

1. Lecture One: Introduction

Within this set of notes the following information will be presented. Firstly, the courses administration details are provided. Secondly, a coverage of the course content is outlined. Shortly thereafter, details about Plutus, the difficulty of the program and the (pseudo-optional) pre-requisites are given. Furthermore, an extensive explanation of the (E)UTxO model (which facilitates the possibility of transactions on the Cardano network - **Cardano's unique accounting model**) is fully explained. In addition, building an example contract is presented, which includes how to setup a NIX shell, how to start plutus-playground-server and enable it to be locally accessible from the browser. Smart Contract compilation is demonstrated and an example of how to use the plutus platform in whole is shown. As pioneers we are then encouraged to try this ourselves.

Note: this document contains information pulled in from the Plutus Pioneer Lecture series in addition to information provided by reliable sources, such as videos, papers and statements made by employees of IOHK. Furthermore, to provide some background information on models such as UTxO (unextended), other academic sources (papers and books) have also been referenced.

1.1 Administration

- Lectures on every Thursday.
- Q&A Sessions Every Tuesday.
- Occasional Guest Lectures.
- Use the Discord server (and slack) to help one another when possible.

1.2 Course Coverage

The Plutus Platform:

- Smart Contracts on Cardano via (E)UTxO
- Writing Smart Contracts with Haskell (Local, Production)
- Compiling Haskell to Plutus-core using Plutus-Tx
- Details Surrounding Plutus-core (GHC Plug-in)
- Plutus-Tx Compiling Details (GHC Core): Intermediary Languages / Intermediary Representations

Further Underlying Concepts

Smart Contracts In Detail (and their various forms)

Testing and Implementing Smart Contracts

- Using the Plutus Playground
- Offline Locally

Native Tokens On Cardano

- Minting Native Tokens
- Burning Native Tokens
- Use of Native Tokens Within Smart Contracts

Deploying Plutus Contracts

Writing Backends for Plutus Contracts

1.3 Plutus Platform

"Plutus Platform Learning is Difficult" - Lars Brünjes

Why So Difficult?

- Using (E)UTxO Model - Less intuitive than other similar technological implementations (E.G. ETH Accounting Model).
- Plutus is brand new and under rapid development, thus is changing all the time.
- Due to constant changes, we're required to regularly update project dependencies.
- Tooling is not ideal...
- Difficult To Access Syntax & Repl Docs.
- Difficult to Build Plutus (It's likely best to use NIX)
- The Docker Image for the Plutus Platform is not yet ready.
- Plutus-core is compiled down from Haskell, Haskell is fairly difficult.
- It is recommended that you spend 40 hours per week, for 10 weeks to gain a solid understanding of Haskell before or whilst undertaking this course.
- Plutus is BRAND new - we are the first people **ever in the world to write plutus code**.
- This means: no quick answers from stackoverflow or google...

However,

- Haskell courses and documentation has been made available to help pioneers learn.

2. The (E)UTxO Model

“Any fool can know. The point is to understand.” — Albert Einstein [1]

2.1 What is (E)UTxO?

(E)UTxO abbreviates: "Extended Unspent Transaction Output", but to understand the extended model, we must first examine the unextended model (UTxO, firstly implemented by BitCoin [2]).

2.2 UTxO Explained

"If you want to get money, you have to consume an output that is laying around and in turn you get more outputs." — Michael Peyton-Jones

What are unspent transaction outputs? Simply put: They are outputs - which can be thought of as remaining financial change - from previous transactions on the Blockchain that are still currently 'unspent' - Imagine you go to a shop, you buy a drink, you give the shopkeeper a £20 note, and you are given £19 back in change. That £19 is an unspent transaction output from the original transaction of the purchase of a drink.

