Formalizing Extended UTxO and BitML Calculus in Agda

Towards formal verification for smart contracts

Orestis Melkonian

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

pure research → mechanized models → reference implementations → production code

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

```
module UTxO (Address : Set) (_#a : Hash 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  : Set⟨ TxInput ⟩
        outputs : List TxOutput
        forge   : Value
        fee     : Value

Ledger : Set
Ledger = List Tx
```

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

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

Validity I

```
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 ≡ D (lookupOutput l (outputRef i)  . . .)
```

Validity II

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

Valid Ledgers

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

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

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):

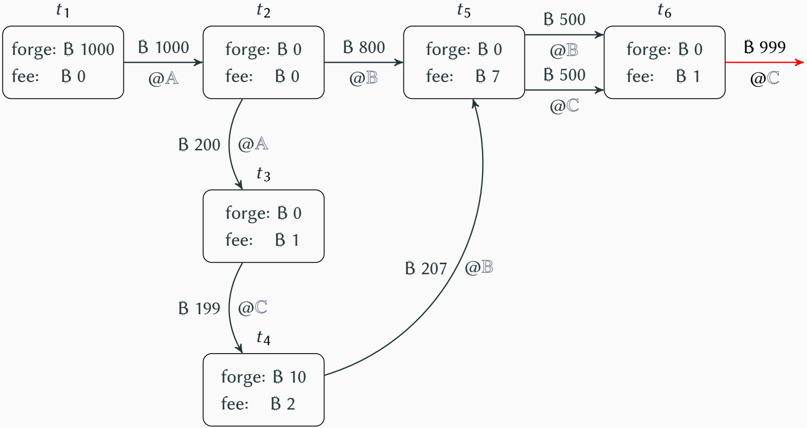
$$
\begin{aligned}
&\textbf{open import}\ \mathit{Data.AVL}\ \mathbb{N}\text{-}\mathit{strictTotalOrder}\\[4pt]
&\_+^{c}\_ : \mathit{Value} \to \mathit{Value} \to \mathit{Value}\\
&c +^{c} c' = \mathit{toList}\ (\mathit{foldl}\ \mathit{go}\ (\mathit{fromList}\ c)\ c')\\
&\quad \textbf{where}\\
&\qquad \mathit{go} : \mathit{Tree}\ \mathit{Hash}\ \mathbb{N} \to (\mathit{Hash} \times \mathbb{N}) \to \mathit{Tree}\ \mathit{Hash}\ \mathbb{N}\\
&\qquad \mathit{go}\ m\ (k\ ,\ v) = \mathit{insertWith}\ k\ ((\_+ v) \circ \mathit{fromMaybe}\ 0)\ m\\[4pt]
&\mathit{sum}^{c} : \mathit{List}\ \mathit{Value} \to \mathit{Value}\\
&\mathit{sum}^{c} = \mathit{foldl}\ \_+^{c}\_\ [\,]
\end{aligned}
$$

Multi-currency: Forging condition

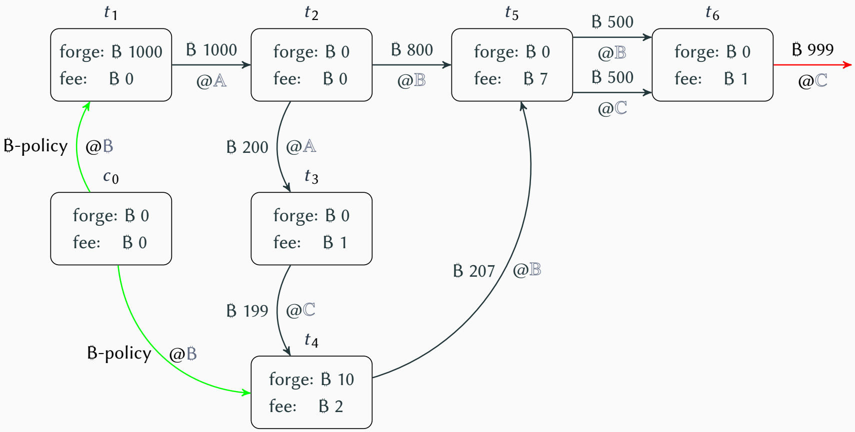
We now need to enforce monoetary policies on forging transactions:

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

Example: Transaction Graph



Example: Transaction Graph + monetary policy



Example: Setting Up

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

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

$$\mathbb{A} = 1 \text{ -- \textit{first address}}$$

$$\mathbb{B} = 2 \text{ -- \textit{second address}}$$

$$\mathbb{C} = 3 \text{ -- \textit{third address}}$$

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

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

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

Example: Smart Constructors

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

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

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

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

$$\_\ @\ \_ : Value \to Index\ addresses \to TxOutput$$
$$v\ @\ addr = \mathbf{record}\ \{ value = v; address = addr; dataScript = \lambda\ \_ \to tt \}$$

Example: Definitions of Transactions

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

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

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

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

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:

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

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

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

Combining Ledgers

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

BitML

$$
\begin{aligned}
&\textbf{module}\ \mathit{BitML}\ (\mathit{Participant} : \mathit{Set})\\
&\qquad\qquad\qquad (\_\overset{?}{=}_{\mathrm{p}}\_ : \mathit{Decidable}\ \{A = \mathit{Participant}\}\ \_\equiv\_)\\
&\qquad\qquad\qquad (\mathit{Honest} : \mathit{List}^{+}\ \mathit{Participant})\ \textbf{where}\\
&\mathit{Time} \quad\ \ = \mathbb{N}\\
&\mathit{Value} \quad\ = \mathbb{N}\\
&\mathit{Secret} \quad = \mathit{String}\\
&\mathit{Deposit} = \mathit{Participant} \times \mathit{Value}
\end{aligned}
$$

Contract Preconditions

```
data Precondition : List Value -- volatile deposits
                  → List Value -- persistent deposits
                  → Set where
  -- volatile deposit
  _?_ : Participant → (v : Value) → Precondition [ v ] [ ]
  -- persistent deposit
  _!_ : Participant → (v : Value) → Precondition [ ] [ v ]
  -- committed secret
  _#_ : Participant → Secret → Precondition [ ] [ ]
  -- conjunction
  _∧_ : Precondition vsᵥ vsₚ → Precondition vsᵥ′ vsₚ′
      → Precondition (vsᵥ ++ vsᵥ′) (vsₚ ++ vsₚ′)
```

```
data Contract : Value -- the monetary value it carries
              → Values -- the volatile deposits it presumes
              → Set where

  -- collect deposits and secrets
  put _ reveal _ if _ ⇒ _ ⊣ _ :
        (vs : List Value) → (s : Secrets) → Predicate s′
    → Contract (v + sum vs) vs′ → s′ ⊆ s
    → Contract v (vs′ ++ vs)

  -- transfer the remaining balance to a participant
  withdraw : ∀ {v vs} → Participant → Contract v vs
```

Contracts II

```
-- split the balance across different branches
split : ∀ {vs} → (cs : List (∃[ v ] Contract v vs))
  → Contract (sum (proj₁ ⟨$⟩ cs)) vs
-- wait for participant's authorization
_:_ : Participant → Contract v vs → Contract v vs
-- wait until some time passes
after_:_ : Time → Contract v vs → Contract v vs
```

```
record Advertisement (v : Value) (vsᶜ vsᵛ vsᵖ : List Value) : Set where
  constructor _⟨_⟩∶-_
  field G     : Precondition vsᵛ vsᵖ
        C     : Contracts v vsᶜ
        valid : length vsᶜ ≤ length vsᵛ
              × participantsᵍ G ++ participantsᶜ C
                  ⊆
                participant <$> persistentDeposits G
              × v ≡ sum vsᵖ
```

