Formal Investigation of the Extended UTxO Model

Laying the foundations for the formal verification of smart contracts

Orestis Melkonian, Wouter Swierstra, Manuel M.T. Chakravarty

August 18, 2019

Universiteit Utrecht

INPUT | OUTPUT

Introduction

Motivation

- A lot of blockchain applications recently
- Sophisticated transactional schemes via smart contracts
- Reasoning about their execution is:
    1. *necessary*, significant funds are involved
    2. *difficult*, due to concurrency
- Hence the need for automatic tools that verify no bugs exist
    - This has to be done statically!

Background

Bitcoin

- Based on *unspent transaction outputs* (UTxO)
- Smart contracts in the simple language SCRIPT

Ethereum

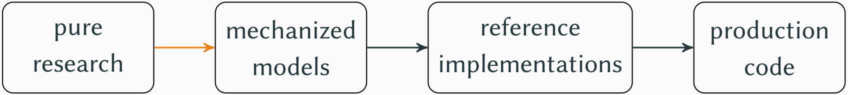
- Based on the notion of accounts
- Smart contracts in (almost) Turing-complete Solidity/EVM

Cardano (IOHK)

- UTxO-based, with several extensions
- Due to the extensions, smart contracts become more expressive

Methodology

- Keep things on an abstract level
  - Setup long-term foundations
- Fully mechanized approach, utilizing Agda's rich type system
- Fits well with IOHK's research-oriented approach



Extended UTxO

```
module UTxO.Types (Value : Set) (Hash : Set) where

record State : Set where
  field height : ℕ
              ⋮

record HashFunction (A : Set) : Set where
  field _#        : A → Hash
        injective : ∀ {x y} → x # ≡ y # → x ≡ y

postulate
  _# : ∀ {A : Set} → HashFunction A
```

Inputs and Output References

```
record TxOutputRef : Set where
  constructor _@_
  field id    : Hash
        index : ℕ

record TxInput : Set where
  field outputRef : TxOutputRef

        R D       : 𝕌
        redeemer  : State → el R
        validator : State → Value → PendingTx → el R → el D → Bool
```

- $\mathbb{U}$ is a simple type universe for first-order data.

Transactions

```
module UTxO (Address : Set) (_#ₐ : HashFunction Address)
            (_≟ₐ_ : Decidable {A = Address} _≡_) where

record TxOutput : Set where
  field value      : Value
        address    : Address

        Data       : 𝕌
        dataScript : State → el Data

record Tx : Set where
  field inputs  : List TxInput
        outputs : List TxOutput
        forge   : Value
        fee     : Value

Ledger : Set
Ledger = List Tx
```

Validation

$$
\begin{aligned}
\mathit{validate}\ :\ & \mathit{PendingTx} \\
& \to (i : \mathit{TxInput}) \\
& \to (o : \mathit{TxOutput}) \\
& \to D\ i \equiv \mathit{Data}\ o \\
& \to \mathit{State} \\
& \to \mathit{Bool}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{validate}\ \mathit{ptx}\ i\ o\ \mathit{refl}\ \mathit{st} = \\
&\quad \mathit{validator}\ i\ \mathit{st}\ (\mathit{value}\ o)\ \mathit{ptx}\ (\mathit{redeemer}\ i\ \mathit{st})\ (\mathit{dataScript}\ o\ \mathit{st})
\end{aligned}
$$

Unspent Outputs

$$
\begin{aligned}
&\mathit{unspentOutputs} : \mathit{Ledger} \rightarrow \mathit{Set}\langle\ \mathit{TxOutputRef}\ \rangle \\
&\mathit{unspentOutputs}\ [\,] \qquad\qquad\quad = \varnothing \\
&\mathit{unspentOutputs}\ (\mathit{tx} :: \mathit{txs}) = (\mathit{unspentOutputs}\ \mathit{txs} \setminus \mathit{spentOutputsTx}\ \mathit{tx}) \\
&\qquad\qquad\qquad\qquad\qquad\quad \cup\ \mathit{unspentOutputsTx}\ \mathit{tx} \\
&\quad \textbf{where} \\
&\qquad \mathit{spentOutputsTx}\ , \mathit{unspentOutputsTx} : \mathit{Tx} \rightarrow \mathit{Set}\langle\ \mathit{TxOutputRef}\ \rangle \\
&\qquad \mathit{spentOutputsTx} \qquad\quad = (\mathit{outputRef}\ \langle\$\rangle\ \_) \circ \mathit{inputs} \\
&\qquad \mathit{unspentOutputsTx}\ \mathit{tx} = (\mathit{tx}\#\ \texttt{@}\ \_)\ \langle\$\rangle\ \mathit{indices}\ (\mathit{outputs}\ \mathit{tx})
\end{aligned}
$$

