved and the 1st player does not reveal his nonce. Again all the all conditions are satisfied as in the non SM example. In the end we say for all other combinations we return nothing. All together the code gets shorter because we do not need any helper functions and do not need to take care of the NFT. What we have to add for the SM is the final state which basically says finished (118-121). Because we left out the nonce check in the transition function we define it now (123-127).

Next we can define our state machine that takes in the game parameter and two byte strings (129-136). In the body we just provide the four fields we have just defined. Our old function *mkGameValidator* can now be replaced with the state machine and we use the *mkValidator* function to transform it (138-140). We define the *gameStateMachine'* where we fix the two additional parameters and we can use then this function in the off-chain code (148-149). The compilation process, computation of the validator and the address is same as in the previous example. What we additionally define is the state machine client and we need this to be able to interact with the SM from our wallet (167-168) (Figure 28).

```
data StateMachineClient s i # Source

Client-side definition of a state machine.

Constructors

StateMachineClient
  scInstance :: StateMachineInstance s i
  scChooser :: [OnChainState s i] -> Either SMContractError (OnChainState s i)

The instance of the state
machine, defining the
machine's transitions, its
final states and its check
function.

A function that chooses the
relevant on-chain state,
given a list of all potential
on-chain states found at the
contract address.
```

Figure 28 - State machine client

If we look at the definition we see it has 2 fields. The state machine instance field is defined by two new fields that are the state machine and the validator code for this SM (Figure 29).

```

data StateMachineInstance s i

```

Constructors

StateMachineInstance	
stateMachine :: StateMachine s i	The state machine specification.
typedValidator :: TypedValidator (StateMachine s i)	The validator code for this state machine.

Figure 29 - State machine instance

The chooser field handles the case if we do not use the thread token mechanism and have several UTXOs sitting at our address. With it we can pick the right one. Now we come to the off-chain code. The *FirstParams* type is defined same way as in the previous example. The state machine contracts have a specific constraint on the Error type called *SMContractError*. But we want to use text for the error type so we define a function that handles the conversion (179-180). The helper function *waitUntilTimeHasPassed* is the same as in the previous code. In the *firstGame* contract we again first look up our own key and then we get our thread token (188). Then we define the game parameter (189-196) and after it we define the game client. Next three variables the stake, choice and hash are same as before. In line 201 the *runInitialise* function first mints the NFT corresponding to the thread token, then it creates the UTXO at the state machine address and we give the datum and value as input arguments. After that we log that we have made the first move and then we use the *tell* function to communicate the thread token. This is for the second player to be able to find the game. Then we wait for the 2nd player to move. In the previous example after the wait we needed to use the helper function *findGameOutput* which we can now replace with the *getOnChainState* function (Figure 30).

```

getOnChainState :: (AsSMContractError e, FromData state, ToData state) => StateMachineClient
state i -> Contract w schema e (Maybe (OnChainState state i, Map TxOutRef ChainIndexTxOut))

```

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.

Figure 30 - Get onchain state

It takes in a state machine client and returns a maybe of on chains state and a map with transaction output references as keys and chain indexes as values. The on-chain state has 3 fields available but we use only the first two (Figure 31). The first two fields are further displayed in Figure 32 and Figure 33. The typed script transaction output contains the transaction output and datum that is already deserialized which in our case means it would be of type *GameDatum*. And the typed script transaction output reference contains the transaction output reference and another typed script transaction output.

```
data OnChainState s i
```

Typed representation of the on-chain state of a state machine instance

Constructors

OnChainState	
<code>ocsTxOut :: TypedScriptTxOut (StateMachine s i)</code>	Typed transaction output
<code>ocsTxOutRef :: TypedScriptTxOutRef (StateMachine s i)</code>	Typed UTXO
<code>ocsTx :: ChainIndexTx</code>	Transaction that produced the output

Figure 31 - Onchain state

```
data TypedScriptTxOut a
```

A `TxOut` tagged by a phantom type: and the connection type of the output.

Constructors

<code>(FromData (DatumType a), ToData (DatumType a)) => TypedScriptTxOut</code>	
<code>tyTxOutTxOut :: TxOut</code>	
<code>tyTxOutData :: DatumType a</code>	

Figure 32 - Typed script transaction output

```
data TypedScriptTxOutRef a
```

A `TxOutRef` tagged by a phantom type: and the connection type of the output.

Constructors

TypedScriptTxOutRef	
<code>tyTxOutRefRef :: TxOutRef</code>	
<code>tyTxOutRefOut :: TypedScriptTxOut a</code>	

Figure 33 - Typed script transaction output reference

Once we have our on-chain state we can use the `tyTxOutData` function to directly access the datum. And after that we have the two cases that the second player has not moved or that he has moved and we see that we won. For the first case we do the reclaim (214). And for the second case we reveal our nonce (219). In both cases we use the `runStep` function that takes in a state machine client and the redeemer and returns a contract (Figure 34). It actually creates a transaction, submits it and transitions the state machine.

```

runStep # Source

:: forall w e state schema input. (AsSMContractError e, FromData state, ToData state, ToData input)
=> StateMachineClient state input

-> input

-> Contract w schema e (TransitionResult state input)

```

Run one step of a state machine, returning the new state.

Figure 34 - Run step

The `TransitionResult` type encodes whether the transition failed or succeeded. All the constraints are available to the state machine mechanism and that allows it to automatically compose the transaction that is needed. So we do not need any lookups or helper functions. The second game contract is similar to the first. We first look up our key hash and then define the game parameters and the client. With the `getOnChainState` function we again get the UTXO that represents the state machine. If we don't find the game we just log a message but if we do find it we look up the output and the game datum. And then again we use `runStep` to play our move. After it we wait until the reveal deadline has passed (254). We then again check the new state and if there is a Nothing the first player has won and we don't do anything. Else we claim our win (261). In order to test this we use the code from the `TestStateMachine.hs` example.

```

1 {-# LANGUAGE DataKinds      #-}
2 {-# LANGUAGE FlexibleContexts #-}
3 {-# LANGUAGE MultiParamTypeClasses #-}
4 {-# LANGUAGE NoImplicitPrelude #-}
5 {-# LANGUAGE NumericUnderscores #-}
6 {-# LANGUAGE OverloadedStrings #-}
7 {-# LANGUAGE ScopedTypeVariables #-}
8 {-# LANGUAGE TypeApplications #-}
9 {-# LANGUAGE TypeFamilies   #-}
10

```

```

11 module Week07.TestStateMachine
12     ( test
13     , test'
14     , GameChoice(..)
15     ) where
16
17 import           Control.Monad          hiding (fmap)
18 import           Control.Monad.Freer.Extras as Extras
19 import           Data.Default          (Default(..))
20 import           Ledger.TimeSlot
21 import           Plutus.Trace.Emulator as Emulator
22 import           PlutusTx.Prelude
23 import           Prelude               (IO, Show(..))
24 import           Wallet.Emulator.Wallet
25
26 import           Week07.StateMachine
27
28 test :: IO()
29 test = do
30     test' Zero Zero
31     test' Zero One
32     test' One Zero
33     test' One One
34
35 test' :: GameChoice -> GameChoice -> IO()
36 test' c1 c2 = runEmulatorTraceIO $ myTrace c1 c2
37
38 myTrace :: GameChoice -> GameChoice -> EmulatorTrace()
39 myTrace c1 c2 = do
40     Extras.logInfo $ "first move: " ++ show c1 ++ ", second move: " ++ show
c2
41
42     let w1 = knownWallet 1
43     let w2 = knownWallet 2
44
45     h1 <- activateContractWallet w1 endpoints
46     h2 <- activateContractWallet w2 endpoints
47
48     let pkh1      = mockWalletPaymentPubKeyHash w1
49     pkh2      = mockWalletPaymentPubKeyHash w2
50     stake      = 5_000_000
51     deadline1 = slotToEndPOSIXTime def 5
52     deadline2 = slotToEndPOSIXTime def 10
53
54     fp = FirstParams

```

```

55           { fpSecond      = pkh2
56           , fpStake       = stake
57           , fpPlayDeadline = deadline1
58           , fpRevealDeadline = deadline2
59           , fpNonce        = "SECRETNONCE"
60           , fpChoice       = c1
61       }
62
63   callEndpoint @"first" h1 fp
64
65   tt <- getTT h1
66
67   let sp = SecondParams
68       { spFirst      = pkh1
69       , spStake       = stake
70       , spPlayDeadline = deadline1
71       , spRevealDeadline = deadline2
72       , spChoice       = c2
73       , spToken        = tt
74   }
75
76   void $ Emulator.waitNSlots 3
77
78   callEndpoint @"second" h2 sp
79
80   void $ Emulator.waitNSlots 10
81 where
82     getTT :: ContractHandle (Last ThreadToken) GameSchema Text ->
83                 EmulatorTrace ThreadToken
84     getTT h = do
85       void $ Emulator.waitNSlots 1
86       Last m <- observableState h
87       case m of
88         Nothing -> getTT h
89         Just tt -> Extras.logInfo ("read thread token " ++ show tt) >>
90             return tt

```

It is very similar to the even odd test code. It is simple because we do not have to specify an initial state. We define our *test'* function that takes in our choices and runs the emulator trace (35-36). The default initial state for the SM is 10 wallets with 100 ADA each. Because of that we decrease the stake from 100 to 5 ADA. What is different in this trace is that we use the *slotToEndPOSIXTime* function. We use this function to fix the issue that off-chain part should construct a transaction that has a correct time interval but the conversion between slots and

posix time does not always work when using state machines because of unfaithfulness. So we see that the state machine approach automatically guarantees that the on-chain and off-chain code fit together. The produced transactions from it are correctly validated.