Example Advertisement

$$\textbf{open}\ \mathit{BitML}\ (A \mid B)\ \ \ldots\ \ [A]^{+}$$

$$\mathit{ex\text{-}ad} : \mathit{Advertisement}\ 5\ [200]\ [200]\ [3\ ,\ 2]$$

$$\begin{aligned}
\mathit{ex\text{-}ad} = \langle\ & B\,!\,3 \land A\,!\,2 \land A?200\ \rangle \\
& \quad \mathit{split}\ \ (\ 2 \multimap \mathit{withdraw}\ B \\
& \qquad\qquad \oplus\, 2 \multimap \mathit{after}\ 42 : \mathit{withdraw}\ A \\
& \qquad\qquad \oplus\, 1 \multimap \mathit{put}\ [200] \Rightarrow B : \mathit{withdraw}\ \{201\}\ A\ ) \\
& \quad \dashv \ldots
\end{aligned}$$

Small-step Semantics: Actions I

$$
\begin{aligned}
\mathit{AdvertisedContracts} &= \mathit{List}\ (\exists[\, v\,, \ldots, vs^{\mathrm{p}}\,]\ \mathit{Advertisement}\ v\ \ldots\ vs^{\mathrm{p}}) \\
\mathit{ActiveContracts} &= \mathit{List}\ (\exists[\, v\,, vs]\ \mathit{Contracts}\ v\ vs)
\end{aligned}
$$

```
data Action (p : Participant) -- the participant that authorizes this action
  : AdvertisedContracts       -- contract advertisements it requires
  → ActiveContracts           -- active contracts it requires
  → Values                    -- deposits it requires from the participant
  → Deposits                  -- deposits it produces
  → Set where
```

Small-step Semantics: Actions II

```
-- join two deposits deposits
_↔_ : ∀ {vs} → (i : Index vs) → (j : Index vs)
      → Action p [] [] vs (p has_ ⟨$⟩ merge i j vs)
-- commit secrets to stipulate an advertisement
#▷_ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
      → Action p [v , vsᶜ , vsᵛ , vsᵖ , ad] [] [] []
-- spend x to stipulate an advertisement
_▷ˢ_ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ) → (i : Index vsᵖ)
      → Action p [v , vsᶜ , vsᵛ , vsᵖ , ad] [] [vsᵖ !! i] []
-- pick a branch
_▷ᵇ_ : (c : Contracts v vs) → (i : Index c)
      → Action p [] [v , vs , c] [] []
⋮
```

Small-step Semantics: Actions Example

```
-- A spends the required ₿ 2, as stated in the pre-condition
ex-spend : Action A [5 , [200] , [200] , [3 , 2] , ex-ad] [] [2] []
ex-spend = ex-ad ▷ˢ 1
```

```
data Configuration′ : --       current        ×       required
                      AdvertisedContracts × AdvertisedContracts
                    → ActiveContracts     × ActiveContracts
                    → Deposits            × Deposits
                    → Set where

  -- empty
  ∅ : Configuration′ ([] , []) ([] , []) ([] , [])

  -- contract advertisement
  `_ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
     → Configuration′ ([v , vsᶜ , vsᵛ , vsᵖ , ad] , []) ([] , []) ([] , [])

  -- active contract
  ⟨_,_⟩ᶜ : (c : Contracts v vs) → Value
           → Configuration′ ([] , []) ([v , vs , c] , []) ([] , [])
```

Small-step Semantics: Configurations II

```
-- deposit redeemable by a participant
⟨ _ , _ ⟩ᵈ      :  (p : Participant) → (v : Value)
                 → Configuration′ ([] , []) ([] , []) ([p has v] , [])

-- authorization to perform an action
_ [_]   :  (p : Participant) → Action p ads cs vs ds
          → Configuration′ ([] , ads) ([] , cs) (ds , ((p has _) ⟨$⟩ vs))

-- committed secret
⟨ _ : _#_ ⟩  :  Participant → Secret → ℕ ⊎ ⊥
                 → Configuration′ ([] , []) ([] , []) ([] , [])

-- revealed secret
_ : _#_  :  Participant → Secret → ℕ
            → Configuration′ ([] , []) ([] , []) ([] , [])
```