UTxO is a model of accounting and is used to identify how much 'money' (in this case: a digital 'currency') any 'wallet' ¹ on a blockchain data structure contains within it ². Each wallet address on a permissionless blockchain such as BTC is associated to a set of cryptographic 'tools' which lock any funds (BTC, Digital Currency) that are tied to the above mentioned wallet. Spending funds (e.g. sending BTC from one wallet to another) equates to the transfer of ownership of BTC from one private key owner to another ³. This is accomplished by signing a transaction with the initiators private key, which can only be confirmed by verifying the signature using the initiators public key.

It is important to note that you can only use complete UTxOs as input. So, if Alice has 100 BTC and wishes to send 10 BTC to bob, Alice creates a transaction which consumes 100 BTC as input and creates two outputs, 90 BTC (for Alice) and 10 BTC (for Bob). This is similar to the example mentioned above regarding the use of a £20 note.

You would therefore think that the sum of all input for any give Tx must equal the output, such that:

$$\sum_{i=0}^X Tx_i = \sum_{o=0}^Y Tx_o$$

However, this is not the case. Firstly, transaction fees must be considered. *Furthermore, on Cardano (that implements the (E)UTxO model, uses ADA and not BTC) native tokens have*

been rolled out. Thus, minting and burning of native tokens creates an imbalance between the cumulative inputs and resulting outputs of a transaction. However, this is content for later lecture (according to Lars).

Note: This is all accomplished through the use of a software wallets implementing transaction algorithms [3](#). Furthermore, technically BTC does implement basic 'smart contracts', which is to say, the transactional algorithms may utilise BitCoin Script (which includes a validator for identifying addresses and a redeemer to unlock the ability to spend funds[\[5\]](#)). As far as I understand, BitCoin Script is essentially a wrapper for managing public-private key cryptographic implementation and unlocking funds to spend through the use of digital signatures [\[6\]](#).

2.3 Advanced Transactions Using UTxO

The extended implementation of UTxO as proposed by IOHK, developed and implemented by Cardano is much more powerful than BitCoins UTxO model. However, there exists 'advanced' payment system elements implemented by 'basic' UTxO. It is important to understand how multiple unspent transaction outputs from different wallets (essentially: different owners of BTC) can combine their UTxOs together to create a transaction output to be sent to an individual address.

Consider the scenario where Bob has 60 BTC (in the form of two UTxOs: 50 BTC and 10 BTC) and Alice has 90 BTC in the form of a single UTxO. However, they wish to send 110 BTC to Tom. Alice must consume her (only) UTxO and Bob must consume both his UTxOs to meet the required amount to be sent to Tom (creating three transaction inputs from two owners with three unspent transaction outputs).

Within this 'simple advanced' transaction, the outputs generated by the initiated transaction ensures Tom receives 110 BTC and all other parties (Alice and Bob) receive their change by splitting the resulting output such that they receive the following [\[7\]](#):

$$\text{Given : } Tx^{Alice} = 90 \text{ BTC and } \sum_{i=0}^X Tx_i^{Bob} = 60 \text{ BTC}$$

Alice And Bob Pay Tom 110 BTC | Thus Change Received By Alice and Bob

=

$$\left(\text{Alice} \mid Tx^{Alice} - (110 \div 2) \right) = 35 \text{ BTC}$$

$$\left(\text{Bob} \mid \left[\sum_{i=0}^X Tx_i^{Bob} \right] - (110 \div 2) \right) = 5 \text{ BTC}$$

2.4 And God Said Let There Be (E)UTxO

Cardano implements an extended model of UTxO. The fundamentals remain the same. Thus, transactions are made up of numerous inputs, which themselves are unspent transaction outputs. However, there are some modifications to the model which are important. These modifications allow for more general transactions through the use of arbitrary logic.

