Lecture Five

> "It might be true that there are six billion people in the world and counting. Nevertheless, what you do makes a difference. It makes a difference, first of all, in material terms. It makes a difference to other people and it sets an example."
> — Robert Solomon, Waking Life

Recently, I noticed an up-tick in questions about Git and GitHub on the IOHK Plutus Pioneer Discord. For this reason, I took a few hours out to write a brief introduction to Git (and how ones local git repo ties to, in this instance, the remote IOHK repo). Really, you can find all the answers you should need at https://git-scm.com/, thus, I have included some links to specific pages on git-scm. You can view the introduction I wrote here. If any further questions pop up, I'll do my best to answer and then append answers to the introduction document. Again, You Can Read The Introduction to Git I Wrote Here.

1. Introduction

Within this particular set of lecture notes we will be addressing Native Tokens on Cardano. Firstly, a discussion surrounding the building blocks of any such Native Token is provided. Secondly, the definition of a Native Token is presented, along with how any such token may be contrasted with Ada / Lovelace. Furthermore, the concepts and implementation details surrounding Policies (to create or destroy Native Tokens under well defined constraints) are demonstrated. Finally, an investigation into Non Fungible Tokens (NFTs) and how they relate to Native Tokens is surveyed.

Let it be known that I have had to take some time out to put together a catalyst proposal. Watch this space.

2. Genesis: And God Said Let Us Make An AssetClass

Turns out God seemed to like the image of Ada Lovelace [1] more than the image of Man!

All jokes aside *(I told you this going to be a somewhat creative technical document, you were warned)*, I suppose the first question is: What Is A Native Token?

At this point, I think most 'techy' people do understand the concept of Cardano's Native Tokens, NFTs (I would love to see fractional NFTs by the way [2], and perhaps some nicer ways of implementing NFT payloads without bloating the chain).

2.1 What Is A Native Token?

To answer that question, we are **required** to understand AssetClasses in order to understand Native Tokens. An AssetClass is a type, which means it takes a constructor, so let's deconstruct that constructor!

`unAssetClass :: (CurrencySymbol, TokenName)`

So, we have a type 'newtype' AssetClass which takes a 'newtype' CurrencySymbol and a 'newtype' TokenName. Don't worry, CurrencySymbol and TokenName are both just wrappers for a ByteString. Further, if one sets `-XOverloadedStrings` then you can simply use a string literal for any ByteString (the CurrencySymbol does have to be written in hexadecimal though). This does in fact mean that a Native Token is defined by the combination of a CurrencySymbol and a TokenName. **There is one exception, and we'll get to that now.**

2.2 Ada Lovelace

Apart from being one of the first ever programmers on the planet and correct about Babbage and his vision for computing... Lovelace is in fact an AssetClass (when it comes to Cardano anyway). Meaning, it is a Native Token (of sorts). However, Native Tokens can be minted (otherwise how would one bring one into existence), and burnt (destroyed). But, you cannot mint anymore Lovelace / Ada, what gives?

As in §2.1, a Native Token that can be **minted** and / or **burned** within a set of constraints requiring a CurrencySymbol and a TokenName. Shortly, it will be demonstrated that Lovelace does not have a CurrencySymbol and therefore cannot be minted or burned (which is a good thing). It means that there is only a set amount of lovelace / ada within the ecosystem.

```
import Plutus.V1.Ledger.Value
import Plutus.V1.Ledger.Ada
:set -XOverLoadedStrings
:t adaSymbol
adaSymbol :: CurrencySymbol
adaSymbol

adaToken :: TokenName
adaToken
""
-- how do we create lovelace for testing purposes?
:t lovelaceValueOf
lovelaceValueOf :: Integer -> Value
lovelaceValueOf 123
Value (Map [(...,Map [("", 123)])]) -- a currency symbol exists at ...
-- This is actually extremely similar to how text search indexing work
s: word -> (page, freq)
```

2.3 Native Tokens: The Importance Of CurrencySymbols

It has been made clear that CurrencySymbols must be written in hexadecimal. The reason for this is because the script address generated to contain the logic (to define the constraints) for minting and burning Native Tokens uses a hash function of the CurrencySymbol. If you think about it, it makes perfect sense: if ada lovelace has no CurrencySymbol or TokenName, then there cannot be a script address responsible for minting and / or burning ada lovelace. However, as the AssetClass constructor takes two ByteStrings as a constructor, the first of which is the CurrencySymbol and hashing the value contained within CurrencySymbol produces an address where the associated policy script sits.

2.4 The Basics: Minting Policy Scripts

As you are now aware, scripts for creating and destroying native tokens sit at an address which results from hashing the CurrencySymbol. However, if you recall the UTxO model, the input must equal the output (minus the fees). This could never be true of the EUTxO model, otherwise it would be impossible to create Native Tokens. Why? Because we can create transactions which are not simply 'spending'. Further, the fees associated with running Native Token scripts depend on the size of the transaction (in bytes, not in value) and also the size of the scripts that need to be run to validate the transaction. In computer science terms, the fees depend on the algorithmic complexity of the arbitrary logic which sites at the script address. It sounds like it is simply the space complexity which determines the fees, but I would imagine it is both space and time. In short: `O(x) ~ fees | x : script`.

3. Creating A Native Token

Enough chit chat, let's go ahead and create a native token, or two (in the repl, step by step).

Let us set overloaded strings such that we can use string literals for the ByteString constructor, then we'll import everything we need to apply what we have learnt thus far (Value.hs and Ada.hs).

```
Prelude Week05.Free> :set -XOverloadedStrings
Prelude Week05.Free> import Plutus.V1.Ledger.Value
Prelude Plutus.V1.Ledger.Value Week05.Free> import Plutus.V1.Ledger.Ad
a
```

Now, we're about to create some Native Tokens. But let's stop for a second because there is something important we need to review. It just so happens that Values are instances of Monoids which means they can be combined in various ways. Thus, it is likely a good idea to review some information about Monoids.

The Monoid Laws

> Before moving on to specific instances of Monoid, let's take a brief look at the monoid laws. We mentioned that there has to be a value that acts as the identity with respect to the binary function and that the binary function has to be associative. It's possible to make instances of Monoid that don't follow these rules, but such instances are of no use to anyone because when using the Monoid type class, we rely on its instances acting like monoids. Otherwise, what's the point? That's why when making instances, we have to make sure they follow these laws:
>
> mempty `mappend` x = x
> x `mappend` mempty = x
> (x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)
>
> The first two state that mempty has to act as the identity with respect to mappend and the third says that mappend has to be associative i.e. that it the order in which we use mappend to reduce several monoid values into one doesn't matter. **Haskell doesn't enforce these laws, so we as the programmer have to be careful that our instances do indeed obey them.** [3]