7.4 Homework

For homework we will modify the state machine code to implement the rock-paper-scissors game. Let's look at the code *RockPaperScissors.hs*.

```
1  data Game = Game
2      { gFirst        :: !PaymentPubKeyHash
3      , gSecond       :: !PaymentPubKeyHash
4      , gStake         :: !Integer
5      , gPlayDeadline :: !POSIXTime
6      , gRevealDeadline :: !POSIXTime
7      , gToken         :: !ThreadToken
8  } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq)
9
10 PlutusTx.makeLift ''Game
11
12 data GameChoice = Rock | Paper | Scissors
13     deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq, Prelude.Ord)
14
15 instance Eq GameChoice where
16     {-# INLINABLE (==) #-}
17     Rock    == Rock    = True
18     Paper   == Paper   = True
19     Scissors == Scissors = True
20     _       == _       = False
21
22 PlutusTx.unstableMakeIsData ''GameChoice
23
24 {-# INLINABLE beats #-}
25 beats :: GameChoice -> GameChoice -> Bool
26 beats Rock    Scissors = True
27 beats Paper   Rock    = True
28 beats Scissors Paper   = True
29 beats _       _       = False
30
31 data GameDatum = GameDatum BuiltinByteString (Maybe GameChoice) | Finished
32     deriving Show
33
34 instance Eq GameDatum where
35     {-# INLINABLE (==) #}
```

```

36     GameDatum bs mc == GameDatum bs' mc' = (bs == bs') && (mc == mc')
37     Finished          == Finished          = True
38     _                 == _                = False
39
40 PlutusTx.unstableMakeIsData ''GameDatum
41
42 data GameRedeemer = Play GameChoice | Reveal BuiltinByteString GameChoice |
43                                ClaimFirst | ClaimSecond
44     deriving Show
45
46 PlutusTx.unstableMakeIsData ''GameRedeemer
47
48 {-# INLINABLE lovelaces #-}
49 lovelaces :: Value -> Integer
50 lovelaces = Ada.getLovelace . Ada.fromValue
51
52 {-# INLINABLE transition #-}
53 transition :: Game -> State GameDatum -> GameRedeemer -> Maybe
54                               (TxConstraints Void Void, State GameDatum)
55 transition game s r = case (stateValue s, stateData s, r) of
56   (v, GameDatum bs Nothing, Play c)
57   | lovelaces v == gStake game           ->
58     Just ( Constraints.mustBeSignedBy (gSecond game) <>
59           Constraints.mustValidateIn (to $ gPlayDeadline game)
60           , State (GameDatum bs $ Just c) (lovelaceValueOf $ 2 *
61                                         gStake game) )
62
63   (v, GameDatum _ (Just c), Reveal _ c')
64   | (lovelaces v == (2 * gStake game)) &&
65     (c' `beats` c)                      ->
66     Just ( Constraints.mustBeSignedBy (gFirst game) <>
67           Constraints.mustValidateIn (to $ gRevealDeadline game)
68           , State Finished mempty )
69
70   | (lovelaces v == (2 * gStake game)) &&
71     (c' == c)                          ->
72     Just ( Constraints.mustBeSignedBy (gFirst game) <>
73           Constraints.mustValidateIn (to $ gRevealDeadline game) <>
74           Constraints.mustPayToPubKey (gSecond game)
75                                         (lovelaceValueOf $ gStake game)
76           , State Finished mempty )
77
78   (v, GameDatum _ Nothing, ClaimFirst)
79   | lovelaces v == gStake game           ->

```

```

                Just ( Constraints.mustBeSignedBy (gFirst game) <>
75                  Constraints.mustValidateIn (from $ 1 + gPlayDeadline game)
76                  , State Finished mempty )
77
78      (v, GameDatum _ (Just _), ClaimSecond)
79      | lovelaces v == (2 * gStake game)           ->
80          Just ( Constraints.mustBeSignedBy (gSecond game) <>
81              Constraints.mustValidateIn (from $ 1 + gRevealDeadline
82                                          game)
83              , State Finished mempty )
84
85      -
86      -> Nothing
87
88      {-# INLINABLE final #-}
89      final :: GameDatum -> Bool
90      final Finished = True
91      final _         = False
92
93      {-# INLINABLE check #-}
94      check :: BuiltinByteString -> BuiltinByteString -> BuiltinByteString ->
95          GameDatum -> GameRedeemer -> ScriptContext -> Bool
96      check bsRock' bsPaper' bsScissors' (GameDatum bs (Just _))
97          (Reveal nonce c) _ =
98          sha2_256 (nonce `appendByteString` toBS c) == bs
99      where
100        toBS :: GameChoice -> BuiltinByteString
101        toBS Rock      = bsRock'
102        toBS Paper     = bsPaper'
103        toBS Scissors  = bsScissors'
104        check _ _ _ _ _ = True
105
106      {-# INLINABLE gameStateMachine #-}
107      gameStateMachine :: Game -> BuiltinByteString -> BuiltinByteString ->
108          BuiltinByteString -> StateMachine GameDatum GameRedeemer
109      gameStateMachine game bsRock' bsPaper' bsScissors' = StateMachine
110          { smTransition = transition game
111          , smFinal     = final
112          , smCheck      = check bsRock' bsPaper' bsScissors'
113          , smThreadToken = Just $ gToken game
114          }
115
116      {-# INLINABLE mkGameValidator #-}
117      mkGameValidator :: Game -> BuiltinByteString -> BuiltinByteString ->
118          BuiltinByteString -> GameDatum -> GameRedeemer ->
119          ScriptContext -> Bool

```

```

112 mkGameValidator game bsRock' bsPaper' bsScissors' = mkValidator $  

    gameStateMachine game bsRock' bsPaper' bsScissors'  

113  

114 type Gaming = StateMachine GameDatum GameRedeemer  

115  

116 bsRock, bsPaper, bsScissors :: BuiltinByteString  

117 bsRock      = "R"  

118 bsPaper     = "P"  

119 bsScissors = "S"  

120  

121 gameStateMachine' :: Game -> StateMachine GameDatum GameRedeemer  

122 gameStateMachine' game = gameStateMachine game bsRock bsPaper bsScissors  

123  

124 typedGameValidator :: Game -> Scripts.TypedValidator Gaming  

125 typedGameValidator game = Scripts.mkTypedValidator @Gaming  

126   ($$($PlutusTx.compile [||| mkGameValidator ||])  

127     `PlutusTx.applyCode` PlutusTx.liftCode game  

128     `PlutusTx.applyCode` PlutusTx.liftCode bsRock  

129     `PlutusTx.applyCode` PlutusTx.liftCode bsPaper  

130     `PlutusTx.applyCode` PlutusTx.liftCode bsScissors)  

131   $$($PlutusTx.compile [||| wrap ||])  

132 where  

133   wrap = Scripts.wrapValidator @GameDatum @GameRedeemer  

134  

135 gameValidator :: Game -> Validator  

136 gameValidator = Scripts.validatorScript . typedGameValidator  

137  

138 gameAddress :: Game -> Ledger.Address  

139 gameAddress = scriptAddress . gameValidator  

140  

141 gameClient :: Game -> StateMachineClient GameDatum GameRedeemer  

142 gameClient game = mkStateMachineClient $ StateMachineInstance  

                           (gameStateMachine' game) (typedGameValidator game)  

143  

144 data FirstParams = FirstParams  

145   { fpSecond        :: !PaymentPubKeyHash  

146   , fpStake         :: !Integer  

147   , fpPlayDeadline :: !POSIXTime  

148   , fpRevealDeadline :: !POSIXTime  

149   , fpNonce         :: !BuiltinByteString  

150   , fpChoice        :: !GameChoice  

151   } deriving (Show, Generic, FromJSON, ToJSON)  

152  

153 mapError' :: Contract w s SMContractError a -> Contract w s Text a  

154 mapError' = mapError $ pack . show

```

```

155
156 waitUntilTimeHasPassed :: AsContractError e => POSIXTime ->
157                                         Contract w s e ()
158 waitUntilTimeHasPassed t = void $ awaitTime t >> waitNSlots 1
159
160 firstGame :: forall s. FirstParams -> Contract (Last ThreadToken) s Text ()
161 firstGame fp = do
162     pkh <- Contract.ownPaymentPubKeyHash
163     tt <- mapError' getThreadToken
164     let game = Game
165         { gFirst      = pkh
166         , gSecond     = fpSecond fp
167         , gStake      = fpStake fp
168         , gPlayDeadline = fpPlayDeadline fp
169         , gRevealDeadline = fpRevealDeadline fp
170         , gToken       = tt
171     }
172     client = gameClient game
173     v      = lovelaceValueOf (fpStake fp)
174     c      = fpChoice fp
175     x      = case c of
176             Rock    -> bsRock
177             Paper   -> bsPaper
178             Scissors -> bsScissors
179     bs      = sha2_256 $ fpNonce fp `appendByteString` x
180     void $ mapError' $ runInitialise client (GameDatum bs Nothing) v
181     logInfo @String $ "made first move: " ++ show (fpChoice fp)
182     tell $ Last $ Just tt
183
184     waitUntilTimeHasPassed $ fpPlayDeadline fp
185
186     m <- mapError' $ getOnChainState client
187     case m of
188         Nothing    -> throwError "game output not found"
189         Just (o, _) -> case tyTxOutData $ ocsTxOut o of
190             GameDatum _ Nothing -> do
191                 logInfo @String "second player did not play"
192                 void $ mapError' $ runStep client ClaimFirst
193                 logInfo @String "first player reclaimed stake"
194
195             GameDatum _ (Just c') | c `beats` c' || c' == c -> do
196                 logInfo @String "second player played and lost or drew"
197                 void $ mapError' $ runStep client $ Reveal (fpNonce fp) c
198                 logInfo @String "first player revealed and won or drew"

```

```

199
200         _ -> logInfo @String "second player played and won"
201
202 data SecondParams = SecondParams
203     { spFirst      :: !PaymentPubKeyHash
204     , spStake      :: !Integer
205     , spPlayDeadline :: !POSIXTime
206     , spRevealDeadline :: !POSIXTime
207     , spChoice     :: !GameChoice
208     , spToken      :: !ThreadToken
209   } deriving (Show, Generic, FromJSON, ToJSON)
210
211 secondGame :: forall w s. SecondParams -> Contract w s Text ()
212 secondGame sp = do
213     pkh <- Contract.ownPaymentPubKeyHash
214     let game = Game
215         { gFirst       = spFirst sp
216         , gSecond      = pkh
217         , gStake       = spStake sp
218         , gPlayDeadline = spPlayDeadline sp
219         , gRevealDeadline = spRevealDeadline sp
220         , gToken       = spToken sp
221     }
222     client = gameClient game
223     m <- mapError' $ getOnChainState client
224     case m of
225         Nothing    -> logInfo @String "no running game found"
226         Just (o, _) -> case tyTxOutData $ ocsTxOut o of
227             GameDatum _ Nothing -> do
228                 logInfo @String "running game found"
229                 void $ mapError' $ runStep client $ Play $ spChoice sp
230                 logInfo @String $ "made second move: " ++ show (spChoice sp)
231
232                 waitUntilTimeHasPassed $ spRevealDeadline sp
233
234                 m' <- mapError' $ getOnChainState client
235                 case m' of
236                     Nothing -> logInfo @String "first player won or drew"
237                     Just _ -> do
238                         logInfo @String "first player didn't reveal"
239                         void $ mapError' $ runStep client ClaimSecond
240                         logInfo @String "second player won"
241
242             _ -> throwError "unexpected datum"
243

```

```

244 type GameSchema = Endpoint "first" FirstParams ./\
                           Endpoint "second" SecondParams
245
246 endpoints :: Contract (Last ThreadToken) GameSchema Text ()
247 endpoints = awaitPromise (first `select` second) >> endpoints
248   where
249     first  = endpoint @"first" firstGame
250     second = endpoint @"second" secondGame

```

The imports and language pragmas are the same as in the SM example. Also the game type stays the same (1-8). What changes is the game choice parameter and the Eq instance of it. Then we define the *beats* function that takes in two game choices and returns a Bool (15-20). Next we define the game datum and its Eq instance same way as before. For the game redeemer there is a difference in the reveal option where we have to specify now also the game choice since there are 2 possibilities (42-43). One is that the game is a draw and the other one is that we have won. This will also have effect on our transition function where we will have now 5 possible cases. The difference compared to the previous code is in the second case where the 1st player chooses to reveal his choice and we consider again 2 possibilities. The first is that the 1st player has won and the second is that there is a draw in which case we put a constraint that the 2nd player gets his stake back. Now we proceed to the *check* function that takes in three byte strings for our three choices and checks weather the checksum is valid (90-99). The *gameStateMachine* function gets also updated accordingly to our three choices (101-108). Same is true for the game validator and the compiled code. The rest of the on-chain code is the same as in our previous example. In the first game contract when defining parameters what changes is that we define an additional parameter x that gives us the byte string for our choice (174-177) and we use it when computing the hash of the nonce (178). And another change is in the second case after the play deadline when the 2nd player reveals his choice and has lost or drew (195-198). Then for the second game everything stays the same except the log message on line 236 changes were we say »first player won or drew«.