Firstly, Cardano implements addresses differently to other cryptocurrencies (blockchains and distributed ledger technologies). Instead of using a simple hash-based derived address (using a public key as input), addresses in Cardano are generated using a derivation scheme outlined [here](#).⁴ These addresses are referential in nature and point to scripts containing arbitrary logic. These scripts are programmes which are (ideally) deterministic, pure, replayable state-machines. Somewhat similar to the aforementioned *validators*, but are far more general in nature. They are, however, still used to determine whether a given UTxO may be consumed based on the *redeemer* input, which is now also arbitrary in nature and has replaced simple digital signature based input.

Secondly, transaction outputs also contains a component called *datum* (*which is the singular noun for data*). *Datum* is useful for identifying state. This means that the (E)UTxO is at least as powerful as the ETH model.

(E)UTxO Advantage: Validation of UTxO Off-Chain

It is possible to check that a transaction will validate within your wallet before ever sending it to the chain.

However, Simultaneous UTxO Consumption Is Possible

Unspent transaction outputs can be consumed by others before your transaction has the opportunity to reach the chain (after being verified by your wallet software). In this instance, your transaction will simply be dropped back into the pool of transactions to be verified (so long as all the inputs are still there) and attempts to verify your transaction will continue.

Scope: (E)UTxO Model Scripts, Redeemers and Datum

The implementation of scripts within the (E)UTxO model ensures that the scope is limited to the transaction (or a small chain of transactions) that are attempting to be facilitated by a user. This means the state of the entire blockchain does not need to be known (as is the case with ETH). Furthermore, there is an appropriate degree of scope that enables the ability to create smart contracts which are (ideally) deterministic in nature, meaning: the output is predictable. This makes it more difficult to introduce bugs into the software.

3. Plutus Platform, Smart Contracts & Nix

During the first week of this program, we have been asked to:

- Clone The Plutus Pioneer Repo
- Clone The Plutus Repo
- Build Plutus (Using Nix, Cabal, The Docker Image Is Not Yet Ready)
- Set Up The Plutus Binary Cache Appropriately (Otherwise It'll Take Forever To Build)
- Start A Nix-Shell
- Start The Plutus Playground Server Locally
- Run NPM To Allow Access To The Playground Via A Local Server
- Copy & Paste A Complicated Haskell Smart Contract Into The Playground
- Compile The Smart Contract
- Interface With The Smart Contract Via A Web Browser
- Apply Some Changes To The Contract
- Play Around With The Interface Via The Browser

3.1 Cloning Repos

I would hope that if you were enrolled in this program, you can skip this chapter. But, in the interest of being exhaustive the process of cloning the repos is outlined below.

I typically like to place all my projects within the following directory:

```
~/code/.
```

Navigate and clone the appropriate projects from IOHK:

```
cd ~/code/  
git clone git@github.com:input-output-hk/plutus-pioneer-program.git  
git clone git@github.com:input-output-hk/plutus.git
```

Congratulations, we're on your way to getting started!

3.2 Building Plutus | Installing Nix (MacOS) | Cabal | Nix-Shell

In order to build Plutus you're most likely going to need Nix (you can install Nix within a VM using nixOS, or you can install on MacOS or Windows. Be aware on MacOS you need xtools and potentially various other dependencies. Installing via Homebrew was suggested to me, but I found simply building the binaries easier.

Navigate to your home directory:

```
cd ~/.
```

Download the latest version of Nix for MacOS and pipe into sh to start the install script:

```
curl -L https://nixos.org/nix/install | sh
```

Note: ensure you have root priv-access.

Add the binary folder to your global PATH variable (note: I'm using zshell):

```
sudo vim ~/.zshrc  
export PATH="$HOME/.nix-profile/bin:$PATH"
```

Ensure you're using the current Plutus build by checking the .cabal-project file in the current weeks code folder.

