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

July 8, 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

pure research → mechanized models → reference implementations → production code

Extended UTxO

Basic Types

```
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

$$\begin{aligned}
&\textbf{module } \mathit{UTxO}\ (\mathit{Address} : \mathit{Set})\ (_\#_{\mathrm{a}} : \mathit{HashFunction}\ \mathit{Address})\\
&\qquad\qquad (_\stackrel{?}{=}_{\mathrm{a}}_ : \mathit{Decidable}\ \{A = \mathit{Address}\}\ _\equiv_)\ \textbf{where}
\end{aligned}$$

$$\begin{aligned}
&\textbf{record } \mathit{TxOutput} : \mathit{Set}\ \textbf{where}\\
&\quad \textbf{field}\ \mathit{value} \quad : \mathit{Value}\\
&\qquad\qquad \mathit{address} : \mathit{Address}\\
&\qquad\qquad \mathit{Data} \quad\ : \mathbb{U}\\
&\qquad\qquad \mathit{dataScript} : \mathit{State} \rightarrow \mathit{el}\ \mathit{Data}
\end{aligned}$$

$$\begin{aligned}
&\textbf{record } \mathit{Tx} : \mathit{Set}\ \textbf{where}\\
&\quad \textbf{field}\ \mathit{inputs} \quad : \mathit{List}\ \mathit{TxInput}\\
&\qquad\qquad \mathit{outputs} : \mathit{List}\ \mathit{TxOutput}\\
&\qquad\qquad \mathit{forge} \quad\ : \mathit{Value}\\
&\qquad\qquad \mathit{fee} \qquad : \mathit{Value}
\end{aligned}$$

$$\begin{aligned}
&\mathit{Ledger} : \mathit{Set}\\
&\mathit{Ledger} = \mathit{List}\ \mathit{Tx}
\end{aligned}$$

Validation

$$
\begin{aligned}
&\mathit{validate} \;:\; \mathit{PendingTx}\\
&\qquad\quad \to (i : \mathit{TxInput})\\
&\qquad\quad \to (o : \mathit{TxOutput})\\
&\qquad\quad \to D\ i \equiv \mathit{Data}\ o\\
&\qquad\quad \to \mathit{State}\\
&\qquad\quad \to \mathit{Bool}\\
&\mathit{validate}\ \mathit{ptx}\ i\ o\ \mathit{refl}\ \mathit{st} =\\
&\quad \mathit{validator}\ i\ \mathit{st}\ (\mathit{value}\ o)\ \mathit{ptx}\ (\mathit{redeemer}\ i\ \mathit{st})\ (\mathit{dataScript}\ o\ \mathit{st})
\end{aligned}
$$

Unspent Outputs

$$
\begin{aligned}
&\mathit{unspentOutputs} : \mathit{Ledger} \rightarrow \mathit{Set}\langle\ \mathit{TxOutputRef}\ \rangle \\
&\mathit{unspentOutputs}\ [\,] \qquad\qquad\quad = \varnothing \\
&\mathit{unspentOutputs}\ (\mathit{tx} :: \mathit{txs}) = (\mathit{unspentOutputs}\ \mathit{txs} \setminus \mathit{spentOutputsTx}\ \mathit{tx}) \\
&\qquad\qquad\qquad\qquad\qquad\quad \cup\ \mathit{unspentOutputsTx}\ \mathit{tx}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{spentOutputsTx}, \mathit{unspentOutputsTx} : \mathit{Tx} \rightarrow \mathit{Set}\langle\ \mathit{TxOutputRef}\ \rangle \\
&\mathit{spentOutputsTx} \qquad\quad = (\mathit{outputRef}\ \langle\$\rangle\ \_) \circ \mathit{inputs} \\
&\mathit{unspentOutputsTx}\ \mathit{tx} = (\mathit{tx}\#\ @\ \_)\ \langle\$\rangle\ \mathit{indices}\ (\mathit{outputs}\ \mathit{tx})
\end{aligned}
$$

Validity I

$$
\begin{array}{l}
\textbf{record}\ \mathit{IsValidTx}\ (\mathit{tx} : \mathit{Tx})\ (\mathit{l} : \mathit{Ledger}) : \mathit{Set}\ \textbf{where} \\
\textbf{field} \\
\quad \mathit{validTxRefs} : \forall\ i \rightarrow i \in \mathit{inputs}\ \mathit{tx} \rightarrow \\
\quad\quad \mathit{Any}\ (\lambda\ t \rightarrow t\# \equiv \mathit{id}\ (\mathit{outputRef}\ i))\ l \\
\quad \mathit{validOutputIndices} : \forall\ i \rightarrow (i\!\in : i \in \mathit{inputs}\ \mathit{tx}) \rightarrow \\
\quad\quad \mathit{index}\ (\mathit{outputRef}\ i) < \\
\quad\quad\quad \mathit{length}\ (\mathit{outputs}\ (\mathit{lookupTx}\ l\ (\mathit{outputRef}\ i)\ (\mathit{validTxRefs}\ i\ i\!\in))) \\
\quad \mathit{validOutputRefs} : \forall\ i \rightarrow i \in \mathit{inputs}\ \mathit{tx} \rightarrow \\
\quad\quad \mathit{outputRef}\ i \in \mathit{unspentOutputs}\ l \\
\quad \mathit{validDataScriptTypes} : \forall\ i \rightarrow (i\!\in : i \in \mathit{inputs}\ \mathit{tx}) \rightarrow \\
\quad\quad D\ i \equiv D\ (\mathit{lookupOutput}\ l\ (\mathit{outputRef}\ i)\ \ldots)
\end{array}
$$

Validity II

