3. Lecture Three

> "My brain just exploded."
> — ehird

1. Introduction

Within this set of lecture notes, the following will be discussed. Firstly, memory issues associated with the local 'Playground' server (where our Haskell smart contracts are being ran) are addressed. Secondly, a reintroduction to the extended unspent transaction output model (EUTxO) is provided. Furthermore, minting tokens, certifying and (stake-related) rewarding is discussed. This is followed by a description of the parameters that validator scripts are provided. Then on-chain and off-chain code is briefly discussed. Finally, time intervals, examples and homework's are presented.

2. Playground Memory Issues

You are now able to modify the timeout in the Plutus Playground Server (located within the Plutus Playground Client folder) to any number of minutes. This is accomplished by running the following command:

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

Note that you can modify the value of 120s to any amount, this will set the timeout in XXX seconds `XXXs` .

---

3. Quick Refresh On The EUTxO Model

EUTxO stands for **Extended Unspent Transaction Output Model**.

Let me just bring you back up to speed on UTxO...

This model is similar to how BitCoin (BTC) manages unspent digital assets held by any given wallet (*these assets can be thought of as money...* in BTCs case, these assets would be BitCoin). However, with BTC, the model is simply built through very simple UTxOs (unextended).

Given a simple UTxO, there is a **redeemer** and a **validator**. In order to spend any unspent transaction output, the redeemer can be thought of as a cryptographic key which is passed to the validator. Once any given UTxO is attempted to be spent, the validator is then responsible for verifying the chain of ownership by using the redeemer to verify that the UTxO belongs to whichever wallet is attempting to create a new UTxO.

With an EUTxO model, you have a script at a given address, a redeemer, context and datum... Validation must still occur in order to spend any given EUTxO. However, the script address is contains a reference to a set of instructions. The instructions **(the script)** contains arbitrary logic, some of which is responsible for creating a validator (plus a possible set of numerous constraints under which the validator may or may not verify the chain of ownership of any given EUTxO).

Furthermore, the EUTxO model has the ability to facilitate ownership of multiple types of asset (for example non fungible tokens can exist within a EUTxO model. Even the ownership of other coins can be transferred from one wallet to another given the EUTxO model).

3.1 EUTxO Scripts

EUTxO scripts are held at a script address. During lecture two we saw a low level implementation of a validator within a script where all three arguments were defined as the Haskell type: `Data`:

```
mkValidator :: Data -> Data -> Data -> ()
mkValidator _ _ _ = ()
validator :: Validator
validator = mkValidatorScript $\$(PlutusTx.compile [|| mkValidator ||]
)
```

In practice this is **not used**. We instead use the typed version. This is where data and redeemer can be custom types, as long as they implement the `IsData` type class. The third argument (the Context) is must be of type `ScriptContext`.

In the examples we have seen so far, we've only been examining the data and the redeemer. We have never given much thought to the context, but the context is of course very important. It can support a given state, allowing us to create types of state machines within

The `Context` is of type: `ScriptContext` and can be found within:

```
plutus-ledger-api
```

This is a package that, until now, we have not needed. But now, we do need it, so it has been included within the cabal.project file for the third week. The Context can be found within:

```
Plutus.V1.Ledger.Contexts
```

The `ScriptContext` is a record type with two fields: `TxInfo` and `ScriptPurpose`. The `ScriptPurpose` is defined within the same module and it (rather obviously) describes for which purpose the script is run. For example, to spend, to certify, to reward or to mint (an NFT for example), etc.

The most important purpose for us is the `Spending TxOutRef`, this is what we have mostly talked about within the Extended UTxO model.

This is when a script is run to validate spending input for a transaction.

3.2 Other Important Purposes

Minting: This is important for when you want to define a native token. For example, the ScriptPurpose may use the Minting constructor to create a native token which describes under which condition the token may be minted or burned.

Rewarding: Related to staking (in some manner, yet to be explained).

Certifying: Related to certificates, like delegation certification.

For now, we are concentrating on the spending purpose.