Now that we're ready to move on, let's get cracking and write some code.

```
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> :t si
ngleton
singleton :: CurrencySymbol -> TokenName -> Integer -> Value
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> singl
eton "a8ff" "ABC" 72
Value (Map [(a8ff,Map [("ABC",72)])])
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> singl
eton "a8ff" "ABC" 72 <> lovelaceValueOf 42 <> singleton "a8ff" "XYZ" 1
00
Value (Map [(,Map [("",42)]),(a8ff,Map [("ABC",72),("XYZ",100)])])
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> let v
 = singleton "a8ff" "ABC" 72 <> lovelaceValueOf 42 <> singleton "a8ff" "
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> v
Value (Map [(,Map [("",42)]),(a8ff,Map [("ABC",72),("XYZ",100)])])
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> :t va
lueOf
valueOf :: Value -> CurrencySymbol -> TokenName -> Integer
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> value
Of v "a8ff" "XYZ"
100
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> value
Of v "a8ff" "ABC"
72
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> value
Of v "a8ff" ""
0
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> value
Of v "a8ff" "abcsds"
0
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> :t fl
attenValue
flattenValue :: Value -> [(CurrencySymbol, TokenName, Integer)]
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free> flatt
enValue v
[(a8ff,"ABC",72),(a8ff,"XYZ",100),(,"",42)]
Prelude Plutus.V1.Ledger.Value Plutus.V1.Ledger.Ada Week05.Free>
```

4. Minting Policies

- Refresher On validation
- No public key address, script address
- UTxO sits at such a script address
- Tx tries to consume UTxO as an input
- For each script input, the co-responding validation script is run
- Validation script (as input) gets:
    - Datum (comes from the UTxO)

    - Redeemer, which comes from the Tx
    - Context (pretty sure context is part of the UTxO)
        - two fields: ScriptPurpose, txInfo

    - Every ScriptPurpose until now always had the Spending txOutRef
    - txOutRef is a reference to the UTxO we're trying to consume
    - txInfo has all the context information about the Tx which is trying to be validated

- For monetary policies, for minting policies: if the txForgeField (txInfoForge is != 0)
- Then txInfoForge can contain a bag of asset classes (map => map) [Value]
- IFF txInfoForge is != 0, then for every CurrencySymbol in this bag that is being forged:
- The co-responding script sitting @ hash(Currencysymbol) is going to be ran
- These minting policy scripts only have 2 validator inputs: redeemer and the context
- Minting Policy Script only have 2 inputs: Redeemer and the Context, No datum
- Tx provides redeemer (bool) + redeemer for all script inputs
- To summerise:
- IFF the Transaction Forge Field (Context -> txInfo -> txInfoForge) is non-zero...
- Meaning that a Value type sits within that transaction attribute...
- And remember, Value is a map->map, like an indexing data structure... THEN:
- For each currency symbol:
- A hash of the symbol is taken which produces a script address, AND
- The script at that address is executed, AND
- For each script where the redeemer returns True,
- A Transaction is created which will create or burn the Native Token associated with that script...
- It will do so in accordance with the scripts arbitrary logic...
- The fee to be paid depends on the computational complexity of the script located at the address derived from hashing the currency symbol, so long as the redeemer returned True.

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

module Week05.Free where
```

```
import           Control.Monad          hiding (fmap)
import           Data.Aeson             (ToJSON, FromJSON)
import           Data.Text              (Text)
import           Data.Void              (Void)
import           GHC.Generics           (Generic)
import           Plutus.Contract        as Contract
import           Plutus.Trace.Emulator  as Emulator
import qualified PlutusTx
import           PlutusTx.Prelude       hiding (Semigroup(..), unless)
import           Ledger                 hiding (mint, singleton)
import           Ledger.Constraints     as Constraints
import qualified Ledger.Typed.Scripts   as Scripts
import           Ledger.Value           as Value
import           Playground.Contract    (printJson, printSchemas, ensu
reKnownCurrencies, stage, ToSchema)
import           Playground.TH          (mkKnownCurrencies, mkSchemaDe
finitions)
import           Playground.Types       (KnownCurrency (..))
import           Prelude                (IO, Show (..), String)
import           Text.Printf            (printf)
import           Wallet.Emulator.Wallet

{-# INLINABLE mkPolicy #-}
mkPolicy :: () -> ScriptContext -> Bool
mkPolicy () _ = True

policy :: Scripts.MintingPolicy
policy = mkMintingPolicyScript $$(PlutusTx.compile [|| Scripts.wrapMin
tingPolicy mkPolicy ||])

curSymbol :: CurrencySymbol
curSymbol = scriptCurrencySymbol policy

data MintParams = MintParams
    { mpTokenName :: !TokenName
    , mpAmount    :: !Integer
    } deriving (Generic, ToJSON, FromJSON, ToSchema)

type FreeSchema = Endpoint "mint" MintParams

mint :: MintParams -> Contract w FreeSchema Text ()
mint mp = do
    let val     = Value.singleton curSymbol (mpTokenName mp) (mpAmount mp
        lookups = Constraints.mintingPolicy policy
        tx      = Constraints.mustMintValue val
    ledgerTx <- submitTxConstraintsWith @Void lookups tx
    void $ awaitTxConfirmed $ txId ledgerTx
    Contract.logInfo @String $ printf "forged %s" (show val)
```

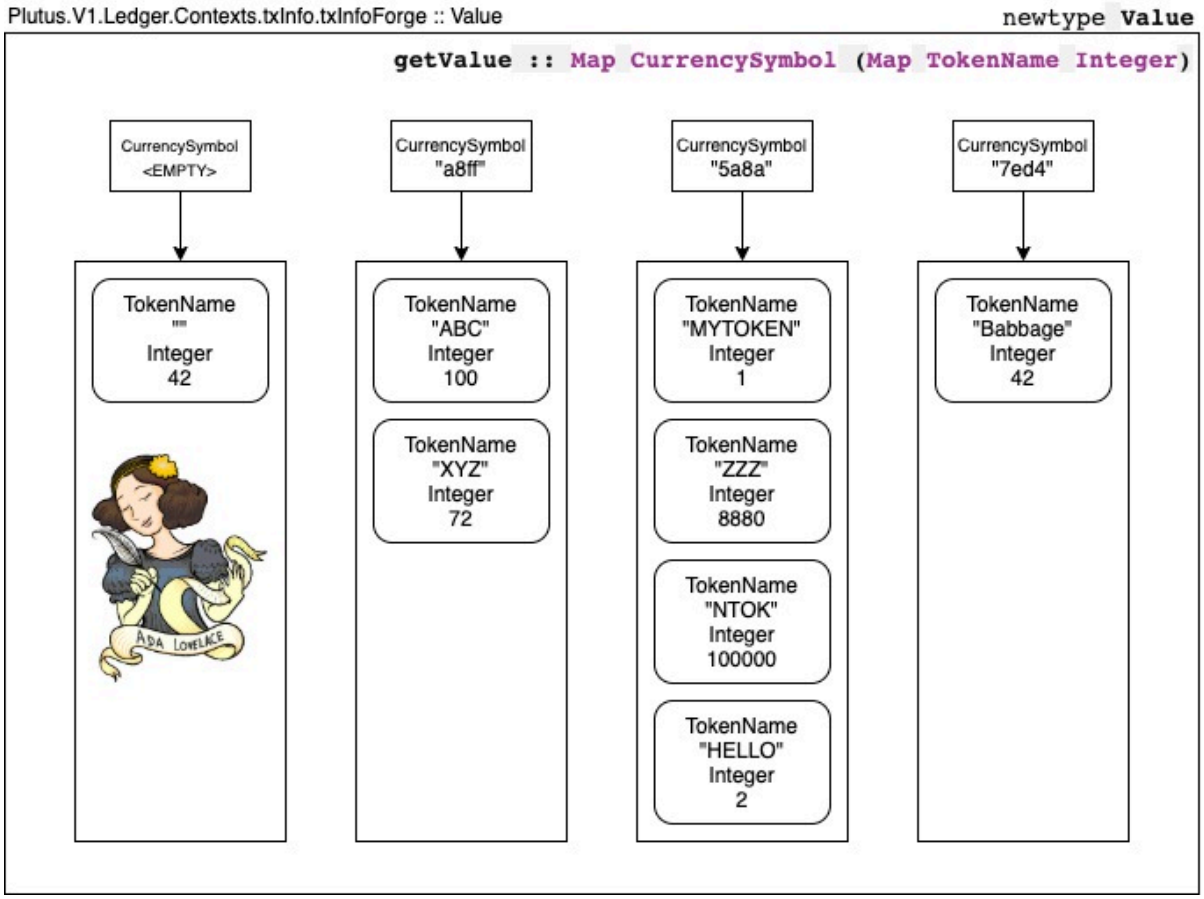
```
endpoints :: Contract () FreeSchema Text ()
endpoints = mint' >> endpoints
  where
    mint' = endpoint @"mint" >>= mint