Ensure the Nix binary cache is setup:

```
sudo vim /etc/nix/nix.conf  
substituters      = https://hydra.iohk.io https://iohk.cachix.org https://cache.nixos.org/  
trusted-public-keys = hydra.iohk.io:f/Ea+s+dFdN+3Y/G+FDgSq+a5NEWhJGzdjvKNGv0/EQ= iohk.cachix.org-1:DpRUyj7h7V830dp/
```

Check the current builds project hash, it's displayed via the 'tag', now checkout to that branch in the Plutus github repo and begin a nix-shell:

```
index-state: 2021-06-10T00:00:00Z

packages: ./

-- You never, ever, want this.
write-ghc-environment-files: never

-- Always build tests and benchmarks.
tests: true
benchmarks: true

source-repository-package
  type: git
  location: https://github.com/input-output-hk/plutus.git
  subdir:
    freer-extras
    playground-common
    plutus-chain-index
    plutus-core
    plutus-contract
    plutus-ledger
    plutus-ledger-api
    plutus-tx
    plutus-tx-plugin
    prettyprinter-configurable
    quickcheck-dynamic
    word-array
  tag: ea0ca4e9f9821a9dbfc5255fa0f42b6f2b3887c4
```

```
cd ~/code/plutus-pioneer-program
cd ~/code/plutus
git checkout ea0ca4e9f9821a9dbfc5255fa0f42b6f2b3887c4
```

Change directory back to week one of the program and build the project using cabal:

```
cd ~/code/plutus-pioneer-program
cd code/week01
cabal build
```

This may take some time, but be patient! When it's done, change directory to the plutus

playground client folder and start the plutus playground server, open a new nix-shell and then npm start to be able to view the application in the browser:

```
cd ~/code/plutus/plutus-playground-client
plutus-playground-server
// new shell window
nix-shell
cd ~/code/plutus/plutus-playground-client
npm start
```

This May Also Take Some Time...

But, you should end up with something like this, where the playground is being hosted locally @ localhost, port 8009:

```
The import of module Halogen.HTML contains the following unused references:
```

```
div_
```

```
It could be replaced with:
```

```
import Halogen.HTML (ClassName(..), HTML, br_, button, div, p_, text)
```

```
[239/239 UnusedExplicitImport] web-common/src/LocalStorage.purs:21:1
```

```
21 import Effect.Aff (Aff, Canceler, effectCanceler, makeAff)
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
The import of module Effect.Aff contains the following unused references:
```

```
Canceler
```

```
It could be replaced with:
```

```
import Effect.Aff (Aff, effectCanceler, makeAff)
```

	Src	Lib	All
Warnings	239	0	239
Errors	0	0	0

```
[info] Build succeeded.
```


```
> plutus-playground-client@1.0.0 webpack:server /Users/jonathondilworth/code/plutus/plutus-playground-client
> webpack-cli serve --progress --inline --hot --mode=development --node-env=development
```

```
1% setup initialize: [wds]: Generating SSL Certificate
10% building 0/1 entries 0/0 dependencies 0/0 modules: [wds]: Project is running at https://localhost:8009/
i [wds]: webpack output is served from /
i [wds]: Content not from webpack is served from /Users/jonathondilworth/code/plutus/plutus-playground-client/dist
△ [wdm]: WARNING in Src Lib All
Warnings    0      0      0
Errors      0      0      0
```

```
70 WARNINGS in child compilations (Use 'stats.children: true' resp. '--stats-children' for more details)
webpack compiled with 71 warnings
i [wdm]: Compiled with warnings.
```

So, head on over to localhost @ port 8009 (<http://localhost:8009>). It looks so pretty! Doesn't it?