3.3 Context - TxInfo (Acronym for Transaction Info) — Possibily Outdated

The TxInfo Data Type describes the transaction[1], which is to say it has fields for:

- txInfoInputs :: **[TxInInfo]** — ALL the inputs of the transaction.
- txInfoOutputs :: **[TxOut]** — ALL the outputs of the transaction.
- txInfoFee :: **Value** — The fee paid to consume the transaction.
- TxInfoForge :: **value** — Either the number of newly created (forged) Native Tokens [2] or if negative, the amount of newly burned Native Tokens.
- txInfoDCert :: ***[TxInfoDCert]*** — List of certificates, such as delegation certificates. ***(Possibly Deprecated)***
- txInfoWdrl :: **[(StakingCredential, Integer)]** — Staking Reward Withdrawal ***(Possibly Deprecated)***
- txInfoValidRange :: **POSIXTimeRange** — Time range in which the transaction is valid. ***(Possibly Modified)***
- txInfoSignatories :: [PubKeyHash] — List of public keys that have signed the transaction.
- txInfoData :: [(DatumHash, Datum)] — The output value of the Tx which have been

spent (required), Lars uses the phrase:

```
Spending Transactions Have To Include The Datum Of The Transactions Th
ey've Spent (The Script Output).

Producing transactions that have spent any given output only have to i
nclude the hash (Which hash exactly?) - dictionary of 'datum-hash' to
hash to full datum values to a given hash?

However, Producing Transactions Can Optionally Do That.
```

- txInfoId :: TxId — Hash of the pending transaction (excluding witnesses)

3.4 Context - TxInfo (Current)

- txInfoInputs :: [TxInInfo] — Transaction inputs
- txInfoOutputs :: [TxOutInfo] — Transaction outputs
- txInfoFee :: Value — The fee paid by this transaction.
- txInfoForge :: Value — The Value forged by this transaction.
- txInfoValidRange :: SlotRange — The valid range for the transaction.
- txInfoForgeScripts :: [MonetaryPolicyHash]
- txInfoSignatories :: [PubKeyHash] — Signatures provided with the transaction
- txInfoData :: [(DatumHash, Datum)]
- txInfoId :: TxId — Hash of the pending transaction (excluding witnesses)

3.5 On-Chain VS Off-Chain Validation

One of the nice things about Cardano is that validation can be tested and verified as spendable off-chain. However, due to time — well latency [2][3], there is always a chance that a given UTxO that you — well, the wallet trying to consume said UTxO — has already been spent by another transaction on-chain, causing a reversal. You don't, however loose any funds.

Furthermore, there is always a chance that the time that the off-chain code is executed and the contract is found to be valid, if the txValidRange *(time)* fell inside the time of validation off-chain but outside the time of validation on-chain, the validation will fail **on-chain** and the contract will not be executed. However, if the time does fall into the txValidRange, then the contract is completely deterministic. The only non-deterministic property of these contracts is the time at which it may or may not be executed, as time is continuous and impossible to test for. When writing scripts and testing, it is likely the case that you will have to assume that the function for checking if the time is valid or not is TRUE. You may also write cases where it is FALSE to be exhaustive, but you know (in this case) the script will simply fail and the UTxO will not be consumed.

By default, if this parameter is not set manually, the time slot / POSIXTIME is set to 'infinite', and as such - this initial check will always pass.

In short: "The trick is" to do a time check before attempting to validate the transaction (initially off-chain), then when you attempt to spend some UTxO on-chain, ideally it'll validate, but there is always the possibility (due to time latency) that validation is not possible. So, there is time on one hand and determinism on the other, essentially.

IMPORTANT: The consensus algorithm uses SLOT time, not POSIXTIME; which is probably why the current TxInfo uses SlotRange, rather than POSIXTimeRange.

As it stands in lecture three, we're still using POSIXTime, apparently it is easy to move between POSIXTime and Slot 'Time'. This is simply a slight complication between Plutus and Ouroboros. We know that if a parameter change occurs, we will always know 36 hours in the future (we'll know if there is a hard fork or changes to be made within 36 hours).

Note: it would appear now that POSIXTime has changed to SlotRange for the reasoning Lars outlines about time? General observation, may be wrong.