mkSchemaDefinitions ''FreeSchema

mkKnownCurrencies []

test :: IO ()
test = runEmulatorTraceIO $ do
    let tn = "ABC"
    h1 <- activateContractWallet (Wallet 1) endpoints
    h2 <- activateContractWallet (Wallet 2) endpoints
    callEndpoint @"mint" h1 $ MintParams
        { mpTokenName = tn
        , mpAmount    = 555
        }
    callEndpoint @"mint" h2 $ MintParams
        { mpTokenName = tn
        , mpAmount    = 444
        }
    void $ Emulator.waitNSlots 1
    callEndpoint @"mint" h1 $ MintParams
        { mpTokenName = tn
        , mpAmount    = -222
        }
    void $ Emulator.waitNSlots 1
```

Similarly to validators for spending, when we are producing a 'validator' that implements some kind of monetary policy, such as minting Native Tokens (known as a minting policy), we provide similar arguments as we would to a spending validator. However, it should be noted that there is simply no purpose for a datum parameter for any given policy, it is non-sensical. Thus, there exists only two parameters for any given policy. The two arguments we typically provide are (although, you can have parameterised policies, just as you may parameterised validators), as you might expect: the redeemer and the context.

At the very least we require a redeemer because we need to know whether minting and / or burning of a given token is permitted. The redeemer is required to satisfy the arbitrary logic found at the script address. Furthermore, the context is also required because we must have access to every CurrencySymbol (which is derived through: context.txInfo.txInfoForge :: Value) such that we can produce the script addresses (by applying a hashing algorithm to the CurrencySymbol(s) ByteString).

As the Value is a mapping from X to a mapping of Y to Z, where X and Y are ByteStrings and

Z is an Integer, it is possible to have multiple CurrencySymbols that point to multiple Tokens with varying degrees of amount. See the image below.



Note: Ada Lovelace is not part of the data structure. She existed a long time ago...

The above example is **the simplest** example you might expect to see, because it's a minting policy which always returns True (you can mint or burn as many tokens as you like). This is demonstrated through the nature of the `mkPolicy` function within free.hs (I have also included the section which compiles the policy to Plutus-core and the procedure used to generate a script address):

```
{-# INLINABLE mkPolicy #-} -- required for the oxford brackets
mkPolicy :: () -> ScriptContext -> Bool
mkPolicy () _ = True
policy :: Scripts.MintingPolicy
policy = mkMintingPolicyScript $$(PlutusTx.compile [|| Scripts.wrapMin
tingPolicy mkPolicy ||])
curSymbol :: CurrencySymbol
curSymbol = scriptCurrencySymbol policy
```

In the above instance, we are using (for the most part) typed Haskell to compile to Plutus-core (because, well, you should). The problem that we would normally encounter when utilising the oxford brackets is solved using a library function, as indicated in the above

snippet by `Scripts.wrapMintingPolicy`.

Notice how the declaration of `mkPolicy` takes (as its arguments): unit and ScriptContext with a return type of Bool. The reason why we're able to use unit as an argument for the redeemer is because it does implements the isData instance, otherwise we would also have to create a typed redeemer. So, within this example, regardless of the script context and the redeemer, the policy is **always going to return True.**

Finally, in terms on on-chain code, the script address is produced by declaring a CurrencySymbol and assigning the result of `scriptCurrencySymbol policy` to it. This takes the compiled policy script and hashes it to leave us with the address at which the script will sit.

Off-Chain Code:

```
data MintParams = MintParams
    { mpTokenName :: !TokenName
    , mpAmount    :: !Integer
    } deriving (Generic, ToJSON, FromJSON, ToSchema)

type FreeSchema = Endpoint "mint" MintParams

mint :: MintParams -> Contract w FreeSchema Text ()
mint mp = do
    let val     = Value.singleton curSymbol (mpTokenName mp) (mpAmount mp
        lookups = Constraints.mintingPolicy policy
        tx      = Constraints.mustMintValue val
    ledgerTx <- submitTxConstraintsWith @Void lookups tx
    void $ awaitTxConfirmed $ txId ledgerTx
    Contract.logInfo @String $ printf "forged %s" (show val)

endpoints :: Contract () FreeSchema Text ()
endpoints = mint' >> endpoints
  where
    mint' = endpoint @"mint" >>= mint

mkSchemaDefinitions ''FreeSchema

mkKnownCurrencies []
```

Run Outside The Playground:

```
test :: IO ()
test = runEmulatorTraceIO $ do
    let tn = "ABC"
    h1 <- activateContractWallet (Wallet 1) endpoints
    h2 <- activateContractWallet (Wallet 2) endpoints
    callEndpoint @"mint" h1 $ MintParams
        { mpTokenName = tn
        , mpAmount    = 555
        }
    callEndpoint @"mint" h2 $ MintParams
        { mpTokenName = tn
        , mpAmount    = 444
        }
    void $ Emulator.waitNSlots 1
    callEndpoint @"mint" h1 $ MintParams
        { mpTokenName = tn
        , mpAmount    = -222
        }
    void $ Emulator.waitNSlots 1
```

4.1 Realistic Policies

More realistic policies will implement some additional constraints. Sometimes these constraints will be fairly basic, such as this example shown below. At least this policy doesn't return True regardless of ANY constraints.

To see the full file, click here.

On-Chain Code:

```
{-# INLINABLE mkPolicy #-}
mkPolicy :: PubKeyHash -> () -> ScriptContext -> Bool
mkPolicy pkh () ctx = txSignedBy (scriptContextTxInfo ctx) pkh

policy :: PubKeyHash -> Scripts.MintingPolicy
policy pkh = mkMintingPolicyScript $
    $$(PlutusTx.compile [|| Scripts.wrapMintingPolicy . mkPolicy ||])
    `PlutusTx.applyCode`
    (PlutusTx.liftCode pkh)