Validity I

```
record IsValidTx (tx : Tx) (l : Ledger) : Set where
field
  validTxRefs : ∀ i → i ∈ inputs tx →
    Any (λ t → t♯ ≡ id (outputRef i)) l

  validOutputIndices : ∀ i → (i∈ : i ∈ inputs tx) →
    index (outputRef i) <
      length (outputs (lookupTx l (outputRef i) (validTxRefs i i∈)))

  validOutputRefs : ∀ i → i ∈ inputs tx →
    outputRef i ∈ unspentOutputs l

  validDataScriptTypes : ∀ i → (i∈ : i ∈ inputs tx) →
    D i ≡ Data (lookupOutput l (outputRef i) ...)
```

Validity II

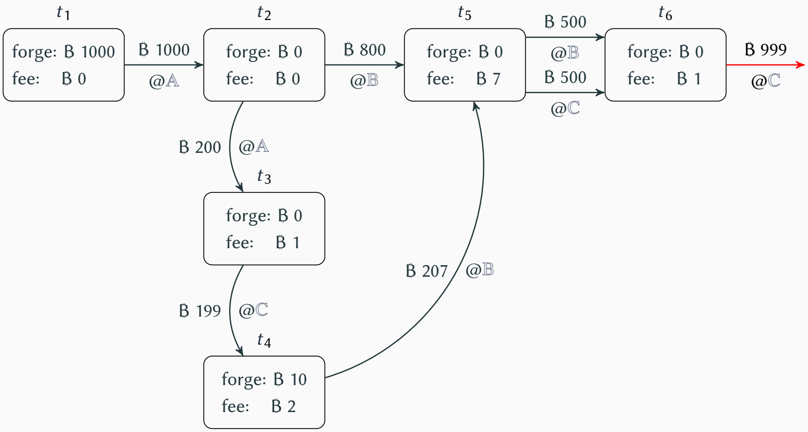
$$
\begin{aligned}
&\mathit{preservesValues} : \\
&\quad \mathit{forge}\ \mathit{tx} + \mathit{sum}\ (\mathit{lookupValue}\ l\ \ldots \langle\$\rangle\ \mathit{inputs}\ \mathit{tx}) \\
&\qquad \equiv \\
&\quad \mathit{fee}\ \mathit{tx} + \mathit{sum}\ (\mathit{value}\ \langle\$\rangle\ \mathit{outputs}\ \mathit{tx}) \\
&\mathit{noDoubleSpending} : \\
&\quad \mathit{noDuplicates}\ (\mathit{outputRef}\ \langle\$\rangle\ \mathit{inputs}\ \mathit{tx}) \\
&\mathit{allInputsValidate} : \forall\ i \rightarrow (i{\in} : i \in \mathit{inputs}\ \mathit{tx}) \rightarrow \\
&\quad \textbf{let}\ \mathit{out} = \mathit{lookupOutput}\ l\ (\mathit{outputRef}\ i)\ \ldots \\
&\qquad \mathit{ptx} = \mathit{mkPendingTx}\ l\ \mathit{tx}\ \mathit{validTxRefs}\ \mathit{validOutputIndices} \\
&\quad \textbf{in}\ \ T\ (\mathit{validate}\ \mathit{ptx}\ i\ \mathit{out}\ (\mathit{validDataScriptTypes}\ i\ i{\in})\ (\mathit{getState}\ l)) \\
&\mathit{validateValidHashes} : \forall\ i \rightarrow (i{\in} : i \in \mathit{inputs}\ \mathit{tx}) \rightarrow \\
&\quad \textbf{let}\ \mathit{out} = \mathit{lookupOutput}\ l\ (\mathit{outputRef}\ i)\ \ldots \\
&\quad \textbf{in}\ \ (\mathit{address}\ \mathit{out})\# \equiv \mathit{validator}\ i\#
\end{aligned}
$$