3.5.1 General Checks (Summary)

- All the inputs are present.
- The balances add up.
- The fees are included.
- The 'time-range' is checked (node checks the current time and compares it to the time-range specified by the transaction. If the current time does not fall into this time range, then then validation fails immediately, without ever running the validator scripts.

However, if the time-range does fall into this interval, then validation is completely deterministic again. This is just a static piece of data attached to the transaction. Thus, the result of validation **does not** depend on when it is run.

- By default, all transactions use an infinite time interval.

3.6 Time Intervals

Specifying a time interval in Haskell can be done in the following manner (using one of many types of constructors, which can be found within the documentation) [4] as is outlined using the repl:

```
> cabal repl
...
> import Plutus.V1.Ledger.Interval
-- Interval from a to b
```

```
> interval (10 :: Integer) 20
Interval {ivFrom = LowerBound (Finite 10) True, ivTo = UpperBound (Fin
ite 20) True}
> member 9 $ interval (10 :: Integer) 20
False
> member 10 $ interval (10 :: Integer) 20
True
> member 11 $ interval (10 :: Integer) 20
True
> member 20 $ interval (10 :: Integer) 20
True
> member 21 $ interval (10 :: Integer) 20
False
-- Interval from 30 to +
> 21 $ from (30 :: Integer)
False
> 30 $ from (30 :: Integer)
True
> 30000 $ from (30 :: Integer)
True
-- Interval to 30 from -
> 30000 $ to (30 :: Integer)
False
> 31 $ to (30 :: Integer)
False
> 30 $ to (30 :: Integer)
True
> 7 $ to (30 :: Integer)
True
-- Intersection
> intersection (interval (10 :: Integer) 20) $ interval 18 30
Interval {ivFrom = LowerBound (Finite 18) True, ivTo = UpperBound (Fin
ite 20) True}
> contains (to (100 :: Integer)) $ interval 30 80
True
-- This means the interval between 30 and 80 is fully contained within
 -inf to 100
> contains (to (100 :: Integer)) $ interval 30 100
True
> contains (to (100 :: Integer)) $ interval 30 101
False
> overlaps (to (100 :: Integer)) $ interval 30 101
True
-- Because the interval does in fact overlap the range -inf to 100, ho
wever
> overlaps (to (100 :: Integer)) $ interval 101 110
False
-- as there is zero overlap
```

Note: this stuff takes a long time to digest and make notes on...

4. Finally Some Interesting Stuff!

Now that we understand the basics of smart contracts on Cardano, we understand the notion of a redeemer, of datum and of context (at least, we kind of do[2]). Time to implement some interesting stuff baby!

4.1 Example: Releasing ADA to a 'beneficiary' Sometime In The Future

```haskell
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE DeriveAnyClass      #-}
{-# LANGUAGE DeriveGeneric       #-}
{-# LANGUAGE FlexibleContexts    #-}
{-# LANGUAGE NoImplicitPrelude   #-}
{-# LANGUAGE OverloadedStrings   #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TemplateHaskell     #-}
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE TypeOperators       #-}

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Week03.Vesting where

import           Control.Monad        hiding (fmap)
import           Data.Aeson           (ToJSON, FromJSON)
import           Data.Map             as Map
import           Data.Text            (Text)
import           Data.Void            (Void)
import           GHC.Generics         (Generic)
import           Plutus.Contract
import           PlutusTx             (Data (..))
import qualified PlutusTx
import           PlutusTx.Prelude     hiding (Semigroup(..), unless)
import           Ledger               hiding (singleton)
import           Ledger.Constraints   as Constraints
import qualified Ledger.Typed.Scripts as Scripts
import           Ledger.Ada           as Ada
import           Playground.Contract  (printJson, printSchemas, ensure
KnownCurrencies, stage, ToSchema)
import           Playground.TH        (mkKnownCurrencies, mkSchemaDefi
nitions)
import           Playground.Types     (KnownCurrency (..))
import           Prelude              (IO, Semigroup (..), Show (..),
String)
import           Text.Printf          (printf)
```

Implementing VestingDatum as a record with the following fields: A beneficiary, who will receive an amount to be provided given: the UTxO may be signed by the beneficiary and the deadline has been surpassed.