Small-step Semantics: Configurations III

-- parallel composition

$$
\begin{array}{lll}
\_\,|\,\_ & : & \mathit{Configuration'}\ (\mathit{ads}^{\mathsf{l}}\ ,\ \mathit{rads}^{\mathsf{l}})\ (\mathit{cs}^{\mathsf{l}}\ ,\ \mathit{rcs}^{\mathsf{l}})\ (\mathit{ds}^{\mathsf{l}}\ ,\ \mathit{rds}^{\mathsf{l}}) \\
 & \to & \mathit{Configuration'}\ (\mathit{ads}^{\mathsf{r}}\ ,\ \mathit{rads}^{\mathsf{r}})\ (\mathit{cs}^{\mathsf{r}}\ ,\ \mathit{rcs}^{\mathsf{r}})\ (\mathit{ds}^{\mathsf{r}}\ ,\ \mathit{rds}^{\mathsf{r}}) \\
 & \to & \mathit{Configuration'}\ (\mathit{ads}^{\mathsf{l}} +\!\!+\ \mathit{ads}^{\mathsf{r}}\ ,\ \mathit{rads}^{\mathsf{l}} +\!\!+\ (\mathit{rads}^{\mathsf{r}} \setminus \mathit{ads}^{\mathsf{l}})) \\
 & & \qquad (\mathit{cs}^{\mathsf{l}} +\!\!+\ \mathit{cs}^{\mathsf{r}}\ ,\ \mathit{rcs}^{\mathsf{l}} +\!\!+\ (\mathit{rcs}^{\mathsf{r}} \setminus \mathit{cs}^{\mathsf{l}})) \\
 & & \qquad ((\mathit{ds}^{\mathsf{l}} \setminus \mathit{rds}^{\mathsf{r}}) +\!\!+\ \mathit{ds}^{\mathsf{r}}\ ,\ \mathit{rds}^{\mathsf{l}} +\!\!+\ (\mathit{rds}^{\mathsf{r}} \setminus \mathit{ds}^{\mathsf{l}}))
\end{array}
$$

Small-step Semantics: Closed Configurations

$$\mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds} = \mathit{Configuration}'\ (\mathit{ads}\,,[\,])\ (\mathit{cs}\,,[\,])\ (\mathit{ds}\,,[\,])$$

Small-step Semantics: Inference Rules I

$$\textbf{data}\ \_ \longrightarrow \_\ :\ \mathit{Configuration}\ ads\ cs\ ds \to \mathit{Configuration}\ ads'\ cs'\ ds' \to \mathit{Set}\ \textbf{where}$$

$\mathit{DEP\text{-}AuthJoin}$:

$$\langle\, A\,,\, v\,\rangle^{\mathrm{d}} \mid \langle\, A\,,\, v'\,\rangle^{\mathrm{d}} \mid \Gamma \longrightarrow \langle\, A\,,\, v\,\rangle^{\mathrm{d}} \mid \langle\, A\,,\, v'\,\rangle^{\mathrm{d}} \mid A\,[0 \leftrightarrow 1] \mid \Gamma$$

$\mathit{DEP\text{-}Join}$:

$$\langle\, A\,,\, v\,\rangle^{\mathrm{d}} \mid \langle\, A\,,\, v'\,\rangle^{\mathrm{d}} \mid A\,[0\ \leftrightarrow\ 1] \mid \Gamma \longrightarrow \langle\, A\,,\, v + v'\,\rangle^{\mathrm{d}} \mid \Gamma$$