curSymbol :: PubKeyHash -> CurrencySymbol
curSymbol = scriptCurrencySymbol . policy
```

Right, let's get some points down here:

- Due to the way in which we're using Haskell Templating and Oxford Brackets, we have

to allow `mkPolicy` to be `INLINABLE`

- During the last section, parameterised policies were mentioned, here we have an example.
- Since a redeemer and a ScriptContext are required to create a policy, we're also parameterising our `mkPolicy` function with a `PubKeyHash`.
- Our policy then takes a public key hash, unit (as a redeemer, since there is no additional arbitrary logic to satisfy) and the contexts.
- We assign the `PubKeyHash` from the script context (as it provides txInfo, which in turn provides the public key hash) as is returned by txSignedBy, to our policy (Bool).
- Now, we need to do the same little trick we did with the validators and compile the parameters separately from the policy (as this is a parameterised policy) using liftCode.
- Finally we generate a CurrencySymbol (which, when hashed = script address).

See The Entire File Here:

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

module Week05.Signed where

import           Control.Monad          hiding (fmap)
import           Data.Aeson             (ToJSON, FromJSON)
import           Data.Text              (Text)
import           Data.Void              (Void)
import           GHC.Generics           (Generic)
import           Plutus.Contract        as Contract
import           Plutus.Trace.Emulator  as Emulator
import qualified PlutusTx
import           PlutusTx.Prelude       hiding (Semigroup(..), unless)
import           Ledger                 hiding (mint, singleton)
import           Ledger.Constraints     as Constraints
import qualified Ledger.Typed.Scripts   as Scripts
import           Ledger.Value           as Value
import           Playground.Contract    (printJson, printSchemas, ensu
reKnownCurrencies, stage, ToSchema)
import           Playground.TH          (mkKnownCurrencies, mkSchemaDe
finitions)
```

```
import           Playground.Types       (KnownCurrency (..))
import           Prelude                (IO, Show (..), String)
import           Text.Printf            (printf)
import           Wallet.Emulator.Wallet

{-# INLINABLE mkPolicy #-}
mkPolicy :: PubKeyHash -> () -> ScriptContext -> Bool
mkPolicy pkh () ctx = txSignedBy (scriptContextTxInfo ctx) pkh

policy :: PubKeyHash -> Scripts.MintingPolicy
policy pkh = mkMintingPolicyScript $
    $$(PlutusTx.compile [|| Scripts.wrapMintingPolicy . mkPolicy ||])
    `PlutusTx.applyCode`
    (PlutusTx.liftCode pkh)

curSymbol :: PubKeyHash -> CurrencySymbol
curSymbol = scriptCurrencySymbol . policy

data MintParams = MintParams
    { mpTokenName :: !TokenName
    , mpAmount    :: !Integer
    } deriving (Generic, ToJSON, FromJSON, ToSchema)

type SignedSchema = Endpoint "mint" MintParams

mint :: MintParams -> Contract w SignedSchema Text ()
mint mp = do
    pkh <- pubKeyHash <$> Contract.ownPubKey
    let val     = Value.singleton (curSymbol pkh) (mpTokenName mp) (mpAm
        lookups = Constraints.mintingPolicy $ policy pkh
        tx      = Constraints.mustMintValue val
    ledgerTx <- submitTxConstraintsWith @Void lookups tx
    void $ awaitTxConfirmed $ txId ledgerTx
    Contract.logInfo @String $ printf "forged %s" (show val)

endpoints :: Contract () SignedSchema Text ()
endpoints = mint' >> endpoints
  where
    mint' = endpoint @"mint" >>= mint

mkSchemaDefinitions ''SignedSchema

mkKnownCurrencies []

test :: IO ()
test = runEmulatorTraceIO $ do
    let tn = "ABC"
    h1 <- activateContractWallet (Wallet 1) endpoints
```

```
    h2 <- activateContractWallet (Wallet 2) endpoints
    callEndpoint @"mint" h1 $ MintParams
        { mpTokenName = tn
        , mpAmount    = 555
        }
    callEndpoint @"mint" h2 $ MintParams
        { mpTokenName = tn
        , mpAmount    = 444
        }
    void $ Emulator.waitNSlots 1
    callEndpoint @"mint" h1 $ MintParams
        { mpTokenName = tn
        , mpAmount    = -222
        }
    void $ Emulator.waitNSlots 1
```

5. Non Fungible Tokens

Within this section we are going to briefly discuss Non Fungible Tokens (NFTs).

Let's quickly recap on what we've learnt thus far:

- It's possible to create a Native Token on Cardano.
- A Native Token is an AssetClass which is constructed using a CurrencySymbol and a TokenName.
- Both CurrencySymbols and TokenNames are simple wrappers for a ByteString.
- Whilst ByteStrings can be implemented using String literals when XOverloadedStrings is set, the CurrencySymbol must be written in hexadecimal.
- A policy script is responsible for minting or burning tokens.
- Once the policy script has been compiled to Plutus-core, it is hashed and can therefore sit within the CurrencySymbol field (as part of an asset class).
- This allows us to generate a script address by compiling the Haskell to Plutus-core and hashing the result. We can therefore always know the script address of any Native Token, as it is made available through the CurrencySymbol.
- The first policy script we wrote was about as simple as you get, we simply created a policy that always returned True, compiled it and generated its CurrencySymbol.
- The next script was slightly more 'real-world' - it only allowed a the appropriate signatory to mint or burn scripts (we parameterised the policy and passed the Public Hash Key required to verifiable sign minting and/or burning of tokens to it).
- However, during both of these examples, there was no limit as to how many tokens you could mint (or burn). This is where NFTs begin to creep into the picture.

NFTs have been (in a sense) available on Cardano for a while now. However, they're not 'true' NFTs, as they've been minted as a Native Token under particular constraints which

make it highly unlikely or impossible to have two of the same Native Tokens. Assuming the aforementioned statement, the implementation of NFTs through the current Native Token implementation is possible. However, this 'work-around' makes it difficult for an average user to verify whether or not any such Native Token being described as Non Fungible is in fact a genuine NFT.

A better idea (as is described below in code) may be to use the consumption of a UTxO (as they are unique: Transactions have fees, which means an INPUT MUST EXIST - where do inputs come from? Well, it's part of the output from a previous transaction. Meaning unspent transaction outputs are verifiably unique: you identify a Tx, find the appropriate UTxO which acts as input to the policy transaction) as a means of ensuring that only one token exists. In this instance we're creating a policy to access a potential unspent transaction output (which may or may not exist), IFF it does exist, the (strangely) the CurrencySymbol and TokenName must exist and the 'amount' MUST EQUAL ONE.

Again, once this policy has been compiled, a script address may be created for it.

Some Notes:

- Making A Policy Which Takes a transaction output reference as a parameter, in addition to the expected: TokenName and ScriptContext.
- The context for this transaction is grabbed. Thus, we extract the set of all inputs and compare it to see if it matches the provided output reference.
- Then we can check that we're only minting one token.
- We can check to ensure that the CurrencySymbol WILL equal the hash of this compiled script.
- And we can check that the tokenNames match.

The policy can then be executed and this ensures that only a single token exists. Thus, it is non fungible.

The off-chain code implements the mint function, which (as I'm starting to get much better at reading Haskell now) is pretty self-explanatory. It's essentially checking that the UTxO matches the input to this Tx, along with a bunch of other constraints associated with verifying the policy to verify some additional conditions, then finally waits on network verification to write to mint.