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

PlutusTx.unstableMakeIsData ''VestingDatum

{-# INLINABLE mkValidator #-}
```

J.D Our validator will take as its datum argument a parameter `VestingDatum`. The redeemer is simply left as type: unit Data, The context parameter is `ScriptContext`.

We then define the validator with shorthand arguments: `dat`, `()`, and `ctx` provide the exception if false: "beneficiary's signature missing" AND (shouldn't this be or?) "deadlined not reached" where we assign `ctx` all the properties of the transaction info (TxInfo).

We also define the two constraints: signedByBeneficiary (boolean) = TxInfo.txSignedBy (who has this UTxO been signed by?) AND it must equate to the PubKeyHash provided within the Datum...

Furthermore, the deadlineReached boolean uses a a convention as described in §3.5.1 (essentially: the valid transaction time range must be contained within the deadline as defined by the datum).

This is then essentially all wrapped up and compiled down into Plutus-core.

```
mkValidator :: VestingDatum -> () -> ScriptContext -> Bool
mkValidator dat () ctx = traceIfFalse "beneficiary's signature missing
" signedByBeneficiary &&
                         traceIfFalse "deadline not reached" deadlineRea
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    signedByBeneficiary :: Bool
    signedByBeneficiary = txSignedBy info $ beneficiary dat

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

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

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

validator :: Validator
validator = Scripts.validatorScript typedValidator

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

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

Now for the wallet logic? Which is off-chain! So, Lars tells me not to worry! **(QUITE YET!)**. Thus, I won't!

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

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

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

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

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

```
mkSchemaDefinitions ''VestingSchema

mkKnownCurrencies []
```

4.2 Parameterised Contracts

Up until this point we have always compiled our 'on-chain' validators using `typedValidator` (a compile-time constant). However, you may define a 'family of scripts' that are parametrised such that you obtain a set of different typedValidator(s).

We can do this by prepending an additional parameter to our validator template:

```
mkValidator validator :: AnyNameParam -> () -> () -> ScriptContext ->
bool
```

For our purposes, we'll use the same script as shown in §4.1:

```


{-# LANGUAGE DataKinds                  #-}
{-# LANGUAGE DeriveAnyClass         #-}
{-# LANGUAGE DeriveGeneric              #-}
{-# LANGUAGE FlexibleContexts           #-}
{-# LANGUAGE MultiParamTypeClasses  #-}
{-# LANGUAGE NoImplicitPrelude      #-}
{-# LANGUAGE OverloadedStrings      #-}
{-# LANGUAGE ScopedTypeVariables        #-}
{-# LANGUAGE TemplateHaskell            #-}
{-# LANGUAGE TypeApplications           #-}
{-# LANGUAGE TypeFamilies               #-}
{-# LANGUAGE TypeOperators              #-}

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Week03.Vesting where

import           Control.Monad        hiding (fmap)
import           Data.Aeson           (ToJSON, FromJSON)
import           Data.Map             as Map
import           Data.Text            (Text)
import           Data.Void            (Void)
import           GHC.Generics         (Generic)
import           Plutus.Contract
import           PlutusTx             (Data (..))
import qualified PlutusTx
import           PlutusTx.Prelude     hiding (Semigroup(..), unless)
import           Ledger               hiding (singleton)
import           Ledger.Constraints   as Constraints
```

```haskell
import qualified Ledger.Typed.Scripts as Scripts
import           Ledger.Ada           as Ada
import           Playground.Contract  (printJson, printSchemas, ensure
KnownCurrencies, stage, ToSchema)
import           Playground.TH        (mkKnownCurrencies, mkSchemaDefi
nitions)
import           Playground.Types     (KnownCurrency (..))
import           Prelude              (IO, Semigroup (..), Show (..),
String)
import           Text.Printf          (printf)

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

PlutusTx.makeLift ''VestingParam

{-# INLINABLE mkValidator #-}
mkValidator :: VestingParam -> () -> () -> ScriptContext -> Bool
mkValidator p () () ctx = traceIfFalse "beneficiary's signature missin
g" signedByBeneficiary &&
                          traceIfFalse "deadline not reached" deadlineRea
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    signedByBeneficiary :: Bool
    signedByBeneficiary = txSignedBy info $ beneficiary p

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

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

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

validator :: VestingParam -> Validator
validator p = Scripts.validatorScript $ typedValidator p
```

```haskell
valHash :: VestingParam -> Ledger.ValidatorHash
valHash p = Scripts.validatorHash $ typedValidator p

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

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

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

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

grab :: forall w s e. AsContractError e => POSIXTime -> Contract w s e
 ()
grab d = do
    now   <- currentTime
    pkh   <- pubKeyHash <$> ownPubKey
    if now < d
        then logInfo @String $ "too early"
        else do
            let p = VestingParam
                        { beneficiary = pkh
                        , deadline    = d
                        }
            utxos <- utxoAt $ scrAddress p
            if Map.null utxos
                then logInfo @String $ "no gifts available"
                else do
                    let orefs   = fst <$> Map.toList utxos
                        lookups = Constraints.unspentOutputs utxos
```

```
                                    Constraints.otherScript (validator p)
                          tx :: TxConstraints Void Void
                          tx      = mconcat [mustSpendScriptOutput oref $ I
                                    mustValidateIn (from now)
                      ledgerTx <- submitTxConstraintsWith @Void lookups tx
                      void $ awaitTxConfirmed $ txId ledgerTx
                      logInfo @String $ "collected gifts"

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

mkSchemaDefinitions ''VestingSchema

mkKnownCurrencies []
```

