Smart Contracts

Plutus & Marlowe

Lars Brünjes

January 10 2020

Introduction

Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.

Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.
- Although both have been created to add smart-contract capabilities to Cardano, they could in principle be used on other blockchains as well.

Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.
- Although both have been created to add smart-contract capabilities to Cardano, they could in principle be used on other blockchains as well.
- Both are written in Haskell.

Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.
- Although both have been created to add smart-contract capabilities to Cardano, they could in principle be used on other blockchains as well.
- Both are written in Haskell.
- Plutus is Turing-complete and general, Marlowe is a *non*-Turing-complete DSL for financial contracts.

Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.

Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.
- In the account-based model, permanently changing account balances constitute mutable state, whereas outputs in the UTxO-model are immutable.

Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.
- In the account-based model, permanently changing account balances constitute mutable state, whereas outputs in the UTxO-model are immutable.
- This is similar to the relationship between imperative programming languages (like Java and Python) and functional languages (like Haskell).

Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.
- In the account-based model, permanently changing account balances constitute mutable state, whereas outputs in the UTxO-model are immutable.
- This is similar to the relationship between imperative programming languages (like Java and Python) and functional languages (like Haskell).
- This makes it a good fit to write smart contracts for an account-based blockchain like Ethereum in an imperative language like Solidity and contracts for the UTxO-based Cardano in the functional language Haskell.

Plutus

Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds "smart" capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.

Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds "smart" capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called data scripts, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex state to outputs, which is not possible in Bitcoin.

Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds “smart” capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called data scripts, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex state to outputs, which is not possible in Bitcoin.
- For validation, input-, output- and data-scripts are combined with the transaction which is being validated, so both the state from the data scripts and the transaction itself (with all its inputs and outputs) are available for scrutiny by the validation logic.

Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds "smart" capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called data scripts, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex state to outputs, which is not possible in Bitcoin.
- For validation, input-, output- and data-scripts are combined with the transaction which is being validated, so both the state from the data scripts and the transaction itself (with all its inputs and outputs) are available for scrutiny by the validation logic.
- Plutus is also Turing-complete (and uses a similar system to the one Ethereum uses in order to prevent infinite loops).

Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds "smart" capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called data scripts, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex state to outputs, which is not possible in Bitcoin.
- For validation, input-, output- and data-scripts are combined with the transaction which is being validated, so both the state from the data scripts and the transaction itself (with all its inputs and outputs) are available for scrutiny by the validation logic.
- Plutus is also Turing-complete (and uses a similar system to the one Ethereum uses in order to prevent infinite loops).
- Taken together, these properties make it possible to write smart contracts in Plutus which are as least as powerful as Ethereum smart contracts.

Plutus Core

- The "machine language" underlying Plutus is called Plutus Core.

Plutus Core

- The "machine language" underlying Plutus is called Plutus Core.
- In contrast to Bitcoin Script and the EVM, Plutus Core is *not* stack based, but is instead System F_ω, a variant of the lambda calculus with a powerful type system, which also happens to be the foundation Haskell is built upon.

Plutus Core

- The "machine language" underlying Plutus is called Plutus Core.
- In contrast to Bitcoin Script and the EVM, Plutus Core is *not* stack based, but is instead System F_ω, a variant of the lambda calculus with a powerful type system, which also happens to be the foundation Haskell is built upon.
- Not only has Plutus been implemented in Haskell, you also program in Haskell, which means that Haskell is for Plutus Core what Solidity is for the EVM.

Plutus Core

- The "machine language" underlying Plutus is called Plutus Core.
- In contrast to Bitcoin Script and the EVM, Plutus Core is *not* stack based, but is instead System F_ω, a variant of the lambda calculus with a powerful type system, which also happens to be the foundation Haskell is built upon.
- Not only has Plutus been implemented in Haskell, you also program in Haskell, which means that Haskell is for Plutus Core what Solidity is for the EVM.
- This enables a seamless interplay between onchain code and offchain code (whereas Solidity only supports onchain code, so that offchain code has to be written in something like JavaScript.)