```
preservesValues :
  forge tx + sum (lookupValue l ... ⟨$⟩ inputs tx)
    ≡
  fee tx + sum (value ⟨$⟩ outputs tx)

noDoubleSpending :
  noDuplicates (outputRef ⟨$⟩ inputs tx)

allInputsValidate : ∀ i → (i∈ : i ∈ inputs tx) →
  let out = lookupOutput l (outputRef i) ...
      ptx = mkPendingTx l tx validTxRefs validOutputIndices
  in  T (validate ptx i out (validDataScriptTypes i i∈) (getState l))

validateValidHashes : ∀ i → (i∈ : i ∈ inputs tx) →
  let out = lookupOutput l (outputRef i) ...
  in  (address out)# ≡ validator i#
```

Valid Ledgers

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{ValidLedger} : \mathit{Ledger} \rightarrow \mathit{Set}\ \textbf{where}} \\
\quad \bullet & : & \mathit{ValidLedger}\ [\,] \\
\quad \_\oplus\_\dashv\_ & : & \mathit{ValidLedger}\ l \\
 & \rightarrow & (tx : \mathit{Tx}) \\
 & \rightarrow & \mathit{IsValidTx}\ tx\ l \\
 & \rightarrow & \mathit{ValidLedger}\ (tx :: l)
\end{array}
$$

Decision Procedures

$$\vdots$$

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

Extension: Multi-currency

1. Generalize values from integers to maps: $\mathit{Value} = \mathit{List}\ (\mathit{Hash} \times \mathbb{N})$
2. Implement additive group operators (on top of AVL trees):

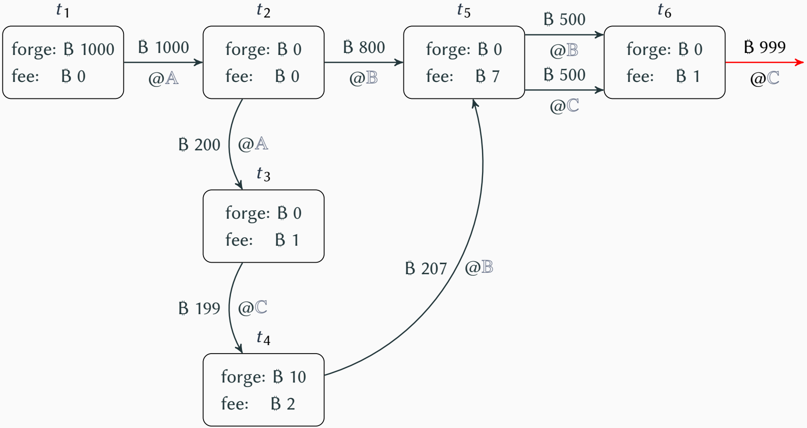
```
open import Data.AVL ℕ-strictTotalOrder