PLUTUS REFRESH - UPDATED 25TH JANUARY 2021

 PLUTUS PLAYGROUND

Getting StartedTutorialsAPIPrivacy

Demo filesHello.worldStarterGameVestingCrowd FundingError Handling

Login

Editor

Key BindingsDefault

CompileSimulate

```
1 -- Vesting scheme as a PLC contract
2 import Control.Monad      (void, when)
3 import qualified Data.Map  as Map
4 import qualified Data.Text  as T
5
6 import Ledger              (Address, POSIXTime, POSIXTimeRange, PubKeyHash, Validator)
7 import qualified Ledger    as Ledger
8 import qualified Ledger.Ada as Ada
9 import Ledger.Constraints  (TxConstraints, mustBeSignedBy, mustPayToTheScript, mustValidateIn)
10 import Ledger.Contexts    (ScriptContext (...), TxInfo (...))
11 import qualified Ledger.Contexts as Validation
12 import qualified Ledger.Interval as Interval
13 import qualified Ledger.TimeSlot as TimeSlot
14 import qualified Ledger.Tx      as Tx
15 import qualified Ledger.Typed.Scripts as Scripts
16 import Ledger.Value           (Value)
17 import qualified Ledger.Value   as Value
18 import Playground.Contract    as Value
19 import Plutus.Contract        as Value
20 import qualified Plutus.Contract.Typed.Tx as Typed
21 import qualified PlutusTx      as PlutusTx
22 import PlutusTx.Prelude       hiding (Semigroup (...), fold)
23 import Prelude                as Haskell (Semigroup (...), show)
24 import Wallet.Emulator.Types  (walletPubKey)
25
26 {- |
27   A simple vesting scheme. Money is locked by a contract and may only be
28   retrieved after some time has passed.
29
30   This is our first example of a contract that covers multiple transactions,
31   with a contract state that changes over time.
32
33   In our vesting scheme the money will be released in two _tranches_ (parts):
34   A smaller part will be available after an initial number of time has
35   passed, and the entire amount will be released at the end. The owner of the
36   vesting scheme does not have to take out all the money at once: They can
37   take out any amount up to the total that has been released so far. The
38   remaining funds stay locked and can be retrieved later.
39 -}
```

Not compiled

cardano.orgiohk.io

© 2020 IOHK Ltd.

GitHubTwitterFeedback

Hit compile, and then hit simulate. You should see the interface to the Haskell program running as a smart contract on the plutus pioneer playground server (locally), you should see (and play around with the following):

Simulation 1

EvaluateTransactions

Wallets

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

Wallet 1

Opening Balances

Lovelace100

Available functions

retrieve funds +vest funds +Pay to Wallet +

Wallet 2

Opening Balances

Lovelace1000

Available functions

retrieve funds +vest funds +Pay to Wallet +

Wallet 3

Opening Balances

Lovelace1000

Available functions

retrieve funds +vest funds +Pay to Wallet +

+ Add Wallet

Actions

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

1

Wallet 1: retrieve funds

Lovelace100

2

Wallet 2: retrieve funds

Lovelace1000

3

Wallet 1: Pay To Wallet

Recipient2

Amount10

Lovelace10

4

Wait

Wait For...Wait Until...

Blocks2

5

Wallet 3: vest funds

6

Wait

Wait For...Wait Until...

Blocks1

7

Wallet 1: Pay To Wallet

Recipient3

Amount25

Lovelace25

8

Wait

Wait For...Wait Until...

Slot7

9

Wallet 2: retrieve funds

Lovelace2000

+ Add Wait Action

EvaluateTransactions

PLUTUS REFRESH - UPDATED 25TH JANUARY 2021

PLUTUS PLAYGROUND

Getting Started Tutorials API Privacy

Demo files Hello_world Starter Game Vesting Crowd.Funding Error Handling

Log In

Simulator

Simulation 1 +

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0 Slot 1, Tx 0 Slot 4, Tx 0

Inputs

Wallet 1
PubKeyHash 21fe31dfa154a261626bf854046d2271b7...
Ada Lovelace 80
Created by: Slot 1, Tx 0

Transaction

Slot 4, Tx 0

Tx: a5e625cde0d533a579c352bcdc070ad3c1f88ee5a3f521f748de477a1463f81d
Validity: All time
Signatures:
• PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

Outputs

Fee
Ada Lovelace 10

Wallet 1
PubKeyHash 21fe31dfa154a261626bf854046d2271b7...
Ada Lovelace 45
Unspent

Wallet 3
PubKeyHash dac073e0123bdea59d9b3bda9c603776...
Ada Lovelace 25
Unspent

4. Exercise: Auction

During the first week of the Plutus program the only real 'homework' we were assigned was to get the playground up and running via Nix, mess about with it and simulate an NFT auction. This was all shown to us during the lecture, so it's pretty easy to follow.

