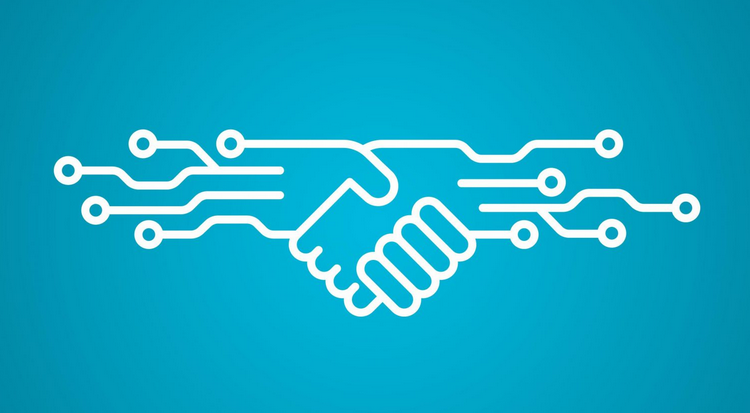
**Plutus: Learning a**

**smart contract language**



**IOG**

**Plutus: Learning a smart contract language**

Copyright © March 2023, IOG (Input Output Global)

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

This book was prepared by IOGs education team. Majority of the text is taken from the Plutus pioneer program 4rd iteration that was presented by Lars Brünjes and his colleagues. All program code in this book is taken from the Plutus pioneer program and is freely available at:

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

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

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

Table of Contents

[1 Plutus introduction 5](#_Toc128476795)

[1.1 Setting up your development environment 5](#_Toc128476796)

[1.1.1 Using Demeter.run 5](#_Toc128476797)

[1.1.2 Using Docker 5](#_Toc128476798)

[1.2 Kuber marketplace 6](#_Toc128476799)

[1.3 Hashing and digital signatures 6](#_Toc128476800)

[1.4 The EUTxO model 6](#_Toc128476801)

[1.5 Plutus code 8](#_Toc128476802)

[2 Resources 10](#_Toc128476803)

Table of Figures

Figure 1 - UTXO model [1] 6

# Plutus introduction

Plutus is the native smart contract language for Cardano. It is a Turing-complete language written mostly in Haskell, and Plutus smart contracts are effectively Haskell programs. By using Plutus, you can be confident in the correct execution of your smart contracts. It draws from modern language research to provide a safe, full-stack programming environment based on Haskell, the leading purely functional programming language. [1]

So, in order to understand the code in this book and one must understand the basics of the Haskell programming language. The Haskell project contains a list of learning resources as books and tutorials under its Documentation section, from which some are free and some commercial. It can be found at Haskell’s official web page:

* <https://www.haskell.org/documentation/>

|  |  |
| --- | --- |
|  | IOG provides its own Haskell course that is tailored to the needs of learning Marlowe, also a smart contract langauage and Plutus. It can be found at IOGs GitHub page:  <https://github.com/input-output-hk/haskell-course> |

If you would like to learn more about Plutus and its software development kit (SDK) in addition to what this book offers, you can check out the online documentation which can be found at:

* <https://plutus.readthedocs.io/en/latest/>
* <https://plutus-apps.readthedocs.io/en/latest/>

## Setting up your development environment

The installation instructions for setting your development environment are split in two parts. The first part is using the online platform *Demeter.run* and the second part is setting up a Docker container locally on your PC. The second instructions are general and can be used on Windows, Mac OS and Linux operating systems.

### Using Demeter.run

### Using Docker

## Kuber marketplace

## Hashing and digital signatures

## The EUTxO model

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

Diagram

Description automatically generated

Figure 1 - UTXO model [1]

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

As soon as an output is used as input in a transaction, it becomes spent and can never be used again. The UTXO output is associated to an address which is represented by a public key hash. We call them public key addresses. The ADA amount and optionally native tokens of an public key address is the sum of ADA and native tokens from all UTXOs belonging to this address. A transaction must be signed by the owner of the private key corresponding to the address that defines the input UTXO. Think of an address as a ‘lock’ that can only be ‘unlocked’ by the right ‘key’ ‒ the correct signature. The user which controls a private key of an address is able to create transactions and use the ADA or native tokens sitting at the UTXOs of this address.

The extended UTXO model introduces in addition to public key addresses also script addresses that can contain some logic. That logic defines under which conditions the UTXOs sitting at this address can be spent. The address is unlocked by, and piece of data called the *redeemer*, which in the conventional UTXO model would be a private key. A UTXO also contains some data called the *datum*, beside the amount of ADA sitting at the address. The datum together with the redeemer and the transaction context are the input information for a script logic that then chooses to weather this transaction is valid and can be processed by a node on the network.

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

A transaction can be classified as a producing transaction that produces UTXOs or as a spending transaction that spends UTXOs. The truth is every transaction except the genesis transaction takes at least one UTXO as input and produces at least one UTXO as output. What we mean by the terms spending and producing is the context. Later in chapter 2 we will see code examples where we first send ADA to a script address and for this reason, we call it a producing transaction. And after that we try to collect that ADA from the script address so we call this transaction a spending transaction.

The script address is defined as a hash of the validator code written in Plutus core language, which also needs to be provided. The script addresses are publicly known. The producing transaction must include this address, and it must include the datum or the hash of the datum that will be attached to the UTXO created at script address. If it includes the hash of the datum, only a person that knows the datum by some other means not by looking at the blockchain is able to ever spend such an UTXO. The spending transaction is responsible for providing the datum, the redeemer and the transaction context. It also provides the validator script. If we construct a transaction where the funds go to a public key address a datum is not needed.

As said the validator script takes the datum, the redeemer and the transaction context as input information. The input for the datum is collected from each UTXO individually that are sitting at a script address. That means if there are multiple UTXOs specified to be consumed in the transaction, the validation logic is checked for each of them separately. So, in each validation there is only one datum as input coming from one UTXO.

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

## Plutus code

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

From the official documentation [1] we get the following description for Plutus Core:

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

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

The off-chain code basically constructs the transaction and submits it to the blockchain. Since both the on-chain and off-chain code are written in Haskell they can reside in one Haskell file while testing your code, which allows them to share code between them. Another option is also to construct the off-chain transaction with command line tools instead of compiling Haskell code. Cardano provides for this the *cardano-cli* command line tool. There are also community alternatives. Those tools will also be presented in later chapters of this book. In the 4th iteration of the Plutus pioneer program we will not write any off-chain code but use only the command line tools. If you want to see examples of off-chain code written in Haskell you can look at the videos of the 3rd PPP iteration or also read the accompanied book. However, some of the code examples may not work today since the Plutus libraries were upgraded in the meantime.

# Resources

[1] The Cardano documentation

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

<https://web.archive.org/web/20220704234049/https://docs.cardano.org/plutus/eutxo-explainer>