_+ᶜ_ : Value → Value → Value
c +ᶜ c′ = toList (foldl go (fromList c) c′)
  where
    go : Tree Hash ℕ → (Hash × ℕ) → Tree Hash ℕ
    go m (k , v) = insertWith k ((_+ v) ∘ fromMaybe 0) m

sumᶜ : Values → Value
sumᶜ = foldl _+ᶜ_ []
```

Multi-currency: Forging condition

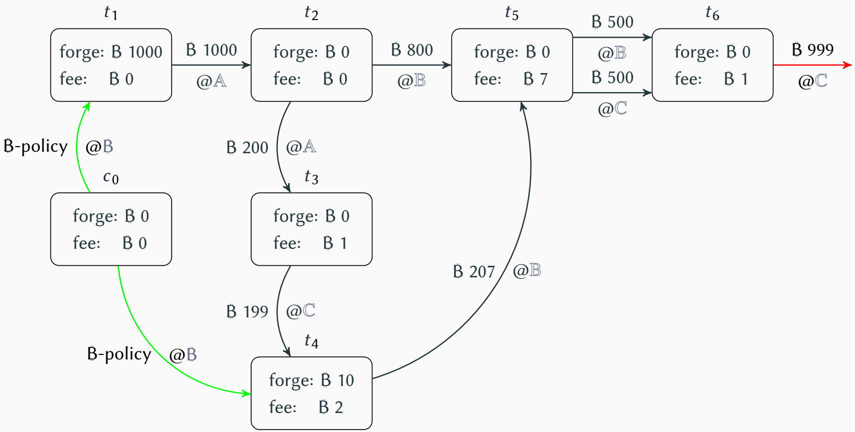
We now need to enforce monetary policies on forging transactions:

```
record IsValidTx (tx : Tx) (l : Ledger) : Set where
  ⋮
  forging :
    ∀ c → c ∈ keys (forge tx) →
      ∃[ i ] ∃λ (i∈ : i ∈ inputs tx) →
        let out = lookupOutput l (outputRef i) ...
        in (address out) # ≡ c
```

Example: Transaction Graph



Example: Transaction Graph + monetary policy



Example: Setting Up

$$\mathit{Address} = \mathbb{N}$$

$$\mathbb{A}\;,\mathbb{B}\;,\mathbb{C} : \mathit{Address}$$

$\mathbb{A} = 1$ -- *first address*

$\mathbb{B} = 2$ -- *second address*

$\mathbb{C} = 3$ -- *third address*

open import $\mathit{UTxO}\ \mathit{Address}\ (\lambda\, x \to x)\ \_\stackrel{?}{=}\_$

$$\text{฿-}\mathit{validator} : \mathit{State} \to \ldots \to \mathit{Bool}$$

$$\text{฿-}\mathit{validator}\ (\textbf{record}\ \{\mathit{height} = h\})\ \ldots = (h \equiv^{b} 1) \lor (h \equiv^{b} 4)$$

Example: Smart Constructors

$$
\begin{aligned}
&\mathit{mkValidator} : \mathit{TxOutputRef} \to (\ldots \to \mathit{TxOutputRef} \to \ldots \to \mathit{Bool})\\
&\mathit{mkValidator}\ o\ \ \ldots\ \ o'\ \ldots = o \stackrel{b}{\equiv} o'
\end{aligned}
$$

$$
\begin{aligned}
&₿\ \_ : \mathbb{N} \to \mathit{Value}\\
&₿\ v = [\,(₿\text{-}\mathit{validator}\#\ ,\ v)\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{withScripts} : \mathit{TxOutputRef} \to \mathit{TxInput}\\
&\mathit{withScripts}\ o = \textbf{record}\ \{\ \mathit{outputRef} = o\\
&\qquad\qquad\qquad\qquad\quad ;\ \mathit{redeemer}\ \ = \lambda\ \_ \to o\\
&\qquad\qquad\qquad\qquad\quad ;\ \mathit{validator}\ \ = \mathit{mkValidator}\ \mathit{tin}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{withPolicy} : \mathit{TxOutputRef} \to \mathit{TxInput}\\
&\mathit{withPolicy}\ \mathit{tin} = \textbf{record}\ \{\ \mathit{outputRef} = \mathit{tin}\\
&\qquad\qquad\qquad\qquad\quad ;\ \mathit{redeemer} = \lambda\ \_ \to \mathit{tt}\\
&\qquad\qquad\qquad\qquad\quad ;\ \mathit{validator} = ₿\text{-}\mathit{validator}\}
\end{aligned}
$$

$$
\begin{aligned}
&\_\ @\ \_ : \mathit{Value} \to \mathit{Index}\ \mathit{addresses} \to \mathit{TxOutput}\\
&v\ @\ \mathit{addr} = \textbf{record}\ \{\ \mathit{value} = v;\ \mathit{address} = \mathit{addr};\ \mathit{dataScript} = \lambda\ \_ \to \mathit{tt}\}
\end{aligned}
$$

Example: Definitions of Transactions

$$
\begin{aligned}
& c_0 \,,\, t_1 \,,\, t_2 \,,\, t_3 \,,\, t_4 \,,\, t_5 \,,\, t_6 : \mathit{Tx} \\
& c_0 = \textbf{record}\, \{\, \mathit{inputs} = [\,] \\
& \qquad ;\, \mathit{outputs} = [\,\text{฿}\ 0 \mathbin{@} (\text{฿-}\mathit{validator}\#) \,,\, \text{฿}\ 0 \mathbin{@} (\text{฿-}\mathit{validator}\#)\,] \\
& \qquad ;\, \mathit{forge} = \text{฿}\ 0 \\
& \qquad ;\, \mathit{fee} = \text{฿}\ 0 \,\} \\
& t_1 = \textbf{record}\, \{\, \mathit{inputs} = [\, \mathit{withPolicy}\ c_{0\,0} \,] \\
& \qquad ;\, \mathit{outputs} = [\,\text{฿}\ 1000 \mathbin{@} \mathbb{A}\,] \\
& \qquad ;\, \mathit{forge} = \text{฿}\ 1000 \\
& \qquad ;\, \mathit{fee} = \text{฿}\ 0 \,\} \\
& \vdots \\
& t_6 = \textbf{record}\, \{\, \mathit{inputs} = [\, \mathit{withScripts}\ t_{5\,0} \,,\, \mathit{withScripts}\ t_{5\,1} \,] \\
& \qquad ;\, \mathit{outputs} = [\,\text{฿}\ 999 \mathbin{@} \mathbb{C}\,] \\
& \qquad ;\, \mathit{forge} = \text{฿}\ 0 \\
& \qquad ;\, \mathit{fee} = \text{฿}\ 1 \,\}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{ex\text{-}ledger} : \mathit{ValidLedger}\ [\, t_6\ ,\ t_5\ ,\ t_4\ ,\ t_3\ ,\ t_2\ ,\ t_1\ ,\ c_0 \,] \\
&\mathit{ex\text{-}ledger} = \\
&\quad \cdot\ c_0 \dashv \textbf{record}\ \{ \ldots \} \\
&\quad \oplus\ t_1 \dashv \textbf{record}\ \{\, \mathit{validTxRefs} = \mathit{toWitness}\ \{\, Q = \mathit{validTxRefs?}\, t_1\ l_0 \,\}\ \mathit{tt} \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\quad ;\ \mathit{forging} \quad = \mathit{toWitness}\ \{\, Q = \mathit{forging?}\ \ldots \,\}\ \mathit{tt} \,\} \\
&\quad \vdots \\
&\quad \oplus\ t_6 \dashv \textbf{record}\ \{ \ldots \}
\end{aligned}
$$

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

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:

$$
\begin{array}{l}
\mathit{weakening} : \forall \{ \mathit{tx} : \mathit{A.Tx} \} \{ l : \mathit{A.Ledger} \} \\
\quad \rightarrow \mathit{A.IsValidTx}\ \mathit{tx}\ l \\
\qquad \overline{\hspace{20em}} \\
\quad \rightarrow \mathit{B.IsValidTx}\ (\mathit{weakenTx}\ \mathit{tx})\ (\mathit{weakenLedger}\ l) \\
\mathit{weakening} = \ldots
\end{array}
$$

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:

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

Combining Ledgers

$$
\begin{aligned}
\_ \leftrightarrow \_ \dashv \_ &: \forall \{l\ l'\ l'' : \mathit{Ledger}\} \\
&\rightarrow \mathit{ValidLedger}\ l \\
&\rightarrow \mathit{ValidLedger}\ l' \\
&\rightarrow \mathit{Interleaving}\ l\ l'\ l'' \\
&\times \mathit{Disjoint}\ l\ l' \\
&\times \mathit{PreserveValidations}\ l\ l'' \\
&\times \mathit{PreserveValidations}\ l'\ l'' \\
&\overline{\hspace{10em}} \\
&\rightarrow \mathit{ValidLedger}\ l''
\end{aligned}
$$

Future Work

Next Steps: UTxO

1. Multi-currency: non-fungible tokens
   - 2-level maps that introduce intermediate layer with tokens
2. Integrate James Chapman's work on plutus-metatheory
   - Plutus terms instead of their denotations (i.e. Agda functions)
3. Support for multi-signature schemes

Next Steps: Others

1. Proof automation via domain-specific tactics
   - Accommodate future formalization efforts
2. Featherweight Solidity
   - Provide proof-of-concept model in Agda
   - Perform some initial comparison with UTxO
3. Investigate Chad Nester's work on monoidal ledgers
   - This leads to another reasoning device: string diagrams

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?