Adding a parameter or parameters is not as easy as it may have originally seemed for the following reasons:

- Parameters are not known at compile-time.
- The template Haskell used to compile down to Plutus via Plutus-tx cannot accept parameters at run-time.
- This means we have to use something called liftCode in order to pass individual parameters through as compiled parameters and essentially inject them into the compiled code.
- There are also a bunch of syntax changes that are required, simply because we're essentially injecting an additional parameter, from what I understand.

5. Homework 1.0

THIS HAS BEEN A LOT TO TAKE IN, IN ONE SHORT SITTING... Solution One...

```
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE DeriveAnyClass      #-}
{-# LANGUAGE DeriveGeneric       #-}
{-# LANGUAGE FlexibleContexts    #-}
{-# LANGUAGE NoImplicitPrelude   #-}
{-# LANGUAGE OverloadedStrings   #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TemplateHaskell     #-}
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE TypeOperators       #-}
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Week03.Solution1 where

import           Control.Monad        hiding (fmap)
import           Data.Aeson           (ToJSON, FromJSON)
import           Data.Map             as Map
import           Data.Text            (Text)
import           Data.Void            (Void)
import           GHC.Generics         (Generic)
import           Plutus.Contract
import qualified PlutusTx
import           PlutusTx.Prelude     hiding (unless)
import           Ledger               hiding (singleton)
import           Ledger.Constraints   as Constraints
import qualified Ledger.Typed.Scripts as Scripts
import           Ledger.Ada           as Ada
import           Playground.Contract  (printJson, printSchemas, ensure
KnownCurrencies, stage, ToSchema)
import           Playground.TH        (mkKnownCurrencies, mkSchemaDefi
nitions)
import           Playground.Types     (KnownCurrency (..))
import           Prelude              (IO, Show (..), String)
import qualified Prelude              as P
import           Text.Printf          (printf)

data VestingDatum = VestingDatum
    { beneficiary1 :: PubKeyHash
    , beneficiary2 :: PubKeyHash
    , deadline     :: POSIXTime
    } deriving Show

PlutusTx.unstableMakeIsData ''VestingDatum

{-# INLINABLE mkValidator #-}
mkValidator :: VestingDatum -> () -> ScriptContext -> Bool
mkValidator dat () ctx
    | (beneficiary1 dat `elem` sigs) && (to       (deadline dat) `contai
    | (beneficiary2 dat `elem` sigs) && (from (1 + deadline dat) `contai
    | otherwise
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    sigs :: [PubKeyHash]
    sigs = txInfoSignatories info

    range :: POSIXTimeRange
```

```
    range = txInfoValidRange info

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

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

validator :: Validator
validator = Scripts.validatorScript typedValidator

scrAddress :: Ledger.Address
scrAddress = scriptAddress validator

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

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

give :: AsContractError e => GiveParams -> Contract w s e ()
give gp = do
    pkh <- pubKeyHash <$> ownPubKey
    let dat = VestingDatum
                { beneficiary1 = gpBeneficiary gp
                , beneficiary2 = pkh
                , deadline     = gpDeadline gp
                }
        tx  = mustPayToTheScript dat $ Ada.lovelaceValueOf $ gpAmount gp
    ledgerTx <- submitTxConstraints typedValidator tx
    void $ awaitTxConfirmed $ txId ledgerTx
    logInfo @String $ printf "made a gift of %d lovelace to %s with deadl
        (gpAmount gp)
        (show $ gpBeneficiary gp)
        (show $ gpDeadline gp)

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