Lambda Calculus

Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.

Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.
- Turing complete.

Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.
- Turing complete.
- Makes two simplifications:
  - only anonymous functions
  - only functions of one argument (curried functions)

Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.
- Turing complete.
- Makes two simplifications:
  - only anonymous functions
  - only functions of one argument (curried functions)
- Haskell is based upon and compiles to a (typed!) version of the lambda Calculus (as first intermediate compiler target, Core).

History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.

History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.

History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.
- Shown to be logically inconsistent in 1935 by Stephen Kleene and J. B. Rosser.

History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.
- Shown to be logically inconsistent in 1935 by Stephen Kleene and J. B. Rosser.
- Fixed by Church in 1936 – Untyped Lambda Calculus.

History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.
- Shown to be logically inconsistent in 1935 by Stephen Kleene and J. B. Rosser.
- Fixed by Church in 1936 – Untyped Lambda Calculus.
- Relation to programming languages clarified in the 1960s.

Lambda expressions

Lambda expressions (or lambda terms) are composed of

- variables $v_1, v_2, \ldots, v_n, \ldots$,
- the abstraction symbols λ and $.$,
- parentheses $(\,)$.

The set of lambda expressions Λ is inductively defined as:

- If x is a variable, then $x \in \Lambda$. (variable)
- If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$. (lambda abstraction)
- If $M, N \in \Lambda$, then $(MN) \in \Lambda$. (application)

Free variables

- Let V be the set of variables. For each lambda expression $M \in \Lambda$, we define the set of free variables $FV(M) \subset V$ as follows:

Free variables

- Let V be the set of variables. For each lambda expression $M \in \Lambda$, we define the set of free variables $FV(M) \subset V$ as follows:
  - For a variable $x \in V$, $FV(x) = \{x\}$.

Free variables

- Let V be the set of variables. For each lambda expression $M \in \Lambda$, we define the set of free variables $FV(M) \subset V$ as follows:
  - For a variable $x \in V$, $FV(x) = \{x\}$.
  - For an abstraction, $FV(\lambda x.M) = FV(M) \setminus \{x\}$.

Free variables

- Let V be the set of variables. For each lambda expression $M \in \Lambda$, we define the set of free variables $FV(M) \subset V$ as follows:
  - For a variable $x \in V$, $FV(x) = \{x\}$.
  - For an abstraction, $FV(\lambda x.M) = FV(M) \setminus \{x\}$.
  - For an application, $FV(MN) = FV(M) \cup FV(N)$.

Free variables

- Let V be the set of variables. For each lambda expression $M \in \Lambda$, we define the set of free variables $FV(M) \subset V$ as follows:
  - For a variable $x \in V$, $FV(x) = \{x\}$.
  - For an abstraction, $FV(\lambda x.M) = FV(M) \setminus \{x\}$.
  - For an application, $FV(MN) = FV(M) \cup FV(N)$.
- Given a lambda expression $M \in \Lambda$, we call a variable $x \in V$ free (in M) if $x \in FV(M)$.

Free variables

- Let V be the set of variables. For each lambda expression $M \in \Lambda$, we define the set of free variables $FV(M) \subset V$ as follows:
  - For a variable $x \in V$, $FV(x) = \{x\}$.
  - For an abstraction, $FV(\lambda x.M) = FV(M) \setminus \{x\}$.
  - For an application, $FV(MN) = FV(M) \cup FV(N)$.
- Given a lambda expression $M \in \Lambda$, we call a variable $x \in V$ free (in M) if $x \in FV(M)$.
- In an abstraction $\lambda x.M$, we call the variable x bound.

β-Reduction

- Consider a lambda expression of the form $(\lambda x.M)\ N$, i.e. an application where the first argument is an abstraction.