We do not want a ledger to be any list of transactions, but a “snoc”-list that carries proofs of validity:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ValidLedger} : \mathit{Ledger} \to \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\bullet & : & \mathit{ValidLedger}\ [\,] \\
\_\oplus\_\dashv\_ & : & \mathit{ValidLedger}\ l \\
 & \to & (\mathit{tx} : \mathit{Tx}) \\
 & \to & \mathit{IsValidTx}\ \mathit{tx}\ l \\
 & \to & \mathit{ValidLedger}\ (\mathit{tx} :: l)
\end{array}
\end{array}
$$

$$
\begin{aligned}
&\vdots\\
&\mathit{validOutputRefs?} : \forall\ (tx : \mathit{Tx})\ (l : \mathit{Ledger})\\
&\quad \to \mathit{Dec}\ (\forall\ i \to i \in \mathit{inputs}\ tx \to \mathit{outputRef}\ i \in \mathit{unspentOutputs}\ l)\\
&\mathit{validOutputRefs?}\, tx\ l =\\
&\quad \forall?\ (\mathit{inputs}\ tx)\ \lambda\ i\ \_ \to \mathit{outputRef}\ i \in?\ \mathit{unspentOutputs}\ l\\
&\quad \vdots\\
&\quad \textbf{where}\\
&\qquad \forall?\ :\ (xs : \mathit{List}\ A)\\
&\qquad\qquad \to \{\, P : (x : A)\ (x\in : x \in xs) \to \mathit{Set}\}\\
&\qquad\qquad \to (\forall\ x \to (x\in : x \in xs) \to \mathit{Dec}\ (P\ x\ x\in))\\
&\qquad\qquad \to \mathit{Dec}\ (\forall\ x\ x\in \to P\ x\ x\in)
\end{aligned}
$$

Example: Transaction Graph



Example: Definitions of Transactions

$$
\begin{aligned}
& t_1 \, , t_2 \, , t_3 \, , t_4 \, , t_5 \, , t_6 : \mathit{Tx} \\
& t_1 = \textbf{record} \, \{ \, \mathit{inputs} = [\mathit{withPolicy} \; c_{0\,0}] \\
& \qquad\qquad ; \mathit{outputs} = [\text{฿} \; 1000 \; @ \; \mathbb{A}] \\
& \qquad\qquad ; \mathit{forge} = \text{฿} \; 1000 \\
& \qquad\qquad ; \mathit{fee} = \text{฿} \; 0 \, \} \\
& \vdots \\
& t_6 = \textbf{record} \, \{ \, \mathit{inputs} = [\mathit{withScripts} \; t_{5\,0} \, , \mathit{withScripts} \; t_{5\,1}] \\
& \qquad\qquad ; \mathit{outputs} = [\text{฿} \; 999 \; @ \; \mathbb{C}] \\
& \qquad\qquad ; \mathit{forge} = \text{฿} \; 0 \\
& \qquad\qquad ; \mathit{fee} = \text{฿} \; 1 \, \}
\end{aligned}
$$

Example: Rewrite Rules

Our hash function is a postulate, so our decision procedures will get stuck...

```
{-# OPTIONS --rewriting #-}
postulate
  eq₁₀ : (mkValidator t₁₀)# ≡ 𝔸
  ⋮
  eq₆₀ : (mkValidator t₆₀)# ≡ ℂ

{-# BUILTIN REWRITE _≡_ #-}
{-# REWRITE eq₀ , eq₁₀ , ... , eq₆₀ #-}
```

Example: Correct-by-construction Ledger

$$
\begin{aligned}
&\mathit{ex\text{-}ledger} : \mathit{ValidLedger}\,[\,t_6\,,\,t_5\,,\,t_4\,,\,t_3\,,\,t_2\,,\,t_1\,]\\
&\mathit{ex\text{-}ledger} =\\
&\quad \cdot\, t_1 \quad \dashv \textbf{record}\,\{\,\mathit{validTxRefs} = \mathit{toWitness}\,\{\,Q = \mathit{validTxRefs?}\,t_1\;l_0\,\}\;\mathit{tt}\\
&\qquad\qquad\qquad\qquad\quad \ldots\,\}\\
&\quad \vdots\\
&\quad \oplus\, t_6 \dashv \textbf{record}\,\{\,\ldots\,\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{utxo} : \mathit{list}\,(\mathit{unspentOutputs}\;\mathit{ex\text{-}ledger}) \equiv [\,t_{6\,0}\,]\\
&\mathit{utxo} = \mathit{refl}
\end{aligned}
$$