8 Testing

In this chapter we will look at another example for state machines and then look at how Haskells property testing is implemented for Plutus smart contracts.

8.1 State Machine Example: Token Sale

Here we will look at an example where a person wants to make a token sale. The idea is we lock tokens in a contract and allow users to buy them for a fixed price. Figure 35 shows a simple example of the token sale and the code follows below.

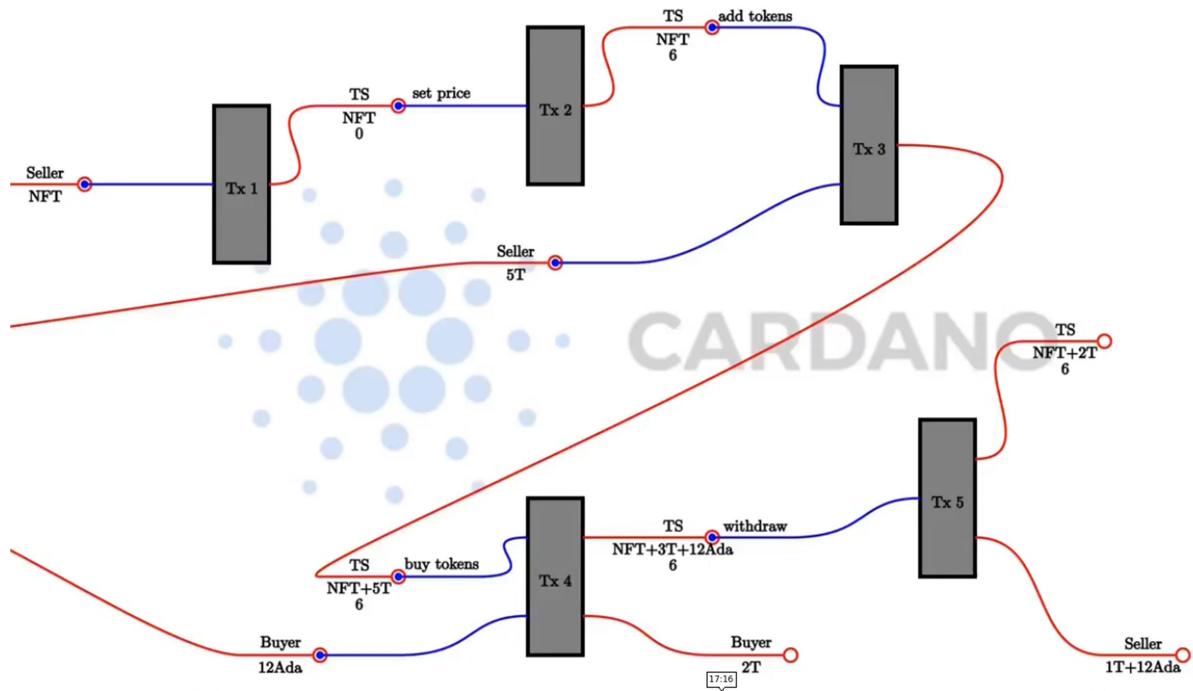


Figure 35 - Token sale

```
1 {-# LANGUAGE DataKinds #-}
2 {-# LANGUAGE DeriveAnyClass #-}
3 {-# LANGUAGE DeriveGeneric #-}
4 {-# LANGUAGE FlexibleContexts #-}
5 {-# LANGUAGE MultiParamTypeClasses #-}
6 {-# LANGUAGE NoImplicitPrelude #-}
7 {-# LANGUAGE OverloadedStrings #-}
8 {-# LANGUAGE ScopedTypeVariables #-}
9 {-# LANGUAGE TemplateHaskell #-}
10 {-# LANGUAGE TypeApplications #-}
11 {-# LANGUAGE TypeFamilies #-}
12 {-# LANGUAGE TypeOperators #-}
```

```

13
14 {-# OPTIONS_GHC -g -fplugin-opt PlutusTx.Plugin:coverage-all #-}
15
16 module Week08.TokenSale
17     ( TokenSale(..)
18     , TSRedeemer(..)
19     , tsCovIdx
20     , TSStartSchema
21     , TSUseSchema
22     , startEndpoint
23     , useEndpoints
24     , useEndpoints'
25     ) where
26
27 import           Control.Monad          hiding (fmap)
28 import           Data.Aeson            (FromJSON, ToJSON)
29 import           Data.Monoid           (Last(..))
30 import           Data.Text             (Text, pack)
31 import           GHC.Generics         (Generic)
32 import           Plutus.Contract      as Contract
33 import           Plutus.Contract.StateMachine
34 import qualified PlutusTx
35 import           PlutusTx.Code        (getCovIdx)
36 import           PlutusTx.Coverage   (CoverageIndex)
37 import           PlutusTx.Prelude    hiding (Semigroup(..), check,
38                                         unless)
39 import           Ledger
40 import           Ledger.Ada           as Ada
41 import           Ledger.Constraints  as Constraints
42 import qualified Ledger.Typed.Scripts as Scripts
43 import           Ledger.Value
44 import           Prelude              (Semigroup(..), Show(..),
45                                         uncurry)
46
47 data TokenSale = TokenSale
48     { tsSeller :: !PaymentPubKeyHash
49     , tsToken  :: !AssetClass
50     , tsTT     :: !ThreadToken
51     } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq)
52
53 PlutusTx.makeLift ''TokenSale
54
55 data TSRedeemer =
56     SetPrice Integer

```

```

56     | AddTokens Integer
57     | BuyTokens Integer
58     | Withdraw Integer Integer
59   deriving (Show, Prelude.Eq)
60
61 PlutusTx.unstableMakeIsData ''TSRedeemer
62
63 {-# INLINABLE lovelaces #-}
64 lovelaces :: Value -> Integer
65 lovelaces = Ada.getLovelace . Ada.fromValue
66
67 {-# INLINABLE transition #-}
68 transition :: TokenSale -> State Integer -> TSRedeemer -> Maybe
69               (TxConstraints Void Void, State Integer)
70 transition ts s r = case (stateValue s, stateData s, r) of
71   (v, _, SetPrice p) | p >= 0 ->
72                     Just ( Constraints.mustBeSignedBy (tsSeller ts)
73                           , State p v )
74
75   (v, p, AddTokens n) | n > 0 ->
76                     Just ( mempty
77                           , State p $ v
78                           <>
79                           assetClassValue (tsToken ts) n )
80
81   (v, p, BuyTokens n) | n > 0 ->
82                     Just ( mempty
83                           , State p $ v
84                           <>
85                           assetClassValue (tsToken ts) (negate n)
86                           <> lovelaceValueOf (n * p) )
87
88   (v, p, Withdraw n l) | n >= 0 && l >= 0 ->
89                     Just ( Constraints.mustBeSignedBy (tsSeller ts)
90                           , State p $ v
91                           <>
92                           assetClassValue (tsToken ts) (negate n)
93                           <> lovelaceValueOf (negate l) )
94
95   _ -> Nothing
96
97 {-# INLINABLE tsStateMachine #-}
98 tsStateMachine :: TokenSale -> StateMachine Integer TSRedeemer
99 tsStateMachine ts = mkStateMachine (Just $ tsTT ts) (transition ts)
100            (const False)

```

```

95
96 {-# INLINABLE mkTSValidator #-}
97 mkTSValidator :: TokenSale -> Integer -> TSRedeemer -> ScriptContext -> Bool
98 mkTSValidator = mkValidator . tsStateMachine
99
100 type TS = StateMachine Integer TSRedeemer
101
102 tsTypedValidator :: TokenSale -> Scripts.TypedValidator TS
103 tsTypedValidator ts = Scripts.mkTypedValidator @TS
104   ($$(PlutusTx.compile [|| mkTSValidator ||])
105   `PlutusTx.applyCode` PlutusTx.liftCode ts)
106   $$$(PlutusTx.compile [|| wrap ||])
107 where
108   wrap = Scripts.wrapValidator @Integer @TSRedeemer
109
110 tsValidator :: TokenSale -> Validator
111 tsValidator = Scripts.validatorScript . tsTypedValidator
112
113 tsAddress :: TokenSale -> Ledger.Address
114 tsAddress = scriptAddress . tsValidator
115
116 tsClient :: TokenSale -> StateMachineClient Integer TSRedeemer
117 tsClient ts = mkStateMachineClient $ StateMachineInstance
118   (tsStateMachine ts) (tsTypedValidator ts)
119
120 tsCovIdx :: CoverageIndex
121 tsCovIdx = getCovIdx $$$(PlutusTx.compile [|| mkTSValidator ||])
122
123 mapErrorSM :: Contract w s SMContractError a -> Contract w s Text a
124 mapErrorSM = mapError $ pack . show
125
126 startTS :: AssetClass -> Contract (Last TokenSale) s Text ()
127 startTS token = do
128   pkh <- Contract.ownPaymentPubKeyHash
129   tt <- mapErrorSM getThreadToken
130   let ts = TokenSale
131     { tsSeller = pkh
132     , tsToken = token
133     , tsTT = tt
134     }
135   client = tsClient ts
136   void $ mapErrorSM $ runInitialise client 0 mempty
137   tell $ Last $ Just ts
138   logInfo $ "started token sale " ++ show ts