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

```
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE TypeOperators       #-}

module Week05.NFT where

import           Control.Monad          hiding (fmap)
import qualified Data.Map               as Map
import           Data.Text              (Text)
import           Data.Void              (Void)
import           Plutus.Contract        as Contract
import           Plutus.Trace.Emulator  as Emulator
import qualified PlutusTx
import           PlutusTx.Prelude       hiding (Semigroup(..), unless)
import           Ledger                 hiding (mint, singleton)
import           Ledger.Constraints     as Constraints
import qualified Ledger.Typed.Scripts   as Scripts
import           Ledger.Value           as Value
import           Playground.Contract    (printJson, printSchemas, ensu
reKnownCurrencies, stage, ToSchema)
import           Playground.TH          (mkKnownCurrencies, mkSchemaDe
finitions)
import           Playground.Types       (KnownCurrency (..))
import           Prelude                (IO, Semigroup (..), Show (..)
, String)
import           Text.Printf            (printf)
import           Wallet.Emulator.Wallet

{-# INLINABLE mkPolicy #-}
mkPolicy :: TxOutRef -> TokenName -> () -> ScriptContext -> Bool
mkPolicy oref tn () ctx = traceIfFalse "UTxO not consumed"   hasUTxO
         &&
                           traceIfFalse "wrong amount minted" checkMinted
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    hasUTxO :: Bool
    hasUTxO = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs info

    checkMintedAmount :: Bool
    checkMintedAmount = case flattenValue (txInfoForge info) of
        [(cs, tn', amt)] -> cs  == ownCurrencySymbol ctx && tn' == tn &&
        _                -> False

policy :: TxOutRef -> TokenName -> Scripts.MintingPolicy
policy oref tn = mkMintingPolicyScript $
    $$(PlutusTx.compile [|| \oref' tn' -> Scripts.wrapMintingPolicy $ mk
```

```
    `PlutusTx.applyCode`
    PlutusTx.liftCode oref
    `PlutusTx.applyCode`
    PlutusTx.liftCode tn

curSymbol :: TxOutRef -> TokenName -> CurrencySymbol
curSymbol oref tn = scriptCurrencySymbol $ policy oref tn

type NFTSchema = Endpoint "mint" TokenName

mint :: TokenName -> Contract w NFTSchema Text ()
mint tn = do
    pk    <- Contract.ownPubKey
    utxos <- utxoAt (pubKeyAddress pk)
    case Map.keys utxos of
        []       -> Contract.logError @String "no utxo found"
        oref : _ -> do
            let val     = Value.singleton (curSymbol oref tn) tn 1
                lookups = Constraints.mintingPolicy (policy oref tn) <> (
                tx      = Constraints.mustMintValue val <> Constraints.mu
            ledgerTx <- submitTxConstraintsWith @Void lookups tx
            void $ awaitTxConfirmed $ txId ledgerTx
            Contract.logInfo @String $ printf "forged %s" (show val)

endpoints :: Contract () NFTSchema Text ()
endpoints = mint' >> endpoints
  where
    mint' = endpoint @"mint" >>= mint

mkSchemaDefinitions ''NFTSchema

mkKnownCurrencies []

test :: IO ()
test = runEmulatorTraceIO $ do
    let tn = "ABC"
    h1 <- activateContractWallet (Wallet 1) endpoints
    h2 <- activateContractWallet (Wallet 2) endpoints
    callEndpoint @"mint" h1 tn
    callEndpoint @"mint" h2 tn
    void $ Emulator.waitNSlots 1
```

References

[1] Isaac, A.M., 2018.
Computational thought from Descartes to Lovelace.
The Routledge Handbook of the Computational Mind, pp.9-22.

[2] Algorand, MAY 07, 2021.
How Algorand Democratizes the Access to the NFT Market with Fractional NFTs.
https://www.algorand.com/resources/blog/algorand-nft-market-fractional-nfts

[3] Lipovaca, M., 2011.
Learn you a haskell for great good!: a beginner's guide.
no starch press.