Meta-theory

Weakening via Injections

```
module Weakening
  (𝔸 : Set) (_♯ᵃ : HashFunction 𝔸) (_≟ᵃ_ : Decidable {A = 𝔸} _≡_)
  (𝔹 : Set) (_♯ᵇ : HashFunction 𝔹) (_≟ᵇ_ : Decidable {A = 𝔹} _≡_)
  (A↪B : 𝔸 , _♯ᵃ ↪ 𝔹 , _♯ᵇ)
  where

  import UTxO.Validity 𝔸 _♯ᵃ _≟ᵃ_ as A
  import UTxO.Validity 𝔹 _♯ᵇ _≟ᵇ_ as B
```

Weakening Lemma

After translating addresses, validity is preserved:

$$\begin{aligned}
&\mathit{weakening} : \forall\ \{\mathit{tx} : \mathit{A.Tx}\}\ \{l : \mathit{A.Ledger}\} \\
&\quad \rightarrow \mathit{A.IsValidTx}\ \mathit{tx}\ l \\
&\quad \overline{\phantom{\rightarrow \mathit{B.IsValidTx}\ (\mathit{weakenTx}\ \mathit{tx})\ (\mathit{weakenLedger}\ l)}} \\
&\quad \rightarrow \mathit{B.IsValidTx}\ (\mathit{weakenTx}\ \mathit{tx})\ (\mathit{weakenLedger}\ l) \\
&\mathit{weakening} = \ldots
\end{aligned}$$

Inspiration from Separation Logic

- One wants to reason in a modular manner
  - Conversely, one can study a ledger by studying its components, that is we can reason *compositionally*
- In concurrency, $P * Q$ holds for disjoint parts of the memory heap
- In blockchain, $P * Q$ holds for disjoint parts of the ledger
  - But what does it mean for two ledgers to be disjoint?

Disjoint Ledgers

Two ledgers l and l' are disjoint, when

1. No common transactions: $\mathit{Disjoint}\ l\ l' = \forall\ t \to (t \in l \times v \in l')$
2. Validation does not break:

$$
\begin{aligned}
&\mathit{PreserveValidations} : \mathit{Ledger} \to \mathit{Ledger} \to \mathit{Set}\\
&\mathit{PreserveValidations}\ l\ l'' =\\
&\quad \forall\ tx \to tx \in l \to tx \in l'' \to\\
&\qquad \forall\ \{ptx\ i\ out\ vds\} \to \mathit{validate}\ ptx\ i\ out\ vds\ (\mathit{getState}\ (\mathit{upTo}\ tx\ l''))\\
&\qquad\qquad \equiv\ \mathit{validate}\ ptx\ i\ out\ vds\ (\mathit{getState}\ (\mathit{upTo}\ tx\ l))
\end{aligned}
$$

Combining Ledgers

$$
\begin{aligned}
&\_\leftrightarrow\_\dashv\_ : \forall \{ l\ l'\ l'' : \mathit{Ledger} \} \\
&\quad \rightarrow \mathit{ValidLedger}\ l \\
&\quad \rightarrow \mathit{ValidLedger}\ l' \\
&\quad \rightarrow \mathit{Interleaving}\ l\ l'\ l'' \\
&\quad \times \mathit{Disjoint}\ l\ l' \\
&\quad \times \mathit{PreserveValidations}\ l\ l'' \\
&\quad \times \mathit{PreserveValidations}\ l'\ l'' \\
&\quad \overline{\hphantom{\mathit{PreserveValidations}\ l'\ l''\ \mathit{PreserveValidations}}} \\
&\quad \rightarrow \mathit{ValidLedger}\ l''
\end{aligned}
$$

Future Work

Next Steps: UTxO

1. Integrate James Chapman's work on plutus-metatheory
    - Plutus terms instead of their denotations (i.e. Agda functions)
2. Support for multi-signature schemes

Next Steps: Certified Compilation

- **BitML**: Idealistic process calculus for Bitcoin smart contracts
- We already have instrinsically-typed BitML contracts in Agda, as well as its small-step semantics and corresponding meta-theory
- **Plan**: Certified compilation from BitML to (extended) UTxO
    - Any attack possible at the transaction level, will also manifest itself in the higher-level BitML semantics
- Come check my poster for more details on formalizing BitML!

Conclusion

Conclusion

- Formal methods are a promising direction for blockchain
    - Especially language-oriented, type-driven approaches
- Although formalization is tedious and time-consuming
    - Strong results and deep understanding of models
    - Certified compilation is here to stay! (c.f. *CompCert, seL4*)
- However, tooling is badly needed....
    - We need better, more sophisticated programming technology for dependently-typed languages

Questions?