```

```

138 setPrice :: TokenSale -> Integer -> Contract w s Text ()
139 setPrice ts p = void $ mapErrorSM $ runStep (tsClient ts) $ SetPrice p
140
141 addTokens :: TokenSale -> Integer -> Contract w s Text ()
142 addTokens ts n = void $ mapErrorSM $ runStep (tsClient ts) $ AddTokens n
143
144 buyTokens :: TokenSale -> Integer -> Contract w s Text ()
145 buyTokens ts n = void $ mapErrorSM $ runStep (tsClient ts) $ BuyTokens n
146
147 withdraw :: TokenSale -> Integer -> Integer -> Contract w s Text ()
148 withdraw ts n l = void $ mapErrorSM $ runStep (tsClient ts) $ Withdraw n l
149
150 type TSStartSchema =
151     Endpoint "start"      (CurrencySymbol, TokenName)
152 type TSUseSchema =
153     Endpoint "set price" Integer
154     .\/ Endpoint "add tokens" Integer
155     .\/ Endpoint "buy tokens" Integer
156     .\/ Endpoint "withdraw" (Integer, Integer)
157
158 startEndpoint :: Contract (Last TokenSale) TSStartSchema Text ()
159 startEndpoint = forever
160             $ handleError logError
161             $ awaitPromise
162             $ endpoint @"start" $ startTS . AssetClass
163
164 useEndpoints' :: ( HasEndpoint "set price" Integer s
165                     , HasEndpoint "add tokens" Integer s
166                     , HasEndpoint "buy tokens" Integer s
167                     , HasEndpoint "withdraw" (Integer, Integer) s
168                     )
169             => TokenSale
170             -> Contract () s Text ()
171 useEndpoints' ts = forever
172             $ handleError logError
173             $ awaitPromise
174             $ setPrice' `select` addTokens' `select` buyTokens'
175                 `select` withdraw'
176
177     where
178         setPrice' = endpoint @"set price" $ setPrice ts
179         addTokens' = endpoint @"add tokens" $ addTokens ts
180         buyTokens' = endpoint @"buy tokens" $ buyTokens ts
181         withdraw' = endpoint @"withdraw"    $ Prelude.uncurry $ withdraw ts
182
183 useEndpoints :: TokenSale -> Contract () TSUseSchema Text ()

```

```
182 useEndpoints = useEndpoints'
```

First we define the TokenSale that will be used for parameterizing our transition function (46-50). It contains the sellers address, the tokens we will sell and the thread token for identifying the UTXO for the token sale. For the redeemer type we provide the operations tha we saw in Figure 35 (54-59). The integers represent price in lovelace and amount of tokens. Then we define a helper function to extract lovelace amount from a value type (63-65). In the transition function we first put in our parameter, then the state that defines the price of the token and the remaining two parameters follow the same patter as in the previous SM example (68). In the case statement we split the state to the value and the datum (69). Next follow the four transitions. In the first case the seller wants to set the price to the new value p (70-71). We only allow that if the price is non-negative. The only constraints we put is that the seller has to sign the transaction and for the state we set now p as the new datum and v as value stays the same. In the second case we add tokens and demand that the amount is positive (73-76). We put no constraints to this action so anybody can add tokens to the contract. For the new state the price does not change but the value changes that are now the old value plus the amount of tokens that are added. Third case is we define buying tokens (78-82). Also here we put no constraints so anybody can buy tokens. The price again does not change but the value is reduced for the amount of tokens that a person bought and the corresponding price in ADA is added. In the last case we define the withdrawal of tokens and lovelace (84-88). Here we insist that the seller has to sign the transaction. And the value decreases for the amount of tokens and lovelace that the seller wants to withdraw. In the end we also define the case that all other transitions are illegal. Now that we have the transition function we define the state machine where we use the smart constructer called *mkStateMachine* (92-94). It takes in 3 arguments. The first is maybe the thread token, then the transition function and the function that determines if states are final or not. And in our case we do not have a final state so we can always return False. After that we can turn our state machine into a validator function (96-98) and compile it (102-107). Then we define the state machine client that is used to interact with the SM in the off-chain code (115-116). What is new is that we define the coverage index (118-119). That is related to getting coverage information for tests. Next we define our helper function that transforms the type of the error from the SM type to text (121-122). This was now the on-chain code.

For the off-chain code we first define the contract function *startTS*. It takes in one parameter and uses the writer functionality. First we look up our own payment public key hash and get the thread token. Then we define the token sale and client parameters (128-133). Next we initialize the client with the initial state 0 and mempty. After that we tell the token sell value so that

other contracts are able to lookup that value and participate in the token sale (135). And in the end we log a message. Then come simple functions corresponding to actions we can take and they are guaranteed to sync with the on-chain code i.e. they create transactions that are valid and will pass validation (138-148). We now have to define 2 schemas. One to start the token sale that will have only one endpoint (150-151) and one to use the token sale that has four endpoints (152-156). For the withdraw action we first put in how many tokens we want to withdraw and then how many lovelace. The start endpoint and use endpoint bundle up the actions we have defined (158-179). The *uncurry* function takes two separate input parameters for a function call and converts them into a pair of integers in a tuple for a function call.

```
uncurry :: (a -> b -> c) -> (a, b) -> c
Prelude> :t uncurry (+)
uncurry (+) :: Num c => (c, c) -> c
```

To try now out this code we can use the emulator trace defined in the test/Spec/Trace.hs file.

```
1  {-# LANGUAGE DataKinds          #-}
2  {-# LANGUAGE FlexibleContexts   #-}
3  {-# LANGUAGE MultiParamTypeClasses #-}
4  {-# LANGUAGE NumericUnderscores #-}
5  {-# LANGUAGE OverloadedStrings   #-}
6  {-# LANGUAGE ScopedTypeVariables #-}
7  {-# LANGUAGE TypeApplications    #-}
8  {-# LANGUAGE TypeFamilies        #-}
9
10 module Spec.Trace
11     ( tests
12     , testCoverage
13     , runMyTrace
14     ) where
15
16 import           Control.Exception          (try)
17 import           Control.Lens
18 import           Control.Monad
19 import           Control.Monad.Freer.Extras
20 import           Data.Default
21 import           Data.IORef
22 import qualified Data.Map                  as Map
23 import           Data.Monoid
24 import           Ledger
25 import           Ledger.Value
26 import           Ledger.Ada
27 import           Plutus.Contract.Test
28 import           Plutus.Contract.Test.Coverage
```



```

69
70  runMyTrace :: IO ()
71  runMyTrace = runEmulatorTraceIO' def emCfg myTrace
72
73  emCfg :: EmulatorConfig
74  emCfg = EmulatorConfig (Left $ Map.fromList [(knownWallet w, v) |
75                                w <- [1 .. 3]]) def def
76    where
77      v :: Value
78      v = Ada.lovelaceValueOf 1_000_000_000 <> assetClassValue token 1000
79
80  currency :: CurrencySymbol
81  currency = "aa"
82
83  name :: TokenName
84  name = "A"
85
86  token :: AssetClass
87  token = AssetClass (currency, name)
88
89  myTrace :: EmulatorTrace ()
90  myTrace = do
91    h <- activateContractWallet w1 startEndpoint
92    callEndpoint @"start" h (currency, name)
93    void $ Emulator.waitNSlots 5
94    Last m <- observableState h
95    case m of
96      Nothing -> Extras.logError @String "error starting token sale"
97      Just ts -> do
98        Extras.logInfo $ "started token sale " ++ show ts
99
100   h1 <- activateContractWallet w1 $ useEndpoints ts
101   h2 <- activateContractWallet w2 $ useEndpoints ts
102   h3 <- activateContractWallet w3 $ useEndpoints ts
103
104   callEndpoint @"set price" h1 1_000_000
105   void $ Emulator.waitNSlots 5
106
107   callEndpoint @"add tokens" h1 100
108   void $ Emulator.waitNSlots 5
109
110   callEndpoint @"buy tokens" h2 20
111   void $ Emulator.waitNSlots 5
112
113   callEndpoint @"buy tokens" h3 5

```

```

113     void $ Emulator.waitNSlots 5
114
115         callEndpoint @"withdraw" h1 (40, 10_000_000)
116     void $ Emulator.waitNSlots 5
117
118     checkPredicateOptionsCoverage :: CheckOptions
119             -> String
120             -> CoverageRef
121             -> TracePredicate
122             -> EmulatorTrace ()
123             -> TestTree
124     checkPredicateOptionsCoverage options nm (CoverageRef ioref)
125         predicate action =
126             HUnit.testCaseSteps nm $ \step -> do
127                 checkPredicateInner options predicate action step
128                 (HUnit.assertBool nm) (\rep -> modifyIORef ioref (rep<>))

```

We run the trace with a custom emulator configuration *emCfg* (73-77). It gives an initial distribution of ADA and native tokens with the token name “A” and currency symbol “aa”. Now for the trace first we activate the start endpoint for wallet one (90). This is the wallet that will run the token sale. Then we call the start endpoint and wait for 5 slots (91-92) to give to the SM to start. Then we check the observable state that is the token sale value and in the case that it’s nothing we log an error message. Else we start the use endpoints that are parameterized by the *ts* value. Then we call various endpoints. First the wallet one sets the price of the token (103). Then wallet one adds 100 tokens (106). Next wallet two buys 20 tokens (109) and wallet three buys 5 tokens (112). In the end wallet one makes a withdraw of 40 tokens and 10 ADA.

If you want to test this code that resides in the Spec/ folder you need to use the commands:

```

$ cabal repl plutus-pioneer-program-week08:test:plutus-pioneer-program-week08-tests
Prelude> :l test/Spec/Trace.hs
Prelude Spec.Trace> runMyTrace

```

When we get the results what’s worth noting is that wallet one will have around 1008 ADA, which is because 2 ADA had to be put to the script containing the SM contract. This is handled automatically when we construct the transaction in the start endpoint call.

8.2 Automatic testing using emulator traces

Plutus uses the Tasty test framework. You can find the tasty package with a description and example on hackage.haskell.org. Tests are of type *TestTree*. There is special support for tests in Plutus in the module *Plutus.Contract.Test*. There are various types of tests that are supported

but we will look at two of those types: one that works with emulator traces and one that uses property based testing. By using the `checkPredicate` function we produce something that the tasty framework can understand. There is also a variation of that function called `checkPredicateOptions` that takes in an additional parameter of type `CheckOptions` (Figure 37). It has no constructors exposed but we can use various functions with it (Figure 38).

checkPredicate	
<code>:: String</code>	Descriptive name of the test
<code>-> TracePredicate</code>	The predicate to check
<code>-> EmulatorTrace ()</code>	
<code>-> TestTree</code>	
	Check if the emulator trace meets the condition

Figure 36 - Check predicate function

checkPredicateOptions	
<code>:: CheckOptions</code>	Options to use
<code>-> String</code>	Descriptive name of the test
<code>-> TracePredicate</code>	The predicate to check
<code>-> EmulatorTrace ()</code>	
<code>-> TestTree</code>	
	A version of <code>checkPredicate</code> with configurable CheckOptions

Figure 37 - Check predicate options

The `Lens`' type is related to Optics in Haskell. We can use it to set an emulator config. To modify the `CheckOptions` without using lens we can use the `changeInitialWalletValue` function. Given a wallet and a function that updates the initial value it modifies the check options where the given wallet has the function applied to it. Going back to the `checkPredicate` function let's look at the `TracePredicate` type (Figure 39). It is a condition on a trace that represents the actual test. We see that there are also logical combinators so given a trace we can negate it or make a logical »and« or »or«. There is a wide variety of available predicates in the Assertion chapter of the `Plutus.Contract.Test` module. We will use the `walletFundsChange` predicate (Figure 40).

```

data CheckOptions

| Options for running the

defaultCheckOptions :: CheckOptions

minLogLevel :: Lens' CheckOptions LogLevel

emulatorConfig :: Lens' CheckOptions EmulatorConfig

changeInitialWalletValue :: Wallet -> (Value -> Value) -> CheckOptions -> CheckOptions

| Modify the value assigned to the given wallet in the initial distribution.