β-Reduction

- Consider a lambda expression of the form $(\lambda x.M)\ N$, i.e. an application where the first argument is an abstraction.
- By definition, this term β-reduces to $M[x := N]$, the substitution of all (free) occurences of variable x in M by N (maybe after first renaming x to a variable which is not free in N).

β-Reduction

- Consider a lambda expression of the form $(\lambda x.M)\ N$, i.e. an application where the first argument is an abstraction.
- By definition, this term β-reduces to $M[x := N]$, the substitution of all (free) occurences of variable x in M by N (maybe after first renaming x to a variable which is not free in N).
- This act of “plugging in” an expression for the bound variable in an abstraction is what constitutes the idea of computation in lambda Calculus.

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n-times to x:

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n-times to x:
- $\texttt{zero} := \lambda fx.x.$

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n-times to x:
- $\mathtt{zero} := \lambda fx.x$.
- $\mathtt{succ} := \lambda nfx.f\ (nfx)$.

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n-times to x:
- $\texttt{zero} := \lambda fx.x.$
- $\texttt{succ} := \lambda nfx.f\ (nfx).$
- We can define addition: $\texttt{add} := \lambda mnfx.mf(nfx)$

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n-times to x:
- $\mathtt{zero} := \lambda fx.x.$
- $\mathtt{succ} := \lambda nfx.f\ (nfx).$
- We can define addition: $\mathtt{add} := \lambda mnfx.mf(nfx)$
- and multiplication: $\mathtt{mul} := \lambda mnfx.m(nf)x$

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n-times to x:
- $\texttt{zero} := \lambda fx.x$.
- $\texttt{succ} := \lambda nfx.f\ (nfx)$.
- We can define addition: $\texttt{add} := \lambda mnfx.mf(nfx)$
- and multiplication: $\texttt{mul} := \lambda mnfx.m(nf)x$
- and predecessor (more complicated!): $\texttt{pred} := \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$.

Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying f and x to a natural number is applying f n-times to x:
- $\texttt{zero} := \lambda fx.x.$
- $\texttt{succ} := \lambda nfx.f\ (nfx).$
- We can define addition: $\texttt{add} := \lambda mnfx.mf(nfx)$
- and multiplication: $\texttt{mul} := \lambda mnfx.m(nf)x$
- and predecessor (more complicated!): $\texttt{pred} := \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u).$
- and many more...

Typed Lambda Calculi

- The simply typed lambda calculus is a typed interpretation of the lambda calculus with only one type constructor "$\rightarrow$" that builds function types.

Typed Lambda Calculi

- The simply typed lambda calculus is a typed interpretation of the lambda calculus with only one type constructor "$\to$" that builds function types.
- System F is a typed lambda calculus that differs from the simply typed lambda calculus by the introduction of a mechanism of universal quantification over types (polymorphism).

$$\mathrm{id} := \Lambda a.\lambda(x : a).x : \forall a.a \to a$$

Typed Lambda Calculi

- The simply typed lambda calculus is a typed interpretation of the lambda calculus with only one type constructor "$\to$" that builds function types.
- System F is a typed lambda calculus that differs from the simply typed lambda calculus by the introduction of a mechanism of universal quantification over types (polymorphism).

$$\mathrm{id} := \Lambda a.\lambda(x : a).x : \forall a.a \to a$$

- System F_ω adds functions from types to types (type constructors) to System F.

Marlowe

Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.

Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.
- It is based on the seminal paper "Composing Contracts: An Adventure in Financial Engineering" by Simon Peyton Jones et al. from 2000 (Simon Peyton Jones is one of the creators of Haskell), which has been adapted from "real world" finance to the special circumstances on the blockchain.

Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.
- It is based on the seminal paper "Composing Contracts: An Adventure in Financial Engineering" by Simon Peyton Jones et al. from 2000 (Simon Peyton Jones is one of the creators of Haskell), which has been adapted from "real world" finance to the special circumstances on the blockchain.
- Marlowe is *not* Turing-complete, and it is simple enough to allow for static analysis, which can automatically derive and prove important properties of Marlowe contracts.

Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.
- It is based on the seminal paper "Composing Contracts: An Adventure in Financial Engineering" by Simon Peyton Jones et al. from 2000 (Simon Peyton Jones is one of the creators of Haskell), which has been adapted from "real world" finance to the special circumstances on the blockchain.
- Marlowe is *not* Turing-complete, and it is simple enough to allow for static analysis, which can automatically derive and prove important properties of Marlowe contracts.
- Marlowe is also powerful enough to implement a significant part of all commonly used financial contracts.

Predictable Runtime

- All Marlowe contracts finish after finitely many steps (so there are no infinite loops), and static analysis can determine an upper bound for the maximum number of steps, so it is possible to know in advance for how many steps a given contract will run at most.

Predictable Runtime

- All Marlowe contracts finish after finitely many steps (so there are no infinite loops), and static analysis can determine an upper bound for the maximum number of steps, so it is possible to know in advance for how many steps a given contract will run at most.
- Marlowe guarantees that no money will be "trapped" in a contract forever: By the end of the contract, all money that has been paid into the contract will have been paid back to one of the parties participating in the contract.

EDSL

- Marlowe is a special type of DSL, a so-called EDSL, an Embedded Domain Specific Language, i.e. a DSL *embedded* into a “host language”.

EDSL

- Marlowe is a special type of DSL, a so-called EDSL, an Embedded Domain Specific Language, i.e. a DSL *embedded* into a “host language”.
- Marlowe’s host language is Haskell.

EDSL

- Marlowe is a special type of DSL, a so-called EDSL, an Embedded Domain Specific Language, i.e. a DSL *embedded* into a "host language".
- Marlowe's host language is Haskell.
- An EDSL has the advantage that all features of the host language are available to help facilitate creating expressions in the DSL.

EDSL

- Marlowe is a special type of DSL, a so-called EDSL, an Embedded Domain Specific Language, i.e. a DSL *embedded* into a "host language".
- Marlowe's host language is Haskell.
- An EDSL has the advantage that all features of the host language are available to help facilitate creating expressions in the DSL.
- This means for Marlowe, that we can use Haskell to write Marlowe contracts and that Marlowe contracts are nothing more than values of a specific Haskell type.

Blockly

- Alternatively, Marlowe contracts can be written in Blockly, a simple graphical language.

Blockly

- Alternatively, Marlowe contracts can be written in Blockly, a simple graphical language.
- This is tedious for complex contracts, but a good way for beginners.

Actus

- The Actus Financial Research Foundation has created a standard taxonomy to categorize and specify financial contracts.

Actus

- The Actus Financial Research Foundation has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in "plain English", which has led to ambiguity and misunderstandings.

Actus

- The Actus Financial Research Foundation has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in "plain English", which has led to ambiguity and misunderstandings.
- Instead of using normal prose, the Actus Standard composes contracts from a well-defined set of contractual terms and deterministic functions, which map those terms to future payment obligations.

Actus

- The Actus Financial Research Foundation has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in "plain English", which has led to ambiguity and misunderstandings.
- Instead of using normal prose, the Actus Standard composes contracts from a well-defined set of contractual terms and deterministic functions, which map those terms to future payment obligations.
- Using this method makes It possible to classify and describe the majority of all financial instruments in about 30 types and patterns.

Actus

- The Actus Financial Research Foundation has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in "plain English", which has led to ambiguity and misunderstandings.
- Instead of using normal prose, the Actus Standard composes contracts from a well-defined set of contractual terms and deterministic functions, which map those terms to future payment obligations.
- Using this method makes It possible to classify and describe the majority of all financial instruments in about 30 types and patterns.
- IOHK plans to use Plutus and Marlowe to implement the complete Actus Standard in Cardano.