```
    now    <- currentTime
    pkh    <- pubKeyHash <$> ownPubKey
    utxos  <- utxoAt scrAddress
    let utxos1 = Map.filter (isSuitable $ \dat -> beneficiary1 dat == pk
        utxos2 = Map.filter (isSuitable $ \dat -> beneficiary2 dat == pk
    logInfo @String $ printf "found %d gift(s) to grab" (Map.size utxos1
    unless (Map.null utxos1) $ do
        let orefs   = fst <$> Map.toList utxos1
            lookups = Constraints.unspentOutputs utxos1 P.<>
                      Constraints.otherScript validator
            tx :: TxConstraints Void Void
            tx      = mconcat [mustSpendScriptOutput oref $ Redeemer $ P
                      mustValidateIn (to now)
        void $ submitTxConstraintsWith @Void lookups tx
    unless (Map.null utxos2) $ do
        let orefs   = fst <$> Map.toList utxos2
            lookups = Constraints.unspentOutputs utxos2 P.<>
                      Constraints.otherScript validator
            tx :: TxConstraints Void Void
            tx      = mconcat [mustSpendScriptOutput oref $ Redeemer $ P
                      mustValidateIn (from now)
        void $ submitTxConstraintsWith @Void lookups tx
  where
    isSuitable :: (VestingDatum -> Bool) -> TxOutTx -> Bool
    isSuitable p o = case txOutDatumHash $ txOutTxOut o of
        Nothing -> False
        Just h  -> case Map.lookup h $ txData $ txOutTxTx o of
            Nothing        -> False
            Just (Datum e) -> maybe False p $ PlutusTx.fromData e

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

mkSchemaDefinitions ''VestingSchema

mkKnownCurrencies []
```

6. Homework 2.0

THIS HAS BEEN A LOT TO TAKE IN, IN ONE SHORT SITTING... Solution Two...

```
{-# LANGUAGE DataKinds                 #-}
{-# LANGUAGE DeriveAnyClass            #-}
{-# LANGUAGE DeriveGeneric             #-}
```

```haskell
{-# LANGUAGE FlexibleContexts      #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoImplicitPrelude     #-}
{-# LANGUAGE OverloadedStrings     #-}
{-# LANGUAGE ScopedTypeVariables   #-}
{-# LANGUAGE TemplateHaskell       #-}
{-# LANGUAGE TypeApplications      #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE TypeOperators         #-}

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Week03.Solution2 where

import           Control.Monad        hiding (fmap)
import           Data.Aeson           (ToJSON, FromJSON)
import           Data.Map             as Map
import           Data.Text            (Text)
import           Data.Void            (Void)
import           GHC.Generics         (Generic)
import           Plutus.Contract
import qualified PlutusTx
import           PlutusTx.Prelude     hiding (Semigroup(..), unless)
import           Ledger               hiding (singleton)
import           Ledger.Constraints   as Constraints
import qualified Ledger.Typed.Scripts as Scripts
import           Ledger.Ada           as Ada
import           Playground.Contract  (printJson, printSchemas, ensure
KnownCurrencies, stage, ToSchema)
import           Playground.TH        (mkKnownCurrencies, mkSchemaDefi
nitions)
import           Playground.Types     (KnownCurrency (..))
import           Prelude              (IO, Semigroup (..), Show (..),
String)
import           Text.Printf          (printf)