```

Figure 38 - Check options type

```

type TracePredicate = FoldM (Eff '[Reader InitialDistribution, Error EmulatorFoldErr,
Writer (Doc Void)]) EmulatorEvent Bool
# Source

```

Figure 39 - Trace predicate

```
walletFundsChange :: Wallet -> Value -> TracePredicate
```

| Check that the funds in the wallet have changed by the given amount, excluding fees.

Figure 40 - Wallet funds change

The *walletFundsChange* checks for a given wallet after the trace is completed, that the wallet funds will have changed by this given value excluding fees. There is also a variation of this function called *walletFundsExactChange* where this fee is not automatically taken care of. If we now look again at the *test/Spec/Trace.hs* file from the previous chapter in the *tests* parameter we are using the *checkPredicateOptions* function (38-43) where we input my options that contains the emulator config (61-62). We start with the default option and add the emulator config. The operator “*~*” is part of Haskell optics and says that we set the *emulatorConfig* part to the given value. When defining *myPredicate* we use the logical “and” to combine three predicates and we use the *walletFundsChange* function (64-68). The fees are taken care of automatically when we use this function but not the minimal ADA deposit that we have to take care of. If we want to run the test we have to first get into our test repl as shown at the end of the previous sub-chapter. Then we execute the command:

```

ghci> :l test/Spec/Trace.hs
Ok, one module loaded.
ghci> import Test.Tasty
ghci> defaultMain tests
All 1 tests passed (0.34s)

```

If the test does not pass we would get a longer log message with an error output.

8.3 Test Coverage

Plutus makes use of code coverage that enables the user to see how much code is covered by your tests. The *checkPredicateCoverage* function uses the *CoverageRef* variable (Figure 41).

checkPredicateCoverage	
:: String	Descriptive name of the test
-> CoverageRef	
-> TracePredicate	The predicate to check
-> EmulatorTrace ()	
-> TestTree	

Figure 41 - Check predicate coverage

If we look again at the code in *test/Spec/Trace.hs* we have the *checkPredicateOptionsCoverage* function (118-126). Basically we looked at the implementation of *checkPredicateOptions* and *checkPredicateCoverage* functions and then combine the two. The *CoverageRef* type is just a newtype wrapper around a *IORef* for *CoverageReport* (Figure 42). And we can get a coverage reference with the *newCoverageRef* function.

newtype CoverageRef	
Constructors	
CoverageRef (IORef CoverageReport)	
newCoverageRef :: IO CoverageRef	
readCoverageRef :: CoverageRef -> IO CoverageReport	

Figure 42 - Coverage reference

So using that we define the `testCoverage` IO action (45-59). First we get the coverage reference, then we use the `checkPredicateOptionsCoverage` function but because we are in IO we need the `defaultMain` function for this to work which always throws a default code exception at the end even if all test pass. With the `try` statement we are catching this exception. Then we provide the parameters as previously but with the new reference. In the body of the case statement we should always get the `Left` constructor and if the test succeeds it should be a zero return code and if it doesn't it should be a non-zero code. But in either case we read the report and write it to the `TokenSaleTrace` HTML file by using the coverage index. The `tsCovIdx` is defined in the previous `TokenSale.hs` file. We can run now the test coverage from the Repl.

```
Prelude Spec.Trace> testCoverage
```

If we open the HTML file in a web-browser the lines highlighted green represent conditions that passed for all tests and the dark highlighted lines represent conditions that were never met.

8.4 Interlude optics

The most used Haskell library for optics is called Lens. Optics are used to reach deeply into hierarchical data types to manipulate parts of them. Let's look at the example code from `Week08/Lens.hs` where we will look at a problem and show how to solve it with lens.

```
1 {-# LANGUAGE TemplateHaskell #-}
2
3 module Week08.Lens where
4
5 import Control.Lens
6
7 newtype Company = Company {_staff :: [Person]} deriving Show
8
9
10 data Person = Person
11   { _name    :: String
12   , _address :: Address
13   } deriving Show
14
15 newtype Address = Address {_city :: String} deriving Show
16
17 alejandro, lars :: Person
18 alejandro = Person
19   { _name    = "Alejandro"
20   , _address = Address {_city = "Zacateca"}
21 }
```

```

22 lars = Person
23   { _name    = "Lars"
24   , _address = Address {_city = "Regensburg"}
25   }
26
27 iohk :: Company
28 iohk = Company { _staff = [alejandro, lars] }
29
30 goTo :: String -> Company -> Company
31 goTo there c = c { _staff = map movePerson (_staff c)}
32   where
33     movePerson p = p { _address = (_address p) {_city = there}}
34
35 makeLenses ''Company
36 makeLenses ''Person
37 makeLenses ''Address
38
39 goTo' :: String -> Company -> Company
40 goTo' there c = c & staff . each . address . city .~ there

```

The company data type is just a newtype wrapper around a list of persons with the accessor called staff (7). Person is a record type with fields name and address (10-13). Address is a newtype wrapper around a string with the accessor called city (15). As an example we define two persons (17-25) and a company where the staff consist of these two persons (27-28). The function *goTo* takes in a city name as a string and a company and changes for all the staff members the address field (30-33). For this example we use the record update syntax. Dealing with nested record types can become quite messy because you always have to keep the old fields in place and update the new ones. And this is what optics is trying to solve by providing first class field accessors. You could say that optics provide a programmable dot, which is similar to other languages as Java where you can access an attribute with the dot notation. The lenses library provides some template Haskell features but it expects some underscore conventions which we used in our accessor names. You need to provide the types for which you want to have lenses (35-37). And the name of the lenses will be the names of the original fields without the underscore. There is a way to inspect what code the template Haskell writes at compile time. You can activate the flag »:set -ddump-splices«. If you reload your Lens module in the Repl you will get an extended output for the template Haskell definitions. Here is a short demonstration of how to use lenses in the Repl.

```

ghci> :l src/Week08/Lens.hs
ghci> import Control.Lens

```

```

ghci> lars ^. name
"Lars"
ghci> lars ^. address
Address {_city = "Regensburg"}
ghci> lars ^. address . city
"Regensburg"
ghci> lars & name .~ "LARS"
Person {_name = "LARS", _address = Address {_city = "Regensburg"}}
ghci> lars & address . city .~ "Munich"
Person {_name = "Lars", _address = Address {_city = "Munich"}}

```

The combination of lenses is done with the dot notation. The ampersand notation is used to set a record type field. There is also a different type of optics called traversals which does not only zoom into one field but into many simultaneously. If you would have a list it would zoom into all elements. Here is an example using the *each* traversable.

```

ghci> [1 :: Int, 3, 4] & each .~ 42
[42, 42, 42]

```

Various types of lenses can be combined with the dot operator. We can see such an example in our code with the *goTo'* function (39-40).

8.5 Property based testing with QuickCheck

QuickCheck is a Haskell library for property based testing. In contrast to Unit testing where the cases of test are hardcoded we only specify a property function which defines a property of our code. It takes in a given variable and returns True if our code upholds the property for the given variable. Then variables are randomly picked by the QuickCheck library and the test passes if for all variables the property function returns True. Let's look at the code in *QuickCheck.hs*.

```

1  module Week08.QuickCheck where
2
3  prop_simple :: Bool
4  prop_simple = 2 + 2 == (4 :: Int)
5
6  -- Insertion sort code:
7
8  -- | Sort a list of integers in ascending order.
9
10 -- >>> sort [5,1,9]
11 -- [1,5,9]
12 --
13 sort :: [Int] -> [Int] -- not correct
14 sort []      = []
15 sort (x:xs) = insert x xs

```

```

16
17 -- | Insert an integer at the right position into an /ascendingly sorted/
18 -- list of integers.
19 --
20 -- >>> insert 5 [1,9]
21 -- [1,5,9]
22 --
23 insert :: Int -> [Int] -> [Int] -- not correct
24 insert x [] = [x]
25 insert x (y:ys) | x <= y = x : ys
26 | otherwise = y : insert x ys
27
28 isSorted :: [Int] -> Bool
29 isSorted [] = True
30 isSorted [_] = True
31 isSorted (x : y : ys) = x <= y && isSorted (y : ys)
32
33 prop_sort_sorts :: [Int] -> Bool
34 prop_sort_sorts xs = isSorted $ sort xs
35
36 prop_sort_preserves_length :: [Int] -> Bool
37 prop_sort_preserves_length xs = length (sort xs) == length xs

```

First we define the *sort* and the helper *insert* functions that sort a list (13-26). Next we define the *isSorted* function that checks whether a list of integers is sorted (28-31). With the function *prop_sort_sorts* we test the sorting property of the *sort* function (33-34) and with the function *prop_sort_preserves_length* we test that the *sort* function preserves the length of the list (36-37). We can run our example code with the following command:

```

Prelude Week08.QuickCheck> import Test.QuickCheck
Prelude Week08.QuickCheck> quickCheck prop_sort_sorts
*** Failed! Falsified (after 8 tests and 4 shrinks):
[0,0,-1]
ghci> sort [0,0,-1]
[0,-1]

```

What the message means quick check tried out 8 examples of a list and after it found one it tried 4 times to further simplify it and returned it as an example in the Repl.



When QuickCheck checks a property it starts with simple random arguments and then makes them more and more complex over time.

By default quick check tries 100 arguments to test our property. For our sort function we could implement a fix as specify line 15 as follows:

```
sort (x:xs) = insert x $ sort xs
```

If we run this code and test it we see that a 100 test pass, but if we manually try the previous list it does get sorted but a zero value disappears from the list.

```
ghci> sort [0,0,-1]
[-1,0]
```

So it means our test is only good enough as good the *prop_sort_sorts* function is. But we can detect this with another property called *prop_sort_preserves_length* (36-37). And for this function we do get a counter example where the test now fails. And the bug is on line 25 that we can change to the following code where both test will pass:

```
insert x (y:ys) | x <= y      = x : y : ys
```

8.6 Property based testing of Plutus contracts

The problem we encounter when writing property based tests for Plutus contracts is how do you test code that has effect on the real world as for example our blockchain. We can explain this on the example of testing file operation with QuickCheck. The idea is you start with a model which is an idealized system. There must be some sort of relation between the real system and the model. And then what quick check does it generates a random sequence of actions so for the file system it would generate for example opening, reading and writing sequences. You can apply an action to your model and apply it to the real world. So both progress to a new state and after that you compare the two and check whether they are still in sync.