1. Open a new nix-shell (in ~/code/plutus).
2. Change directory to ~/code/plutus-pioneer-program/code/week01
3. Run cabal repl
4. Now you can run handy commands such as: import Ledger.TimeSlot (as you'll need this) + others.

Now, you're going to need the EnglishAuction.hs script, which can be found @

```
~/code/plutus-pioneer-program/code/week01/src/week01/EnglishAuction.hs
```

Copy and paste this into the Plutus Playground within your browser and hit compile. You should no be able to simulate something that looks like the following:

Simulation 1

+

[< Return to Editor](#)

Wallets

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

Wallet 1

×

Opening Balances

Lovelace

1000000

T

1

Available functions

bid

+

close

+

start

+

Pay to Wallet

+

Wallet 2

×

Opening Balances

Lovelace

100000

T

0

Available functions

bid

+

close

+

start

+

Pay to Wallet

+

Wallet 3

×

Opening Balances

Lovelace

1000000

T

0

Available functions

bid

+

close

+

start

+

Pay to Wallet

+

+

Add Wallet

Actions

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

1

×

Wallet 1: start

spDeadline

getPOSIXTime

Integer

Required

spMinBid

Integer

Required

spCurrency

unCurrencySymbol

String

✓

spToken

+

Add Wait Action

4.1 Simulating

As an NFT auction, we're using the T parameter in the wallets to represent an NFT. Go ahead and add an additional wallet and add some ADA to them. Now, when you go to add the start of the auction, you'll require an expiry time. The problem is, time is measured in POSIXTime. However, you can open the repl and run the following:

```
import Ledger.TimeSlot
slotToPOSIXTime 10
```

This generates a time value for slot 10, as is shown below:

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

Now you can start filling in the auction that we're about to simulate (I used some random values for ADA, etc, so the number of my Tx's may be different to yours, but typically you should have at least three or so, with the first being the genesis transaction, which fills individuals wallets:

Transaction, Slot Zero

Transaction, Slot One

Simulator < Return to Editor

Simulation 1 +

Transactions

Blockchain

Click a transaction for details

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

Inputs

Wallet 1
PubKeyHash 21fe31df8154a261626b7854046d2271b7bed4b6be45aa588...
Ada Lovelace 1,000,000
66 T 1
Created by: Slot 0, Tx 0

Transaction

Slot 1, Tx 0

Tx: d5c7b056c0f3act770737dfc700889991cb78a4967bb1bc792e983b0d95500
Validity: All time
Signatures:
• PubKey d75a980182b10ab7d54bfec3c964073a0ee172f3daa62325af021a68f707511a

Outputs

Fee
Ada Lovelace 10

Wallet 1
PubKeyHash 21fe31df8154a261626b7854046d2271b7bed4b6be45aa588...
Ada Lovelace 999,990
66 T 0
Spent in: Slot 1, Tx 0

Script aff3035b5ba64d5d6805efb94b97ee568eb1c6eb70d8...
66 T 1
Unspent

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

Beneficial Owner	Ada	66
Wallet 1 PubKeyHash 21fe31df8154a261626b7854046d2271b7bed4b6be45aa588... Lovelace	999,990	0
Wallet 2 PubKeyHash 397f13d96a44233042942169516b6089793d62939594f9399a461795139f T	100,000	0
Wallet 3 PubKeyHash da0779e7123bde959b93b0d937763a83271f78c0544c29674020c T	100,000,000	0
Script aff3035b5ba64d5d6805efb94b97ee568eb1c6eb70d8... T	0	1

