Formal Investigation of the Extended UTxO Model

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

July 8, 2019

Utrecht University, The Netherlands

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

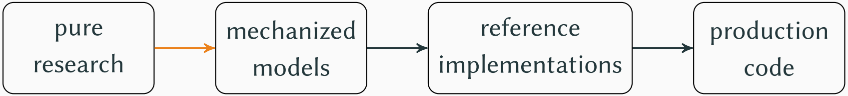
- Based on the notion of accounts
- Smart contracts in (almost) Turing-complete Solidity/EVM

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

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

$$
\begin{array}{l}
\textbf{record}\ \mathit{TxOutputRef} : \mathit{Set}\ \textbf{where} \\
\quad \textbf{constructor}\ \_@\_ \\
\quad \textbf{field}\ \mathit{id} \quad\ \ : \mathit{Hash} \\
\qquad\qquad\ \ \mathit{index} : \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\textbf{record}\ \mathit{TxInput} : \mathit{Set}\ \textbf{where} \\
\quad \textbf{field}\ \mathit{outputRef} : \mathit{TxOutputRef} \\
\qquad\qquad\ \ R\ D \qquad\quad\ : \mathbb{U} \\
\qquad\qquad\ \ \mathit{redeemer} \ \ : \mathit{State} \to \mathit{el}\ R \\
\qquad\qquad\ \ \mathit{validator} \ \ : \mathit{State} \to \mathit{Value} \to \mathit{PendingTx} \to \mathit{el}\ R \to \mathit{el}\ D \to \mathit{Bool}
\end{array}
$$

- $\mathbb{U}$ 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
```

Validation

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

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

Unspent Outputs

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

Validity I

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

Validity II

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

Valid Ledgers

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

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

Decision Procedures