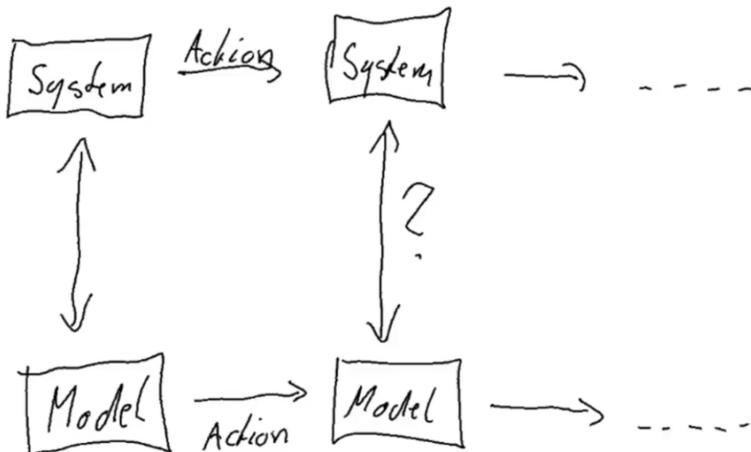


Figure 43 - Testing schema

And shrinking in this case would be if you have a list of actions you could skip some and try if the test still fails. This is how property testing also works in Plutus. We will look at the example of our token sale contract. The code for this is in *test/Spec/Model.hs*.

```
1  {-# LANGUAGE DataKinds          #-}
2  {-# LANGUAGE DeriveAnyClass     #-}
3  {-# LANGUAGE DeriveGeneric      #-}
4  {-# LANGUAGE FlexibleContexts   #-}
5  {-# LANGUAGE FlexibleInstances   #-}
6  {-# LANGUAGE GADTs              #-}
7  {-# LANGUAGE InstanceSigs       #-}
8  {-# LANGUAGE MultiParamTypeClasses #-}
9  {-# LANGUAGE NumericUnderscores #-}
10 {-# LANGUAGE OverloadedStrings  #-}
11 {-# LANGUAGE RankNTypes         #-}
12 {-# LANGUAGE ScopedTypeVariables #-}
13 {-# LANGUAGE StandaloneDeriving #-}
14 {-# LANGUAGE TemplateHaskell    #-}
15 {-# LANGUAGE TypeApplications   #-}
16 {-# LANGUAGE TypeFamilies       #-}
17 {-# LANGUAGE TypeOperators      #-}

18
19 module Spec.Model
20   ( tests
21   , test
22   , TSModel(..)
23   ) where

24
25 import           Control.Lens
26 import           Control.Monad
27 import           Data.Map
28 import qualified Data.Map
29 import           Data.Maybe
30 import           Data.Monoid
31 import           Data.String
32 import           Data.Text
33 import           Plutus.Contract
34 import           Plutus.Contract.Test
35 import           Plutus.Contract.Test.ContractModel
36 import           Plutus.Contract.Test.ContractModel.Symbolics
37 import           Plutus.Trace.Emulator
```

```

38 import           Ledger
39 import           Ledger.Ada
40 import           Ledger.Value
41 import           Test.QuickCheck
42 import           Test.Tasty
43 import           Test.Tasty.QuickCheck
44
45 import           Week08.TokenSaleFixed          (TokenSale
46                                        (..), TSStartSchema, TSUseSchema, startEndpoint, useEndpoints')
47 type TSUseSchema' = TSUseSchema .\` Endpoint "init" TokenSale
48
49 useEndpoints' :: Contract () TSUseSchema' Text ()
50 useEndpoints' = awaitPromise $ endpoint @"init" useEndpoints'
51
52 data TSState = TSState
53   { _tssPrice    :: !Integer
54   , _tssLovelace :: !Integer
55   , _tssToken    :: !Integer
56   } deriving Show
57
58 makeLenses ''TSState
59
60 newtype TSModel = TSModel {_tsModel :: Map Wallet TSState}
61   deriving Show
62
63 makeLenses ''TSModel
64
65 tests :: TestTree
66 tests = testProperty "token sale model" prop_TS
67
68 instance ContractModel TSModel where
69
70   data Action TSModel =
71     Start Wallet
72     | SetPrice Wallet Wallet Integer
73     | AddTokens Wallet Wallet Integer
74     | Withdraw Wallet Wallet Integer Integer
75     | BuyTokens Wallet Wallet Integer
76   deriving (Show, Eq)
77
78   data ContractInstanceKey TSModel w s e p where
79     StartKey :: Wallet -> ContractInstanceKey TSModel
79                                         (Last TokenSale) TSStartSchema Text ()

```

```

80      UseKey    :: Wallet -> Wallet -> ContractInstanceKey
81                      TSModel () TSUseSchema' Text ()
82
83      instanceWallet :: ContractInstanceKey TSModel w s e p -> Wallet
84      instanceWallet (StartKey w) = w
85      instanceWallet (UseKey _ w) = w
86
87      instanceTag :: SchemaConstraints w s e => ContractInstanceKey
88                      TSModel w s e p ->
89                      ContractInstanceTag
90      instanceTag key = fromString $ "instance tag for: " ++ show key
91
92      arbitraryAction :: ModelState TSModel -> Gen (Action TSModel)
93      arbitraryAction _ = oneof
94          [ Start      <$> genWallet
95            , SetPrice   <$> genWallet <*> genWallet <*> genNonNeg
96            , AddTokens  <$> genWallet <*> genWallet <*> genNonNeg
97            , BuyTokens   <$> genWallet <*> genWallet <*> genNonNeg
98            , Withdraw    <$> genWallet <*> genWallet <*> genNonNeg <*> genNonNeg
99          ]
100
101     initialState :: TSModel
102     initialState = TSModel Map.empty
103
104     initialInstances :: [StartContract TSModel]
105     initialInstances =      [StartContract (StartKey v) () | v <- wallets]
106                               ++
107                               [StartContract (UseKey v w) () | v <- wallets,
108                                w <- wallets]
109
110     precondition :: ModelState TSModel -> Action TSModel -> Bool
111     precondition s (Start w)           = isNothing $ getTSState' s w
112     precondition s (SetPrice v _)     = isJust    $ getTSState' s v
113     precondition s (AddTokens v _)    = isJust    $ getTSState' s v
114     precondition s (BuyTokens v _)    = isJust    $ getTSState' s v
115     precondition s (Withdraw v _)    = isJust    $ getTSState' s v
116
117     nextState :: Action TSModel -> Spec TSModel ()
118     nextState (Start w) = do
119         wait 3
120         (tsModel . at w) $= Just (TSState 0 0 0)
121         withdraw w $ Ada.toValue minAdaTxOut
122
123     nextState (SetPrice v w p) = do
124         wait 3
125         when (v == w) $
126             (tsModel . ix v . tssPrice) $= p

```

```

121 nextState (AddTokens v w n) = do
122     wait 3
123     started <- hasStarted v
124     -- has the token sale started?
125     when (n > 0 && started) $ do
126         bc <- actualValPart <$> askModelState (view $ balanceChange w)
127         let token = tokens Map.! v
128         when (tokenAmt + assetClassValueOf bc token >= n) $ do
129             -- does the wallet have the tokens to give?
130             withdraw w $ assetClassValue token n
131             (tsModel . ix v . tssToken) $~ (+ n)
132     nextState (BuyTokens v w n) = do
133         wait 3
134         when (n > 0) $ do
135             m <- getTSSState v
136             case m of
137                 Just t
138                     | t ^. tssToken >= n -> do
139                         let p = t ^. tssPrice
140                         l = p * n
141                         withdraw w $ lovelaceValueOf l
142                         deposit w $ assetClassValue (tokens Map.! v) n
143                         (tsModel . ix v . tssLovelace) $~ (+ 1)
144                         (tsModel . ix v . tssToken) $~ (+ (- n))
145                         _ -> return ()
146     nextState (Withdraw v w n l) = do
147         wait 3
148         when (v == w) $ do
149             m <- getTSSState v
150             case m of
151                 Just t
152                     | t ^. tssToken >= n && t ^. tssLovelace >= l -> do
153                         deposit w $ lovelaceValueOf l <>
154                         assetClassValue (tokens Map.! w) n
155                         (tsModel . ix v . tssLovelace) $~ (+ (- l))
156                         (tsModel . ix v . tssToken) $~ (+ (- n))
157                         _ -> return ()
158
159     startInstances :: ModelState TSModel -> Action TSModel ->
160             [StartContract TSModel]
161     startInstances _ _ = []
162
163     instanceContract :: (SymToken -> AssetClass) -> ContractInstanceKey
164             TSModel w s e p -> p -> Contract w s e ()
165     instanceContract _ (StartKey _) () = startEndpoint

```

```

161     instanceContract _ (UseKey _) () = useEndpoints''
162
163     perform :: HandleFun TSModel -> (SymToken -> AssetClass) -> ModelState
164         TSModel -> Action TSModel -> SpecificationEmulatorTrace ()
165     perform h _ m (Start v)      = do
166         let handle = h $ StartKey v
167         withWait m $ callEndpoint @"start" handle
168             (tokenCurrencies Map.! v, tokenNames Map.! v)
169         Last mts <- observableState handle
170         case mts of
171             Nothing -> Trace.throwError $ GenericError $
172                 "starting token sale for wallet " ++ show v ++ " failed"
173             Just ts -> forM_ wallets $ \w ->
174                 callEndpoint @"init" (h $ UseKey v w) ts
175     perform h _ m (SetPrice v w p) = withWait m $ callEndpoint
176                     @"set price" (h $ UseKey v w) p
177     perform h _ m (AddTokens v w n) = withWait m $ callEndpoint
178                     @"add tokens" (h $ UseKey v w) n
179     perform h _ m (BuyTokens v w n) = withWait m $ callEndpoint
180                     @"buy tokens" (h $ UseKey v w) n
181     perform h _ m (Withdraw v w n l) = withWait m $ callEndpoint
182                     @"withdraw" (h $ UseKey v w) (n, l)
183
184     withWait :: ModelState TSModel -> SpecificationEmulatorTrace () ->
185         SpecificationEmulatorTrace ()
186     withWait m c = void $ c >> waitUntilSlot ((m ^. Test.currentSlot) + 3)
187
188     deriving instance Eq (ContractInstanceKey TSModel w s e p)
189     deriving instance Show (ContractInstanceKey TSModel w s e p)
190
191     getTSSState' :: ModelState TSModel -> Wallet -> Maybe TSSState
192     getTSSState' s v = s ^. contractState . tsModel . at v
193
194     getTSSState :: Wallet -> Spec TSModel (Maybe TSSState)
195     getTSSState v = do
196         s <- getModelState
197         return $ getTSSState' s v
198
199     hasStarted :: Wallet -> Spec TSModel Bool
200     hasStarted v = isJust <$> getTSSState v
201
202     wallets :: [Wallet]
203     wallets = [w1, w2]
204
205     tokenCurrencies :: Map Wallet CurrencySymbol