Appendix

Value.hs

```
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE DeriveAnyClass        #-}
{-# LANGUAGE DeriveGeneric         #-}
{-# LANGUAGE DerivingVia           #-}
{-# LANGUAGE LambdaCase            #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoImplicitPrelude     #-}
{-# LANGUAGE OverloadedLists       #-}
{-# LANGUAGE OverloadedStrings     #-}
{-# LANGUAGE TemplateHaskell       #-}
{-# LANGUAGE TypeApplications      #-}
{-# OPTIONS_GHC -Wno-orphans #-}
-- Prevent unboxing, which the plugin can't deal with
{-# OPTIONS_GHC -fno-strictness #-}
{-# OPTIONS_GHC -fno-omit-interface-pragmas #-}
{-# OPTIONS_GHC -fno-spec-constr #-}
{-# OPTIONS_GHC -fno-specialise #-}

-- | Functions for working with 'Value'.
module Plutus.V1.Ledger.Value(
    -- ** Currency symbols
      CurrencySymbol(..)
    , currencySymbol
    , mpsSymbol
    , currencyMPSHash
    -- ** Token names
    , TokenName(..)
    , tokenName
    , toString
    -- * Asset classes
    , AssetClass(..)
    , assetClass
    , assetClassValue
    , assetClassValueOf
    -- ** Value
```

```
    , Value(..)
    , singleton
    , valueOf
    , scale
    , symbols
      -- * Partial order operations
    , geq
    , gt
    , leq
    , lt
      -- * Etc.
    , isZero
    , split
    , unionWith
    , flattenValue
    ) where

import qualified Prelude                            as Haskell

import           Codec.Serialise.Class              (Serialise)
import           Control.DeepSeq                    (NFData)
import           Control.Monad                      (guard)
import           Data.Aeson                         (FromJSON, FromJSONK
ey, ToJSON, ToJSONKey, (.:))
import qualified Data.Aeson                         as JSON
import qualified Data.Aeson.Extras                  as JSON
import qualified Data.ByteString                    as BS
import           Data.Hashable                      (Hashable)
import qualified Data.List                          (sortBy)
import           Data.String                        (IsString (fromStrin
g))
import           Data.Text                          (Text)
import qualified Data.Text                          as Text
import qualified Data.Text.Encoding                 as E
import           Data.Text.Prettyprint.Doc
import           Data.Text.Prettyprint.Doc.Extras
import           GHC.Generics                       (Generic)
import           GHC.Show                           (showList__)
import           Plutus.V1.Ledger.Bytes             (LedgerBytes (Ledger
Bytes))
import           Plutus.V1.Ledger.Orphans           ()
import           Plutus.V1.Ledger.Scripts
import qualified PlutusTx                           as PlutusTx
import qualified PlutusTx.AssocMap                  as Map
import           PlutusTx.Lift                      (makeLift)
import qualified PlutusTx.Ord                       as Ord
import           PlutusTx.Prelude                   as PlutusTx
import           PlutusTx.These
```

```
newtype CurrencySymbol = CurrencySymbol { unCurrencySymbol :: PlutusTx
.BuiltinByteString }
    deriving (IsString, Haskell.Show, Serialise, Pretty) via LedgerBytes
    deriving stock (Generic)
    deriving newtype (Haskell.Eq, Haskell.Ord, Eq, Ord, PlutusTx.ToData,
    deriving anyclass (Hashable, ToJSONKey, FromJSONKey,  NFData)

instance ToJSON CurrencySymbol where
  toJSON currencySymbol =
    JSON.object
      [ ( "unCurrencySymbol"
        , JSON.String .
          JSON.encodeByteString .
          PlutusTx.fromBuiltin .
          unCurrencySymbol $
          currencySymbol)
      ]

instance FromJSON CurrencySymbol where
  parseJSON =
    JSON.withObject "CurrencySymbol" $ \object -> do
      raw <- object .: "unCurrencySymbol"
      bytes <- JSON.decodeByteString raw
      Haskell.pure $ CurrencySymbol $ PlutusTx.toBuiltin bytes

makeLift ''CurrencySymbol

{-# INLINABLE mpsSymbol #-}
-- | The currency symbol of a monetay policy hash
mpsSymbol :: MintingPolicyHash -> CurrencySymbol
mpsSymbol (MintingPolicyHash h) = CurrencySymbol h

{-# INLINABLE currencyMPSHash #-}
-- | The minting policy hash of a currency symbol
currencyMPSHash :: CurrencySymbol -> MintingPolicyHash
currencyMPSHash (CurrencySymbol h) = MintingPolicyHash h

{-# INLINABLE currencySymbol #-}
-- | Creates `CurrencySymbol` from raw `ByteString`.
currencySymbol :: BS.ByteString -> CurrencySymbol
currencySymbol = CurrencySymbol . PlutusTx.toBuiltin

-- | ByteString of a name of a token, shown as UTF-8 string when possi
ble
newtype TokenName = TokenName { unTokenName :: PlutusTx.BuiltinByteStr
ing }
    deriving (Serialise) via LedgerBytes
```

```
    deriving stock (Generic)
    deriving newtype (Haskell.Eq, Haskell.Ord, Eq, Ord, PlutusTx.ToData,
    deriving anyclass (Hashable, NFData)
    deriving Pretty via (PrettyShow TokenName)

instance IsString TokenName where
    fromString = fromText . Text.pack

{-# INLINABLE tokenName #-}
-- | Creates `TokenName` from raw `ByteString`.
tokenName :: BS.ByteString -> TokenName
tokenName = TokenName . PlutusTx.toBuiltin

fromText :: Text -> TokenName
fromText = tokenName . E.encodeUtf8

fromTokenName :: (BS.ByteString -> r) -> (Text -> r) -> TokenName -> r
fromTokenName handleBytestring handleText (TokenName bs) = either (\_
-> handleBytestring $ PlutusTx.fromBuiltin bs) handleText $ E.decodeUt
f8' (PlutusTx.fromBuiltin bs)

asBase16 :: BS.ByteString -> Text
asBase16 bs = Text.concat ["0x", JSON.encodeByteString bs]

quoted :: Text -> Text
quoted s = Text.concat ["\"", s, "\""]

toString :: TokenName -> Haskell.String
toString = Text.unpack . fromTokenName asBase16 id

instance Haskell.Show TokenName where
    show = Text.unpack . fromTokenName asBase16 quoted

{- note [Roundtripping token names]

How to properly roundtrip a token name that is not valid UTF-8 through
 PureScript
without a big rewrite of the API?
We prefix it with a zero byte so we can recognize it when we get a byt
estring value back,
and we serialize it base16 encoded, with 0x in front so it will look a
s a hex string.
(Browsers don't render the zero byte.)
-}

instance ToJSON TokenName where
    toJSON = JSON.object . Haskell.pure . (,) "unTokenName" . JSON.toJSO
        fromTokenName
```

```
            (\bs -> Text.cons '\NUL' (asBase16 bs))
            (\t -> case Text.take 1 t of "\NUL" -> Text.concat ["\NUL\NUL

instance FromJSON TokenName where
    parseJSON =
        JSON.withObject "TokenName" $ \object -> do
        raw <- object .: "unTokenName"
        fromJSONText raw
        where
            fromJSONText t = case Text.take 3 t of
                "\NUL0x"       -> either Haskell.fail (Haskell.pure . to
                "\NUL\NUL\NUL" -> Haskell.pure . fromText . Text.drop 2 $
                _              -> Haskell.pure . fromText $ t

makeLift ''TokenName

-- | An asset class, identified by currency symbol and token name.
newtype AssetClass = AssetClass { unAssetClass :: (CurrencySymbol, Tok
enName) }
    deriving stock (Generic)
    deriving newtype (Haskell.Eq, Haskell.Ord, Haskell.Show, Eq, Ord, Se
    deriving anyclass (Hashable, NFData, ToJSON, FromJSON)
    deriving Pretty via (PrettyShow (CurrencySymbol, TokenName))

{-# INLINABLE assetClass #-}
assetClass :: CurrencySymbol -> TokenName -> AssetClass
assetClass s t = AssetClass (s, t)

makeLift ''AssetClass

-- | A cryptocurrency value. This is a map from 'CurrencySymbol's to a
-- quantity of that currency.
--
-- Operations on currencies are usually implemented /pointwise/. That
is,
-- we apply the operation to the quantities for each currency in turn.
 So
-- when we add two 'Value's the resulting 'Value' has, for each curren
cy,
-- the sum of the quantities of /that particular/ currency in the argu
ment
-- 'Value'. The effect of this is that the currencies in the 'Value' a
re "independent",
-- and are operated on separately.
--
-- Whenever we need to get the quantity of a currency in a 'Value' whe
re there
-- is no explicit quantity of that currency in the 'Value', then the q
```

```
uantity is
-- taken to be zero.
--
-- See note [Currencies] for more details.
newtype Value = Value { getValue :: Map.Map CurrencySymbol (Map.Map To
kenName Integer) }
    deriving stock (Generic)
    deriving anyclass (ToJSON, FromJSON, Hashable, NFData)
    deriving newtype (Serialise, PlutusTx.ToData, PlutusTx.FromData, Plu
    deriving Pretty via (PrettyShow Value)

instance Haskell.Show Value where
    showsPrec d v =
        Haskell.showParen (d Haskell.== 11) $
            Haskell.showString "Value " . (Haskell.showParen True (showsM
        where Value rep = normalizeValue v
              showsMap sh m = Haskell.showString "Map " . showList__ sh
              showPair s (x,y) = Haskell.showParen True $ Haskell.shows x

normalizeValue :: Value -> Value
normalizeValue = Value . Map.fromList . sort . filterRange (/=Map.empt
y)
               . mapRange normalizeTokenMap . Map.toList . getValue
  where normalizeTokenMap = Map.fromList . sort . filterRange (/=0) . Map
        filterRange p kvs = [(k,v) | (k,v) <- kvs, p v]
        mapRange f xys = [(x,f y) | (x,y) <- xys]
        sort xs = Data.List.sortBy compare xs

-- Orphan instances for 'Map' to make this work
instance (ToJSON v, ToJSON k) => ToJSON (Map.Map k v) where
    toJSON = JSON.toJSON . Map.toList

instance (FromJSON v, FromJSON k) => FromJSON (Map.Map k v) where
    parseJSON v = Map.fromList Haskell.<$> JSON.parseJSON v

deriving anyclass instance (Hashable k, Hashable v) => Hashable (Map.M
ap k v)
deriving anyclass instance (Serialise k, Serialise v) => Serialise (Ma
p.Map k v)

makeLift ''Value

instance Haskell.Eq Value where
    (==) = eq

instance Eq Value where
    {-# INLINABLE (==) #-}
    (==) = eq
```

```
-- No 'Ord Value' instance since 'Value' is only a partial order, so '
compare' can't
-- do the right thing in some cases.

instance Haskell.Semigroup Value where
    (<>) = unionWith (+)

instance Semigroup Value where
    {-# INLINABLE (<>) #-}
    (<>) = unionWith (+)

instance Haskell.Monoid Value where
    mempty = Value Map.empty

instance Monoid Value where
    {-# INLINABLE mempty #-}
    mempty = Value Map.empty

instance Group Value where
    {-# INLINABLE inv #-}
    inv = scale @Integer @Value (-1)

deriving via (Additive Value) instance AdditiveSemigroup Value
deriving via (Additive Value) instance AdditiveMonoid Value
deriving via (Additive Value) instance AdditiveGroup Value

instance Module Integer Value where
    {-# INLINABLE scale #-}
    scale i (Value xs) = Value (fmap (fmap (\i' -> i * i')) xs)

instance JoinSemiLattice Value where
    {-# INLINABLE (\/) #-}
    (\/) = unionWith Ord.max

instance MeetSemiLattice Value where
    {-# INLINABLE (/\) #-}
    (/\) = unionWith Ord.min

{- note [Currencies]

The 'Value' type represents a collection of amounts of different curre
ncies.

We can think of 'Value' as a vector space whose dimensions are
currencies. At the moment there is only a single currency (Ada), so 'V
alue'
contains one-dimensional vectors. When currency-creating transactions
```

```
are
implemented, this will change and the definition of 'Value' will chang
e to a
'Map Currency Int', effectively a vector with infinitely many dimensio
ns whose
non-zero values are recorded in the map.

To create a value of 'Value', we need to specifiy a currency. This can
 be done
using 'Ledger.Ada.adaValueOf'. To get the ada dimension of 'Value' we
use
'Ledger.Ada.fromValue'. Plutus contract authors will be able to define
 modules
similar to 'Ledger.Ada' for their own currencies.

-}

{-# INLINABLE valueOf #-}
-- | Get the quantity of the given currency in the 'Value'.
valueOf :: Value -> CurrencySymbol -> TokenName -> Integer
valueOf (Value mp) cur tn =
    case Map.lookup cur mp of
        Nothing -> 0 :: Integer
        Just i  -> case Map.lookup tn i of
            Nothing -> 0
            Just v  -> v

{-# INLINABLE symbols #-}
-- | The list of 'CurrencySymbol's of a 'Value'.
symbols :: Value -> [CurrencySymbol]
symbols (Value mp) = Map.keys mp

{-# INLINABLE singleton #-}
-- | Make a 'Value' containing only the given quantity of the given cu
rrency.
singleton :: CurrencySymbol -> TokenName -> Integer -> Value
singleton c tn i = Value (Map.singleton c (Map.singleton tn i))

{-# INLINABLE assetClassValue #-}
-- | A 'Value' containing the given amount of the asset class.
assetClassValue :: AssetClass -> Integer -> Value
assetClassValue (AssetClass (c, t)) i = singleton c t i

{-# INLINABLE assetClassValueOf #-}
-- | Get the quantity of the given 'AssetClass' class in the 'Value'.
assetClassValueOf :: Value -> AssetClass -> Integer
assetClassValueOf v (AssetClass (c, t)) = valueOf v c t
```

```
{-# INLINABLE unionVal #-}
-- | Combine two 'Value' maps
unionVal :: Value -> Value -> Map.Map CurrencySymbol (Map.Map TokenNam
e (These Integer Integer))
unionVal (Value l) (Value r) =
    let
        combined = Map.union l r
        unThese k = case k of
            This a    -> This <$> a
            That b    -> That <$> b
            These a b -> Map.union a b
    in unThese <$> combined

{-# INLINABLE unionWith #-}
unionWith :: (Integer -> Integer -> Integer) -> Value -> Value -> Valu
e
unionWith f ls rs =
    let
        combined = unionVal ls rs
        unThese k' = case k' of
            This a    -> f a 0
            That b    -> f 0 b
            These a b -> f a b
    in Value (fmap (fmap unThese) combined)

{-# INLINABLE flattenValue #-}
-- | Convert a value to a simple list, keeping only the non-zero amoun
ts.
flattenValue :: Value -> [(CurrencySymbol, TokenName, Integer)]
flattenValue v = do
    (cs, m) <- Map.toList $ getValue v
    (tn, a) <- Map.toList m
    guard $ a /= 0
    return (cs, tn, a)

-- Num operations

{-# INLINABLE isZero #-}
-- | Check whether a 'Value' is zero.
isZero :: Value -> Bool
isZero (Value xs) = Map.all (Map.all (\i -> 0 == i)) xs

{-# INLINABLE checkPred #-}
checkPred :: (These Integer Integer -> Bool) -> Value -> Value -> Bool
checkPred f l r =
    let
      inner :: Map.Map TokenName (These Integer Integer) -> Bool
      inner = Map.all f
```

```
    in
      Map.all inner (unionVal l r)

{-# INLINABLE checkBinRel #-}
-- | Check whether a binary relation holds for value pairs of two 'Val
ue' maps,
--   supplying 0 where a key is only present in one of them.
checkBinRel :: (Integer -> Integer -> Bool) -> Value -> Value -> Bool
checkBinRel f l r =
    let
        unThese k' = case k' of
            This a    -> f a 0
            That b    -> f 0 b
            These a b -> f a b
    in checkPred unThese l r

{-# INLINABLE geq #-}
-- | Check whether one 'Value' is greater than or equal to another. Se
e 'Value' for an explanation of how operations on 'Value's work.
geq :: Value -> Value -> Bool
-- If both are zero then checkBinRel will be vacuously true, but this
is fine.
geq = checkBinRel (>=)

{-# INLINABLE gt #-}
-- | Check whether one 'Value' is strictly greater than another. See '
Value' for an explanation of how operations on 'Value's work.
gt :: Value -> Value -> Bool
-- If both are zero then checkBinRel will be vacuously true. So we hav
e a special case.
gt l r = not (isZero l && isZero r) && checkBinRel (>) l r

{-# INLINABLE leq #-}
-- | Check whether one 'Value' is less than or equal to another. See '
Value' for an explanation of how operations on 'Value's work.
leq :: Value -> Value -> Bool
-- If both are zero then checkBinRel will be vacuously true, but this
is fine.
leq = checkBinRel (<=)

{-# INLINABLE lt #-}
-- | Check whether one 'Value' is strictly less than another. See 'Val
ue' for an explanation of how operations on 'Value's work.
lt :: Value -> Value -> Bool
-- If both are zero then checkBinRel will be vacuously true. So we hav
e a special case.
lt l r = not (isZero l && isZero r) && checkBinRel (<) l r
```

```
{-# INLINABLE eq #-}
-- | Check whether one 'Value' is equal to another. See 'Value' for an
 explanation of how operations on 'Value's work.
eq :: Value -> Value -> Bool
-- If both are zero then checkBinRel will be vacuously true, but this
is fine.
eq = checkBinRel (==)

-- | Split a value into its positive and negative parts. The first ele
ment of
--   the tuple contains the negative parts of the value, the second el
ement
--   contains the positive parts.
--
--   @negate (fst (split a)) `plus` (snd (split a)) == a@
--
{-# INLINABLE split #-}
split :: Value -> (Value, Value)
split (Value mp) = (negate (Value neg), Value pos) where
  (neg, pos) = Map.mapThese splitIntl mp

  splitIntl :: Map.Map TokenName Integer -> These (Map.Map TokenName Int
  splitIntl mp' = These l r where
    (l, r) = Map.mapThese (\i -> if i <= 0 then This i else That i) mp'
```