```
⋮
validOutputRefs? : ∀ (tx : Tx) (l : Ledger)
  → Dec (∀ i → i ∈ inputs tx → outputRef i ∈ unspentOutputs l)
validOutputRefs? tx l =
  ∀? (inputs tx) λ i _ → outputRef i ∈? unspentOutputs l
  ⋮
  where
    ∀?  :  (xs : List A)
        → {P : (x : A) (x∈ : x ∈ xs) → Set}
        → (∀ x → (x∈ : x ∈ xs) → Dec (P x x∈))
        → Dec (∀ x x∈ → P x x∈)
```

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):

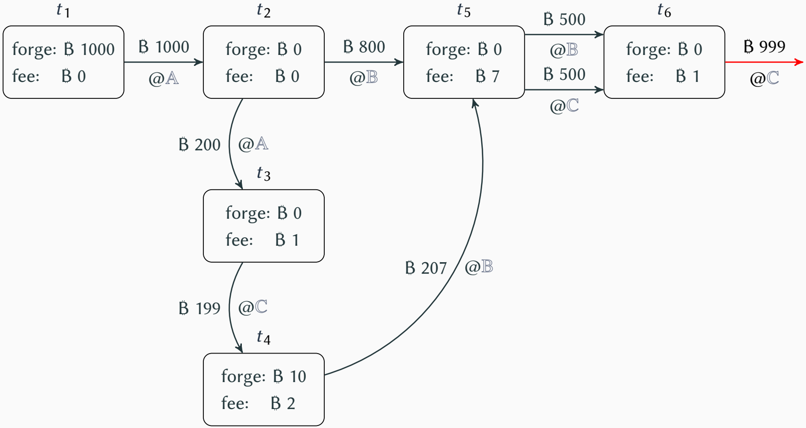
$$
\begin{aligned}
&\textbf{open import}\ \mathit{Data.AVL}\ \mathbb{N}\text{-}\mathit{strictTotalOrder}\\
&\_+^{c}\_ : \mathit{Value} \to \mathit{Value} \to \mathit{Value}\\
&c +^{c} c' = \mathit{toList}\ (\mathit{foldl}\ \mathit{go}\ (\mathit{fromList}\ c)\ c')\\
&\quad \textbf{where}\\
&\qquad \mathit{go} : \mathit{Tree}\ \mathit{Hash}\ \mathbb{N} \to (\mathit{Hash} \times \mathbb{N}) \to \mathit{Tree}\ \mathit{Hash}\ \mathbb{N}\\
&\qquad \mathit{go}\ m\ (k , v) = \mathit{insertWith}\ k\ ((\_+ v) \circ \mathit{fromMaybe}\ 0)\ m\\
&\mathit{sum}^{c} : \mathit{Values} \to \mathit{Value}\\
&\mathit{sum}^{c} = \mathit{foldl}\ \_+^{c}\_\ [\,]
\end{aligned}
$$

Multi-currency: Forging condition

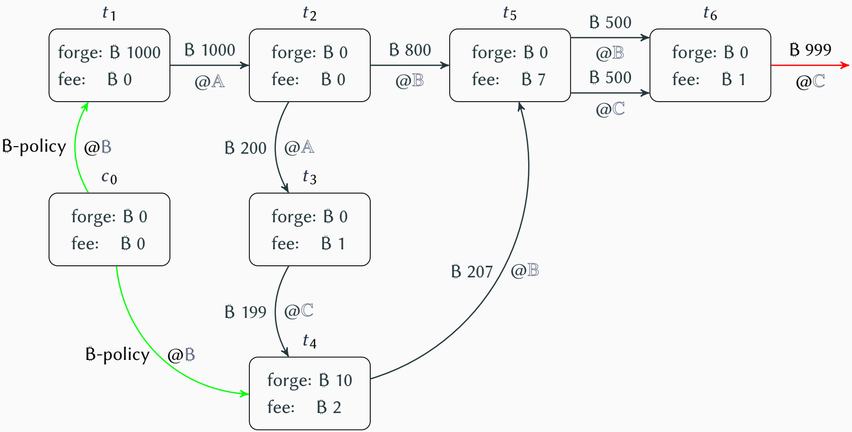
We now need to enforce monetary policies on forging transactions:

$$
\begin{array}{l}
\textbf{record } \mathit{IsValidTx}\ (\mathit{tx} : \mathit{Tx})\ (\mathit{l} : \mathit{Ledger}) : \mathit{Set}\ \textbf{where} \\
\quad \vdots \\
\quad \mathit{forging} : \\
\qquad \forall\ c \rightarrow c \in \mathit{keys}\ (\mathit{forge}\ \mathit{tx}) \rightarrow \\
\qquad\quad \exists[\, i \,]\ \exists\lambda\ (i{\in} : i \in \mathit{inputs}\ \mathit{tx}) \rightarrow \\
\qquad\qquad \textbf{let } \mathit{out} = \mathit{lookupOutput}\ \mathit{l}\ (\mathit{outputRef}\ i)\ \ \ldots \\
\qquad\qquad \textbf{in } (\mathit{address}\ \mathit{out})\# \equiv c
\end{array}
$$

Example: Transaction Graph



Example: Transaction Graph + monetary policy



Example: Setting Up

$Address = \mathbb{N}$

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

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

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

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

open import $UTxO\ Address\ (\lambda\ x \rightarrow x)\ \_\overset{?}{=}\_$

$₿\text{-}validator : State \rightarrow \ldots \rightarrow Bool$

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

Example: Smart Constructors

$$mkValidator : TxOutputRef \to (\ldots \to TxOutputRef \to \ldots \to Bool)$$
$$mkValidator\ o\ \ \ldots\ \ o'\ \ldots = o \equiv^{b}\ o'$$

$$฿\ \_ : \mathbb{N} \to Value$$
$$฿\ v = [\,(฿\text{-}validator\#\ ,\ v)\,]$$

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

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

$$\_\ @\ \_ : Value \to Index\ addresses \to TxOutput$$

Example: Definitions of Transactions

$$c_0 , t_1 , t_2 , t_3 , t_4 , t_5 , t_6 : Tx$$

$$\begin{aligned}
c_0 = \mathbf{record}\ \{\ & inputs = [\,] \\
;\ & outputs = [\,\mathbb{B}\ 0\ @\ (\mathbb{B}\text{-}validator\#)\ ,\ \mathbb{B}\ 0\ @\ (\mathbb{B}\text{-}validator\#)\,] \\
;\ & forge = \mathbb{B}\ 0 \\
;\ & fee = \mathbb{B}\ 0\ \}
\end{aligned}$$

$$\begin{aligned}
t_1 = \mathbf{record}\ \{\ & inputs = [\, withPolicy\ c_{0\,0}\,] \\
;\ & outputs = [\,\mathbb{B}\ 1000\ @\ \mathbb{A}\,] \\
;\ & forge = \mathbb{B}\ 1000 \\
;\ & fee = \mathbb{B}\ 0\ \}
\end{aligned}$$

$$\vdots$$

$$\begin{aligned}
t_6 = \mathbf{record}\ \{\ & inputs = [\, withScripts\ t_{5\,0}\ ,\ withScripts\ t_{5\,1}\,] \\
;\ & outputs = [\,\mathbb{B}\ 999\ @\ \mathbb{C}\,] \\
;\ & forge = \mathbb{B}\ 0 \\
;\ & fee = \mathbb{B}\ 1\ \}
\end{aligned}$$

Example: Rewrite Rules

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

$$\{\text{-}\#\ \ \textbf{OPTIONS}\ \text{--rewriting}\ \ \#\text{-}\}$$

postulate

$$eq_{1\,0} : (mkValidator\ t_{1\,0})\# \equiv \mathbb{A}$$

$$\vdots$$

$$eq_{6\,0} : (mkValidator\ t_{6\,0})\# \equiv \mathbb{C}$$

$$\{\text{-}\#\ \ \textbf{BUILTIN REWRITE}\ \_\equiv\_\ \ \#\text{-}\}$$

$$\{\text{-}\#\ \ \textbf{REWRITE}\ eq_0\ , eq_{1\,0}\ , \ldots , eq_{6\,0}\ \ \#\text{-}\}$$

Example: Correct-by-construction Ledger

$$\mathit{ex\text{-}ledger} : \mathit{ValidLedger}\ [\, t_6\ ,\ t_5\ ,\ t_4\ ,\ t_3\ ,\ t_2\ ,\ t_1\ ,\ c_0 \,]$$

$$\mathit{ex\text{-}ledger} =$$

$$\begin{array}{l}
\quad \cdot\ c_0 \dashv \textbf{record}\ \{ \ldots \} \\
\quad \oplus\ t_1 \dashv \textbf{record}\ \{\ \mathit{validTxRefs} = \mathit{toWitness}\ \{\ Q = \mathit{validTxRefs?}\ t_1\ l_0\ \}\ \mathit{tt} \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\quad ;\ \mathit{forging} \qquad = \mathit{toWitness}\ \{\ Q = \mathit{forging?}\ \ldots\ \}\ \mathit{tt}\ \} \\
\quad \vdots \\
\quad \oplus\ t_6 \dashv \textbf{record}\ \{ \ldots \}
\end{array}$$

$$\mathit{utxo} : \mathit{list}\ (\mathit{unspentOutputs}\ \mathit{ex\text{-}ledger}) \equiv [\, t_{6\,0} \,]$$

$$\mathit{utxo} = \mathit{refl}$$

UTxO: 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:

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

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: $\mathit{Disjoint}\ l\ l' = \forall\ t \rightarrow (t \in l \times v \in l')$
2. Validation does not break:

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

Combining Ledgers

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

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

1. … 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?