```

```

198 tokenCurrencies = Map.fromList $ zip wallets ["aa", "bb"]
199
200 tokenNames :: Map Wallet TokenName
201 tokenNames = Map.fromList $ zip wallets ["A", "B"]
202
203 tokens :: Map Wallet AssetClass
204 tokens = Map.fromList [(w, AssetClass (tokenCurrencies Map.! w,
205                         tokenNames Map.! w)) | w <- wallets]
206
207 genWallet :: Gen Wallet
208 genWallet = elements wallets
209
210 genNonNeg :: Gen Integer
211 genNonNeg = getNonNegative <$> arbitrary
212
213 tokenAmt :: Integer
214 tokenAmt = 1_000
215
216 prop_TS :: Actions TSModel -> Property
217 prop_TS = withMaxSuccess 10 . propRunActionsWithOptions
218     (defaultCheckOptions & emulatorConfig . initialChainState .~ Left d)
219     defaultCoverageOptions
220     (const $ pure True)
221 where
222
223     d :: InitialDistribution
224     d = Map.fromList $ [ (w
225         , lovelaceValueOf 1_000_000_000 <>
226             mconcat [assetClassValue t tokenAmt |
227                 t <- Map.elems tokens])
228             | w <- wallets
229     ]
230
231 test :: IO ()
232 test = quickCheck prop_TS

```

In the import section we import the *Plutus.Contract.Test*, *Test.QuickCheck*, *Test.Tasty* and *Test.Tasty.QuickCheck* modules. The last one provides a link between the quick check and tasty libraries. We define the *TSUseSchema*' schema which adds a new init endpoint to our previous schema that takes the argument of type *TokenSale* (47). The idea is that once we start our token sale we somehow must be able to communicate the *TokenSale* parameter to the use contract. The init endpoint simply calls the *useEndpoints*' function with the value provided in the endpoint (49-50). The reason we can add an endpoint to our existing schema is that in the

TokenSaleFixed.hs file that we import in the beginning we define *useEndpoints'* in such a way that it works for every schema that contains the four endpoints stated in the type signature. The only thing the *useEndpoints''* can do is to call the init endpoint and expose the *TokenSale* parameter. Then we define the token sale state data type that supposed to represent the state of one token sale instance (52-56). It has three fields: the current price, the current supply of lovelace and current supply of tokens in the contract UTXO. Now we define our model that we presented in Figure 43 (60-61). That is just a Map from wallet to token sale state. The idea is we will have two wallets and each of the wallet will run their token sale and the wallets will trade different tokens. We implement lenses for that (58, 63). All the logic is now in the instance of the type class contract model for the *TSModel* type. Here we provide how our model should behave and is linked to the actual contract. First we have a so called associated data type (70-76). It is an associated action type which represents the actions that quick check will generate. We have one constructor for each of the endpoints and have different arguments to keep track of the wallets that are in play. There are two Wallet parameters where the first one represents the wallet that started a given contract and the second one wants to make the given action. This should work only if the wallets are owned by one owner with the exception of the buy tokens action. Then we define another associated data type called contract instance key (78-80). The idea is for each instance of the contract that we are running we want a key that identifies this instance. Here instead of just providing the constructors we write them in the form of just providing the type signatures. This is called generalized algebraic data type (GADT). GADT allow us to have different type parameters for the cunstroctors and we need this because our contracts can have different type parameters. For the *UseKey* type the first wallet is the one running the token sale and the second wallet is the one interacting with it. There are two signatures because we had a start and a use contract. The next function *instanceWallet* tells the system how to extract the wallet that a given contract is running on by its constraint instance key (82-84). Then we implement the *instanceTag* method that takes one of these keys and turns it into a contract instance tag which implements the *isString* type class (86-87). And the contract instance tags are then used on the blockchain site, on the actual implementation site to identify running instances of contracts and interact with them by calling endpoints. We use this type class when we switch on the overloaded string extension. For the contract instance key we are also deriving Show and because we have a GADT we have to put the name of the type after the Show keyword (181). So it is important that this instance tag function results in a different tag for each instance that we will ever run in our simulation or in our tests. We will have one start instance for each wallet and then one use instance for each pair of wallets. The *arbitraryAction* function is supposed to generate an arbitrary action (89-96). »oneof« is one of the combinators provided by quick check and given a list of arbitrary actions it picks one of those. The *genWallet* function generates a random wallet from wallet 1 or wallet 2. We

combine the `genWallet` then through the `fmap` operator with several wallet actions. The `genNonNeg` function generates a non negative integer. Next comes `initialState` which is the initial state of our model that contains just an empty map which means no token sale has started yet (98-99). The `initialInstances` function gives the initial contract instances that have to run (101-103). In the body of the function we have contracts that are supposed to run in the beginning. With them we want to start the token sale for each wallet and then the use contract for each pair of wallets. The precondition function allows us to say that some actions may not be legal in a given model state (105-110). So for example we can say that we can only start a token sale if it hasn't already started. The precondition on the start should be it hasn't started yet and for other actions it should be that it has started. During the quick check shrinking procedure that drops some actions, this is also checked that all the preconditions are still satisfied. As input we use the model state type that has no constructors exposed (**Error! Reference source not found.**).

```
data ModelState state
```

The `ModelState` models the state of the blockchain. It contains,

- the contract-specific state (`contractState`)
- the current slot (`currentSlot`)
- the wallet balances (`balances`)
- the amount that has been minted (`minted`)

Figure 44 - Model state

But it contains a couple of functions that operate on the model state. The most important one is the `contractState` function which is a lens from model state to state (Figure 45).

```
contractState :: forall state state. Lens (ModelState state) (ModelState state)
```

```
state state
```

Source

Lens for the contract-specific part of the model state.

Spec monad update functions: `$=` and `$~`.

Figure 45 - Contract state function

There is also a current slot optic that is a getter which means we can only look at it but cannot set it. With lenses getting and setting is possible. The `getTSState'` function given a model state and a wallet it tries to extract the token sale that this wallet is running (183-184). With the `at` keyword we can look up a value given a Map key and if the key does not exist we will get a

Nothing value. If we would set a given key to Nothing we would remove it from the map. When we explained our testing principle in the beginning we said for each action we must know what effect that action will have on our model. And that's what the *nextState* function is for (112-154). It takes an action and results in a Spec monad that has the purpose of describing effects on our TS model. In this model there is a concept of how much every wallet owns. And in the Spec monad you can say that a given wallet earns some funds or sends them elsewhere. And you can say that time passes in between. The model and the actual simulator must keep in sync as far as slots are concerned. The first do block says when we do the start for the token sale then in the model for key w (our wallet) there now will be a *TSState* and the value of it will be 0 0. The \$= operator comes from the Spec monad and on the left side it takes a lens. And then it will update the corresponding focus of the lens to that new value. And the last statement is the effect of the funds on the wallet. We say the wallet w in question loses funds. That's because when we start the token sale we need to put down the minimal ADA for the token sale UTXO. Similar we define other states where we deal with setting the price, adding tokens, buying tokens and make a withdrawal. In the set price action we check weather the wallet that started the token sale is the wallet that tries to set the price and if yes we set the price to the given value (117-120). For the add tokens action we first ask weather the token sale has started. If yes then we construct a function from model state to value change with help of the *balanceChange* getter function and apply the *askModelState* function to it (Figure 46).

```
askModelState :: GetModelState m => (ModelState (StateType m) -> a) -> m a
```

| Get a component of the model state.

Figure 46 - Ask model state

Then we get that value in the spec monad which is not a Plutus value. It's something more complicated something of type symbolic value. We can extract the normal value with the *actualValPart* function so what the line 125 does is the bc is now the actual balance change of my wallet w. The purpose why we are doing this is we want to check that we actually have enough tokens to give. And if we do we make a withdraw and update the TS model where we add the tokens. This dollar tilde operator in line 129 is similar to the dollar equal, so the dollar equal sets a value and the dollar tilde applies a function to the value. For the buy tokens action if the amount we want to buy is positive we first get the token sale state and we just consider the case that the token sale is actually running. If yes we check that there are at least n tokens available, then we look up the price and calucalte how much it will cost us in lovelace. Then we make a withdraw from our wallet for this amount of lovelace and with the deposit function we gain funds to our wallet which will be the tokens that we bought. In the end we update our

model with the lovelace and token amount values. The last action is withdraw. This is only allowed for the wallet that created the token sale (146). The code is very similar to the previous action. First we check again that the token sale is still running and that there are sufficient funds available to withdraw. If yes we make a deposit to our wallet and we update our model again with the lovelace and token amount values we will withdraw.

Then we have the *startInstances* function (156-157). There is the possibility to start our contract instances later so given a model statement action you can give a list of contracts that are supposed to be started. But because we start them all in the beginning we put here an empty list. Now we start providing the link between our model and the actual emulator or the blockchain and the first part of this is done by the *instanceContract* function (159-161). It takes in a function that we don't use, a key and a parameter value which in our case is always unit. And then we have to say which contract corresponds to that. If the key is a start key it's the start endpoint contract and if the key is a use key it's the use endpoint contract. And finally the *perform* function tells us how an action is actually expressed in an actual action in the emulator on the blockchain (163-175). In the end we get the specification emulator trace monad that you can think of as an emulator trace. In the cases following after the first one we use the *withWait* function that should ensure that something takes exactly three slots (177-178). It takes in a model and an action and performs it and then waits until three slots have passed from the start of the action. And the actions are calls to various endpoints with the key parameters that correspond to the actions. This provides the links between the model and the actions with actual operations that are supposed to happen on the blockchain or in the emulator. The start action is a bit trickier because of the mechanism that when we start a token sale we have to call the init endpoint which is now happening in this do block. The start contract actually writes the token sale into this observable state with tell which is a Maybe value (167). And if the value is something we call the init endpoint for the use contracts and we loop over all our wallets. Calling this init endpoint will actually kick off the the original use contract. To tie everything together we can use now the *propRunActionsWithOptions* function (Figure 47).

propRunActionsWithOptions	
<code>:: ContractModel state</code>	
<code>=> CheckOptions</code>	Emulator options
<code>-> CoverageOptions</code>	Coverage options
<code>-> (ModelState state -> TracePredicate)</code>	Predicate to check at the end
<code>-> Actions state</code>	The actions to run
<code>-> Property</code>	