Ada.hs

```
{-# LANGUAGE DataKinds         #-}
{-# LANGUAGE DeriveAnyClass    #-}
{-# LANGUAGE DeriveGeneric     #-}
{-# LANGUAGE DerivingVia       #-}
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE TemplateHaskell   #-}
-- Otherwise we get a complaint about the 'fromIntegral' call in the g
enerated instance of 'Integral' for 'Ada'
{-# OPTIONS_GHC -Wno-identities #-}
{-# OPTIONS_GHC -fno-omit-interface-pragmas #-}
-- | Functions for working with 'Ada' in Template Haskell.
module Plutus.V1.Ledger.Ada(
      Ada (..)
    , getAda
    , adaSymbol
    , adaToken
    -- * Constructors
    , fromValue
    , toValue
    , lovelaceOf
    , adaOf
```

```
    , lovelaceValueOf
    , adaValueOf
    -- * Num operations
    , divide
    -- * Etc.
    , isZero
    ) where

import qualified Prelude                           as Haskell

import           Data.Fixed

import           Codec.Serialise.Class             (Serialise)
import           Data.Aeson                        (FromJSON, ToJSON)
import           Data.Tagged
import           Data.Text.Prettyprint.Doc.Extras
import           GHC.Generics                      (Generic)
import           Plutus.V1.Ledger.Value            (CurrencySymbol (..)
, TokenName (..), Value)
import qualified Plutus.V1.Ledger.Value            as TH
import qualified PlutusTx                          as PlutusTx
import           PlutusTx.Lift                     (makeLift)
import           PlutusTx.Prelude                  hiding (divide)
import qualified PlutusTx.Prelude                  as P

{-# INLINABLE adaSymbol #-}
-- | The 'CurrencySymbol' of the 'Ada' currency.
adaSymbol :: CurrencySymbol
adaSymbol = CurrencySymbol emptyByteString

{-# INLINABLE adaToken #-}
-- | The 'TokenName' of the 'Ada' currency.
adaToken :: TokenName
adaToken = TokenName emptyByteString

-- | ADA, the special currency on the Cardano blockchain. The unit of
Ada is Lovelace, and
--   1M Lovelace is one Ada.
--   See note [Currencies] in 'Ledger.Validation.Value.TH'.
newtype Ada = Lovelace { getLovelace :: Integer }
    deriving (Haskell.Enum)
    deriving stock (Haskell.Eq, Haskell.Ord, Haskell.Show, Generic)
    deriving anyclass (ToJSON, FromJSON)
    deriving newtype (Eq, Ord, Haskell.Num, AdditiveSemigroup, AdditiveM
    deriving Pretty via (Tagged "Lovelace:" Integer)

instance Haskell.Semigroup Ada where
    Lovelace a1 <> Lovelace a2 = Lovelace (a1 + a2)
```

```
instance Semigroup Ada where
    Lovelace a1 <> Lovelace a2 = Lovelace (a1 + a2)

instance Haskell.Monoid Ada where
    mempty = Lovelace 0

instance Monoid Ada where
    mempty = Lovelace 0

makeLift ''Ada

{-# INLINABLE getAda #-}
-- | Get the amount of Ada (the unit of the currency Ada) in this 'Ada
' value.
getAda :: Ada -> Micro
getAda (Lovelace i) = MkFixed i

{-# INLINABLE toValue #-}
-- | Create a 'Value' containing only the given 'Ada'.
toValue :: Ada -> Value
toValue (Lovelace i) = TH.singleton adaSymbol adaToken i

{-# INLINABLE fromValue #-}
-- | Get the 'Ada' in the given 'Value'.
fromValue :: Value -> Ada
fromValue v = Lovelace (TH.valueOf v adaSymbol adaToken)

{-# INLINABLE lovelaceOf #-}
-- | Create 'Ada' representing the given quantity of Lovelace (the uni
t of the currency Ada).
lovelaceOf :: Integer -> Ada
lovelaceOf = Lovelace

{-# INLINABLE adaOf #-}
-- | Create 'Ada' representing the given quantity of Ada (1M Lovelace)
.
adaOf :: Micro -> Ada
adaOf (MkFixed x) = Lovelace x

{-# INLINABLE lovelaceValueOf #-}
-- | A 'Value' with the given amount of Lovelace (the currency unit).
--
--   @lovelaceValueOf == toValue . lovelaceOf@
--
lovelaceValueOf :: Integer -> Value
lovelaceValueOf = TH.singleton adaSymbol adaToken
```

```
{-# INLINABLE adaValueOf #-}
-- | A 'Value' with the given amount of Ada (the currency unit).
--
--   @adaValueOf == toValue . adaOf@
--
adaValueOf :: Micro -> Value
adaValueOf (MkFixed x) = TH.singleton adaSymbol adaToken x

{-# INLINABLE divide #-}
-- | Divide one 'Ada' value by another.
divide :: Ada -> Ada -> Ada
divide (Lovelace a) (Lovelace b) = Lovelace (P.divide a b)

{-# INLINABLE isZero #-}
-- | Check whether an 'Ada' value is zero.
isZero :: Ada -> Bool
isZero (Lovelace i) = i == 0
```