Transaction, Slot Two

Simulator < Return to Editor

Simulation 1 +

Transactions

Blockchain

Click a transaction for details

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

Inputs

Wallet 1
PubKeyHash 21fe31df8154a261626b7854046d2271b7bed4b6be45aa588...
Ada Lovelace 999,990
66 T 0
Created by: Slot 1, Tx 0

Script aff3035b5ba64d5d6805efb94b97ee568eb1c6eb70d8...
66 T 1
Created by: Slot 1, Tx 0

Transaction

Slot 1, Tx 0

Tx: d1bab31f2d548d64a78554380f99ce21c374cd39d0bd9f96453c80f6c9445
Validity: From Slot 10 (inclusive) to the end of time (inclusive)
Signatures:
• PubKey d75a980182b10ab7d54bfec3c964073a0ee172f3daa62325af021a68f707511a

Outputs

Fee
Ada Lovelace 13,573

Wallet 1
PubKeyHash 21fe31df8154a261626b7854046d2271b7bed4b6be45aa588...
Ada Lovelace 986,417
66 T 0
Unspent

Wallet 1
PubKeyHash 21fe31df8154a261626b7854046d2271b7bed4b6be45aa588...
66 T 1
Unspent

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

Beneficial Owner	Ada	66
Wallet 1 PubKeyHash 21fe31df8154a261626b7854046d2271b7bed4b6be45aa588... Lovelace	986,417	1
Wallet 2 PubKeyHash 397f13d96a44233042942169516b6089793d62939594f9399a461795139f T	100,000	0
Wallet 3 PubKeyHash da0779e7123bde959b93b0d937763a83271f78c0544c29674020c T	100,000,000	0
Script aff3035b5ba64d5d6805efb94b97ee568eb1c6eb70d8... T	0	0

4.2 Simulation Issue (Resolved)

So, I noticed the NFT was not transferring from wallet one to wallet three. It turns out I hadn't put enough ADA into wallet three to bid for the NFT in addition to paying the transaction fee. All in all, good learning experience. See below:

Transactions

Blockchain

Click a transaction for details

Slot 0, Tx 0

Slot 1, Tx 0

Slot 2, Tx 0

Slot 3, Tx 0

Slot 11, Tx 0

Inputs

Wallet 1

PubKeyHash 21f631dfa154a261626bfb54046d2271b7bed4babe45aa588...

Ada

Lovelace

999,990

66

T

0

Created by: Slot 1, Tx 0

Script aff3035b5ba64d5d6805efb94b97ee568eb1c6eb70d8...

66

T

1

Ada

Lovelace

10,000,000

Created by: Slot 3, Tx 0

Transaction

Slot 11, Tx 0

Tx: bba2e9419aef163065a3689b1ced01aa3dbbe617878e94c57bed517ce1acc

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

Signatures:

- PubKey d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a688707511a

Outputs

Fee

Ada

Lovelace

13,784

Wallet 1

PubKeyHash 21f631dfa154a261626bfb54046d2271b7bed4babe45aa588...

Ada

Lovelace

986,206

66

T

0

Unspent

Wallet 1

PubKeyHash 21f631dfa154a261626bfb54046d2271b7bed4babe45aa588...

Ada

Lovelace

10,000,000

Unspent

Wallet 3

PubKeyHash d4c073e0123bdea9dd9b3bd9c76037b3aca82627d7abcd5c...

66

T

1

Unspent

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

Beneficial Owner	Ada	66
Wallet 1		
PubKeyHash 21f631dfa154a261626bfb54046d2271b7bed4babe45aa5887d407972109	10,986,206	0
Wallet 2		
PubKeyHash 39f138f0a44c239a629421b9f51b9b08975b9c93995c4f5990ee1775139f	85,871	0
Wallet 3		
PubKeyHash d4c073e0123bdea9dd9b3bd9c76037b3aca82627d7abcd5c...	89,985,660	1

5. Possible Problems & Uncertainties

The Following Paragraphs are sections that I removed, as I wasn't 100% sure as to whether they were accurate or not.