{-# INLINABLE mkValidator #-}
mkValidator :: PubKeyHash -> POSIXTime -> () -> ScriptContext -> Bool
mkValidator pkh s () ctx =
    traceIfFalse "beneficiary's signature missing" checkSig      &&
    traceIfFalse "deadline not reached"            checkDeadline
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    checkSig :: Bool
    checkSig = pkh `elem` txInfoSignatories info
```

```
    checkDeadline :: Bool
    checkDeadline = from s `contains` txInfoValidRange info

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

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

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

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

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

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

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

grab :: forall w s e. AsContractError e => Contract w s e ()
grab = do
    now   <- currentTime
    pkh   <- pubKeyHash <$> ownPubKey
    utxos <- Map.filter (isSuitable now) <$> utxoAt (scrAddress pkh)
```

```
    if Map.null utxos
        then logInfo @String $ "no gifts available"
        else do
            let orefs   = fst <$> Map.toList utxos
                lookups = Constraints.unspentOutputs utxos        <>
                          Constraints.otherScript (validator pkh)
                tx :: TxConstraints Void Void
                tx      = mconcat [mustSpendScriptOutput oref $ Redeemer
                          mustValidateIn (from now)
            ledgerTx <- submitTxConstraintsWith @Void lookups tx
            void $ awaitTxConfirmed $ txId ledgerTx
            logInfo @String $ "collected gifts"
  where
    isSuitable :: POSIXTime -> TxOutTx -> Bool
    isSuitable now o = case txOutDatumHash $ txOutTxOut o of
        Nothing -> False
        Just h  -> case Map.lookup h $ txData $ txOutTxTx o of
            Nothing        -> False
            Just (Datum e) -> case PlutusTx.fromData e of
                Nothing -> False
                Just d  -> d <= now

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

mkSchemaDefinitions ''VestingSchema

mkKnownCurrencies []
```

7. Summary

To summarise what has been discussed within this lecture:

- Playground Memory Issues
- Quick Refresh On The EUTxO Model
- EUTxO Scripts
- Other Important Purposes of ScriptPurpose
- Context - TxInfo (Acronym for Transaction Info)
- Current Context (Updated)
- On-Chain VS Off-Chain Validation
- General Checks
- Time Intervals
- Interesting Use Cases: Smart Contracts

- Homework 1.0
- Homework 2.0

References

[1] Plutus Development Team, 2021. plutus-ledger-api-0.1.0.0: Interface to the Plutus ledger for the Cardano ledger. Plutus.V1.Ledger.Contexts, section: Pending transactions and related types https://playground.plutus.iohkdev.io/tutorial/haddock/plutus-ledger-api/html/Plutus-V1-Ledger-Contexts.html

[2] Manish Jain and Constantinos Dovrolis. 2004. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In Proceedings of the 4th ACM SIGCOMM conference on Internet measurement (IMC '04). Association for Computing Machinery, New York, NY, USA, 272–277. DOI: https://doi.org/10.1145/1028788.1028825

[3] Learn Cloud Native Development Team, 2019. The Basics - Fallacies of Distributed Systems https://www.learncloudnative.com/blog/2019-11-26-fallacies_of_distributed_systems

[4] Plutus Engineering Team, IOHK. plutus-ledger-api-0.1.0.0: Interface to the Plutus ledger for the Cardano ledger. https://alpha.marlowe.iohkdev.io/doc/haddock/plutus-ledger-api/html/Plutus-V1-Ledger-Contexts.html

Footnotes

1. The description is that of a pending transaction. This is the view as seen by validator scripts, so some details are stripped out. [1]

2. In order to produce a validator, which, in turn may consume any given UTxO (or not consume it!), we must provide (even if they're empty arguments) a datum (typically of type 'record' & implements isData), a redeemer (typically a set of functions or executable code which makes checks against unspent transaction output details) and the context (is of type: ScriptContext). We also understand that empty arguments must be in the form of unit data: `()` and if we're doing something useful, we'll likely be providing genuine arguments. Thus, the datum will likely contain some data (as you may expect) to be used in conjunction with the redeemer, which will be defined as a set of functions to likely draw from the ScriptContext; the compiled product of which (the validator) may then decide as to whether or not to spend the EUTxO in consideration. At least, this is **my understanding thus far.**