Figure 47 - Properties run action with options

It takes check options which allow us for example to specify a fund distribution in our wallets. Then it takes coverage options but we will not use this in our example. Then it takes an additional predicate to check at the end. With the final model state you can provide a trace predicate and then that will also be tested but we won't use this either. Then it returns something of action state to property. We use this function in the *prop_TS* function (215-227). We are composing it with the *withMaxSuccess* function that comes from quick check and we can specify how many test cases to run. We start with the default check option and add our emulator config and initial chain state and then set it to an initial distribution where every wallet should get a thousand ADA and a thousand tokens. Wallet 1 will be selling token »A« and wallet two will be selling token »B«. The *prop_TS* function returns a property check which you can think of as a Bool used by quick check. Because we saw before we can randomly generate action sequences quick check can handle our *prop_TS* function since it takes in the *Actions* type. And the property that is actually checked is that what we say in the next state function what happens to our model that is actually reflected when we perform the actions on the blockchain in our emulator traces. So this will check that what we say should happen in the model actually does happen on the blockchain. And in the Repl we can then run our test function that calls our property test function (229-230). When we run this in the Repl we also get some statistics. We see which actions were actually tried and how many of them were rejected due to preconditions. If we would import the *TokenSale* module instead of *TokenSaleFixed* in the beginning we would get a test failure due to the fact that we could try to withdraw some ADA from the min ADA amount. Let's look now how we can grab all of this into a test suite.

```

1  module Main
2    ( main
3    ) where
4
5  import qualified Spec.Model
6  import qualified Spec.Trace
7  import           Test.Tasty
8
9  main :: IO ()
10 main = defaultMain tests
11
12 tests :: TestTree
13 tests = testGroup "token sale"
14   [ Spec.Trace.tests
15   , Spec.Model.tests
16   ]

```

This represents now the main program of our test suite. `tests` is our test tree where we just put these two tests that we wrote. And we can run the tests with the »*cabal test*« command. What we have to note is that when testing with quick check we provide the off-chain code but a user could write their own off-chain code that for example would try to steal the funds. And such scenarios are then not covered by our testing procedures. The quick check tests only checks the flow of funds whether that agrees at each point with what we specified in the model but it is possible to add additional checks and it is also possible to influence these action sequences. It's also possible to do some more unit test like scenarios where we specify certain flows of actions to steer the tests into certain directions. That's called dynamic logic.

8.7 Homework

For homework we will modify the token sale contract such that it accepts an additional transition called `close` that can be called only by the seller to close the UTXO and collect all the remaining tokens, lovelace and the NFT. Let's look at the `Week08 TokenNameWithClose.hs` code.

```

1  data TokenSale = TokenSale
2    { tsSeller :: !PaymentPubKeyHash
3    , tsToken :: !AssetClass
4    , tsTT    :: !ThreadToken
5    } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq)
6
7  PlutusTx.makeLift ''TokenSale
8
9  data TSRedeemer =
10   SetPrice Integer
11   | AddTokens Integer
12   | BuyTokens Integer
13   | Withdraw Integer Integer
14   | Close
15   deriving (Show, Prelude.Eq)
16
17 PlutusTx.unstableMakeIsData ''TSRedeemer
18
19 {-# INLINABLE lovelaces #-}
20 lovelaces :: Value -> Integer
21 lovelaces = Ada.getLovelace . Ada.fromValue
22
23 {-# INLINABLE transition #-}
24 transition :: TokenSale -> State (Maybe Integer) -> TSRedeemer -> Maybe
25           (TxConstraints Void Void, State (Maybe Integer))
25 transition ts s r = case (stateValue s, stateData s, r) of

```

```

26     (v, Just _, SetPrice p)    | p >= 0           ->
27     Just ( Constraints.mustBeSignedBy (tsSeller ts)
28           , State (Just p) v )
29
29     (v, Just p, AddTokens n)   | n > 0           ->
30     Just ( mempty
31           , State (Just p) $
32             v <>
33             assetClassValue (tsToken ts) n )
33
34     (v, Just p, BuyTokens n)   | n > 0           ->
35     Just ( mempty
36           , State (Just p) $
37             v <>
38             assetClassValue (tsToken ts) (negate n) <>
39             lovelaceValueOf (n * p) )
39
40     (v, Just p, Withdraw n l) | n >= 0 && l >= 0 &&
41       v `geq` (w <> toValue minAdaTxOut) ->
42     Just ( Constraints.mustBeSignedBy (tsSeller ts)
43           , State (Just p) $
44             v <>
45             negate w )
45
46     where
47       w = assetClassValue (tsToken ts) n <>
48         lovelaceValueOf l
49     (_ , Just _, Close)           ->
50       Just ( Constraints.mustBeSignedBy (tsSeller ts)
51           , State Nothing mempty )
52
52     _ -> Nothing
53
54 {-# INLINABLE tsStateMachine #-}
55 tsStateMachine :: TokenSale -> StateMachine (Maybe Integer) TSRedeemer
56 tsStateMachine ts = mkStateMachine (Just $ tsTT ts)
57                                         (transition ts) isNothing
57
58 {-# INLINABLE mkTSValidator #-}
59 mkTSValidator :: TokenSale -> Maybe Integer -> TSRedeemer ->
60                                         ScriptContext -> Bool
60 mkTSValidator = mkValidator . tsStateMachine
61
62 type TS = StateMachine (Maybe Integer) TSRedeemer
63

```

```

64  tsTypedValidator :: TokenSale -> Scripts.TypedValidator TS
65  tsTypedValidator ts = Scripts.mkTypedValidator @TS
66    ($$($PlutusTx.compile [||| mkTSValidator |||])
67     `PlutusTx.applyCode` PlutusTx.liftCode ts)
68    $$($PlutusTx.compile [||| wrap |||])
69  where
70    wrap = Scripts.wrapValidator @(Maybe Integer) @TSRedeemer
71
72  tsValidator :: TokenSale -> Validator
73  tsValidator = Scripts.validatorScript . tsTypedValidator
74
75  tsAddress :: TokenSale -> Ledger.Address
76  tsAddress = scriptAddress . tsValidator
77
78  tsClient :: TokenSale -> StateMachineClient (Maybe Integer) TSRedeemer
79  tsClient ts = mkStateMachineClient $ StateMachineInstance
80    (tsStateMachine ts) (tsTypedValidator ts)
81
82  mapErrorSM :: Contract w s SMContractError a -> Contract w s Text a
83  mapErrorSM = mapError $ pack . show
84
85  startTS :: AssetClass -> Contract (Last TokenSale) s Text ()
86  startTS token = do
87    pkh <- Contract.ownPaymentPubKeyHash
88    tt <- mapErrorSM getThreadToken
89    let ts = TokenSale
90      { tsSeller = pkh
91      , tsToken = token
92      , tsTT = tt
93      }
94    client = tsClient ts
95    void $ mapErrorSM $ runInitialise client (Just 0) mempty
96    tell $ Last $ Just ts
97    logInfo $ "started token sale " ++ show ts
98
99  setPrice :: TokenSale -> Integer -> Contract w s Text ()
100 setPrice ts p = void $ mapErrorSM $ runStep (tsClient ts) $ SetPrice p
101
102 addTokens :: TokenSale -> Integer -> Contract w s Text ()
103 addTokens ts n = void (mapErrorSM $ runStep (tsClient ts) $ AddTokens n)
104
105 buyTokens :: TokenSale -> Integer -> Contract w s Text ()
106 buyTokens ts n = void $ mapErrorSM $ runStep (tsClient ts) $ BuyTokens n
107
108 withdraw :: TokenSale -> Integer -> Integer -> Contract w s Text ()

```

```

107 withdraw ts n l = void $ mapErrorSM $ runStep (tsClient ts) $ Withdraw n l
108
109 close :: TokenSale -> Contract w s Text ()
110 close ts = void $ mapErrorSM $ runStep (tsClient ts) Close
111
112 type TSStartSchema =
113     Endpoint "start"      (CurrencySymbol, TokenName)
114 type TSUseSchema =
115     Endpoint "set price" Integer
116     .\|/ Endpoint "add tokens" Integer
117     .\|/ Endpoint "buy tokens" Integer
118     .\|/ Endpoint "withdraw"  (Integer, Integer)
119     .\|/ Endpoint "close"    ()
120
121 startEndpoint :: Contract (Last TokenSale) TSStartSchema Text ()
122 startEndpoint = forever
123             $ handleError logError
124             $ awaitPromise
125             $ endpoint @"start" $ startTS . AssetClass
126
127 useEndpoints' :: ( HasEndpoint "set price" Integer s
128                     , HasEndpoint "add tokens" Integer s
129                     , HasEndpoint "buy tokens" Integer s
130                     , HasEndpoint "withdraw" (Integer, Integer) s
131                     , HasEndpoint "close" () s
132                     )
133             => TokenSale
134             -> Promise () s Text ()
135 useEndpoints' ts = setPrice' `select` addTokens' `select` buyTokens'
136                         `select` withdraw' `select` close'
137     where
138         setPrice' = endpoint @"set price" $ \p      ->
139                     handleError logError (setPrice ts p)
140         addTokens' = endpoint @"add tokens" $ \n      ->
141                     handleError logError (addTokens ts n)
142         buyTokens' = endpoint @"buy tokens" $ \n      ->
143                     handleError logError (buyTokens ts n)
144         withdraw' = endpoint @"withdraw"   $ \(n, l) ->
145                     handleError logError (withdraw ts n l)
146         close'    = endpoint @"close"      $ \()      ->
147                     handleError logError (close ts)
148
149 useEndpoints :: TokenSale -> Contract () TSUseSchema Text ()
150 useEndpoints = forever . awaitPromise . useEndpoints'

```

In the *TSRedeemer* data type we define our Close action (9-15). Then in the transition function our state parameter takes in a Maybe integer. And at the end of the body we add the case of close which closes the token sale (49-50). The type signatures for state machine, validator, TS type and ts client get updated accordingly. Also the wrap helper function takes in the maybe integer type. When we call the *runInitialise* function in the startTS contract we have to provide now a Just 0 value. In lines 109-110 we define our close contract. And in the *TSUseSchema* we add the close endpoint. Similar we do in the type signature of the *useEndpoints'* function and in its body we add the close endpoint call.

9 Resources

[1] The Plutus Pioneer Programm git repository

<https://github.com/input-output-hk/plutus-pioneer-program>

[2] The Cardano documentation

<https://docs.cardano.org/plutus/learn-about-plutus>

<https://docs.cardano.org/plutus/eutxo-explainer>

[3] The Extended UTXO Ledger Model paper

<https://hydra.iohk.io/build/12982960/download/1/extended-utxo-specification.pdf>

[4] The Plutus apps repository

<https://github.com/input-output-hk/plutus-apps>

[5] Plutus application backend blog post

<https://iohk.io/en/blog/posts/2021/10/28/plutus-application-backend-pab-supporting-dapp-development-on-cardano/>