Random Notes Whilst Listening To Lecture...

- UTxO : Address & Value
- EUTxO : Address & Value & Datum
- newType New Type : Value
    - Value
        - getValue :: Map CurrencySymbol (Map TokenName Integer)
        - If I understand correctly, this is embedding Native Tokens or Metadata Or Sorts Into Another Native Token?
        - Value === Map AssetClass Integer
        - This makes sense, because the value is an AssetClass which contains an Integer (Value) - although I would have thought this would need to be a floating point number.
        - Value returns the number of units which are in each AssetClass

- The new types: TokenName and CurrencySymbol implement the IsString class, so we can use -XOverLoadedStrings to enter string literals.
- Since TokenName and CurrencySymbol are both just essentially wrappers for a ByteString, we can enter string literals to their constructors.
- Values Containing Native Tokens
- :t singleton -- currencysymbol token name integear <- args
- Singleton :: CurrencySymbol -> TokenName -> Integer -> Value
- Singleton takes, as arguments, a CurrencySymbol (which has to be a hex value), a TokenName (which can be a string literal) and an Integer (which is the number of tokens) and it will return a Value, which if you remember correctly, is a mapping, similar to what you might see when you're running an indexing algorithm (CurrencySymbol -> (TokenName, NumberOfTokens)).

- Why do we need a currency symbol + token name?
- Why don't we just use a single identifier?
- Why does the currency symbol need to be built from hexadecimal?
- This is where minting policies come in.
- In general: a Tx cannot create a delete tokens.
- Everything goes in, comes out
- exception: fees
- fees depend on the size of the transaction (in bytes, not in value) and also the size of the scripts that need to be run to validate the transaction (space complexity, I believe? General computational complexity, I would have thought, O(x) | x ~ fees)
- If this was true for all cases, then we could never create native tokens
- Hence: Minting Policies & relevance of currency sybol
- reason why currency symbol bytestring needs to be hexadec is because it's the hash of a script
- this scipt is called the mintng policy
- if we have a tx where we want to create or burn native tkens, then
- for each native token we want to create or burn
- the currency symbol is looked up as a hash of the script
- so the co-responding script must be contained within the transaction
- that script is executed + other validation scripts
- simiar to validation scrpts that validate inputs
- this script needs to know whether the Tx has the right to mint or burn tokens
- and since ada has no currency symbol, there is no hash, there is no script, there is no minting, there is no burning.
- all ada that exists comes from genesis block
- total amount of ada in the system is fixxed
- only custom native tokens can have custom minting policies and can be minted + burned under certain conditions
- Next we'll look at a minting policy script, it's similar to validation script, but not identical