Firstly, the *validator* in Cardano is extended to also implement the aforementioned arbitrary logic and is no longer a simple transactional algorithm which returns a boolean value based on the input from the *redeemer*. Therefore, the *validator* is a programmable.

Furthermore, the *validator* ultimately remains unmodified (recall that the *validator* determines the address of the transaction output. Thus, it's not a great idea to add too much complexity to the on-chain validation script).

(From the video by Michael Peyton Jones)

As you have most likely identified by now, the *validator* is required to (rather self-evidently) validate the entire transaction on-chain. In order for it to perform this function, it must have the appropriate scope such that it can evaluate the entire context of the transaction. This is why an additional component is added to all Cardano transactions called *context*. The *validator* can now see all inputs and outputs for the transaction that it is validating and can be re-used within a chain of potential transactions. A chain of transactions can be thought of as a state machine.

(Information on Context and State Machines which may be outdated? They were not in the first lecture)

Secondly, transaction outputs also contains a component called *datum* (which is the singular

noun for data). *Datum* is useful for identifying state (this is important for the implementation of state machines, as is explained in the next paragraph).

(Again, this may be useful later, but it wasn't really present in the first lecture, state machines that is)

References

- [1] Simmons, G.F., 2003. Precalculus mathematics in a nutshell: geometry, algebra, trigonometry. Wipf and Stock Publishers.
- [2] Antonopoulos, A.M., 2014. Mastering Bitcoin: unlocking digital cryptocurrencies. O'Reilly Media, Inc.
- [3] Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review, p.21260.
- [4] Pardalos, P., Kotsireas, I., Guo, Y. and Knottenbelt, W., 2020. Mathematical Research for Blockchain Economy. Springer International Publishing.
- [5] Zahnentferner, J. and HK, I.O., 2018. An Abstract Model of UTxO-based Cryptocurrencies with Scripts. IACR Cryptol. ePrint Arch., 2018, p.469.
- [6] Pérez-Solà, C., Delgado-Segura, S., Navarro-Arribas, G. and Herrera-Joancomartí, J., 2019. Double-spending prevention for bitcoin zeroconfirmation transactions. International Journal of Information Security, 18(4), pp.451-463.
- [7] Antonopoulos, A.M., 2017. Mastering Bitcoin: Programming the open blockchain. O'Reilly Media, Inc.

Footnotes

1. The use of the word: wallet can be misleading (and this may be why UTxO models are not hugely intuitive). A wallet is simply the stored set of cryptographic 'elements' generated through the use of ECC Public-Private Key cryptography [2] and hashing algorithms. In the case of Bitcoin, a user generates a public-private key pair, and their 'wallet' address is generated using the following hashing algorithm: $ADDR = RIPEMD160(SHA256(K))$, where K represents their public key and is then shortened further using additional functions that you can read about [here](#). These keys (or 'cryptographic elements') are stored within their wallet, which, if they are managing themselves (and not through the use of a central exchange or central online wallet) are kept offline (and are always off-chain). However, wallet addresses (which are tied to a wallet computer program) are available on-chain (they have to be in order to initiate a Tx).

2. Wallets can contain hundreds of public-private key pairs and the owner for X amount of BTC can in turn have hundreds of unspent transaction outputs. However, it must be noted that when such a user creates a new transaction, the set of their unspent transaction outputs will be used as inputs into the following transaction.
3. Transaction algorithms such as: P2PKH: "Pay To Public Key Hash", P2PK: "Pay To Public Key", P2SH: "Pay To Script Hash", P2WPKH: "Pay To Witness Public Key Hash" [4].
4. You can view the implementation of addresses here: <https://github.com/input-output-hk/cardano-addresses/blob/master/core/lib/Cardano/Address.hs>