$\mathit{C\text{-}Advertise} : \forall\ \{\Gamma\ ad\}$

$$\to \exists[\,p \in \mathit{participants}^{\mathrm{g}}\ (G\ ad)\,]\ p \in \mathit{Hon}$$

$$\overline{\hphantom{\exists[\,p \in \mathit{participants}^{\mathrm{g}}\ (G\ ad)\,]\ p \in \mathit{Hon}}}$$

$$\to \Gamma \longrightarrow {}`ad \mid \Gamma$$

Small-step Semantics: Inference Rules II

$$\begin{array}{l}
\textit{C-AuthCommit} : \forall \{ A\ ad\ \Gamma \} \\
\quad \to \textit{secrets}\ (G\ ad) \equiv a_1\ \ldots\ a_n \\
\quad \to (A \in \textit{Hon} \to \forall\ [i \in 1\ \ldots\ n]\ a_i \not\equiv \bot) \\
\quad \overline{} \\
\quad \to `ad \mid \Gamma \longrightarrow `ad \mid \Gamma \mid \ldots \langle\, A : a_i \# N_i \,\rangle \ldots \mid A\ [\#\ ad]
\end{array}$$

$$\begin{array}{l}
\textit{C-Control} : \forall \{ \Gamma\ C\ i\ D \} \\
\quad \to C\ !!\ i \equiv A_1 : \ldots : A_n : D \\
\quad \overline{} \\
\quad \to \langle\, C , v \,\rangle^{c} \mid \ldots\ A_i\ [C \rhd^{b}\ i]\ \ldots \mid \Gamma \longrightarrow \langle\, D , v \,\rangle^{c} \mid \Gamma
\end{array}$$

$\vdots$

Small-step Semantics: Timed Inference Rules I

```
record Configurationᵗ ads cs ds : Set where
  constructor _@_
  field cfg  : Configuration ads cs ds
        time : Time

data _—→ₜ_ : Configurationᵗ ads cs ds → Configurationᵗ ads′ cs′ ds′
           → Set where
  Action : ∀ {Γ Γ′ t}
    → Γ —→ Γ′
      ────────────────
    → Γ @ t —→ₜ Γ′ @ t

  Delay : ∀ {Γ t δ}
      ──────────────────────
    → Γ @ t —→ₜ Γ @ (t + δ)
```

Small-step Semantics: Timed Inference Rules II

$$
\begin{aligned}
&\mathit{Timeout} : \forall \{\Gamma\ \Gamma'\ t\ i\ \mathit{contract}\} \\
&\qquad \text{-- all time constraints are satisfied} \\
&\quad \to \mathit{All}\ (\_ \leqslant t)\ (\mathit{timeDecorations}\ (\mathit{contract}\ !!\ i)) \\
&\qquad \text{-- resulting state if we pick this branch} \\
&\quad \to \langle\ [\mathit{contract}\ !!\ i]\ ,\ v\ \rangle^{\mathrm{c}}\ |\ \Gamma \longrightarrow \Gamma' \\
&\quad \overline{\hspace{20em}} \\
&\quad \to (\langle\ \mathit{contract}\ ,\ v\ \rangle^{\mathrm{c}}\ |\ \Gamma)\ @\ t \longrightarrow_{\mathrm{t}} \Gamma'\ @\ t
\end{aligned}
$$

Small-step Semantics: Reordering I

```
_≈_ : Configuration ads cs ds → Configuration ads cs ds → Set
c ≈ c′ = cfgToList c ↭ cfgToList c′
  where
    open import Data.List.Permutation using (_↭_)
    cfgToList ∅                  = []
    cfgToList (l | r)            = cfgToList l ++ cfgToList r
    cfgToList {p₁} {p₂} {p₃} c = [p₁ , p₂ , p₃ , c]
```

Small-step Semantics: Reordering II

$$
\begin{array}{l}
\textit{DEP-AuthJoin} : \\
\qquad \textit{Configuration ads cs } (A \textit{ has } v :: A \textit{ has } v' :: ds) \ni \\
\qquad\quad \Gamma' \approx \langle\, A \,,\, v \,\rangle^{\mathrm{d}} \mid \langle\, A \,,\, v' \,\rangle^{\mathrm{d}} \mid \Gamma \\
\quad \rightarrow \textit{Configuration ads cs } (A \textit{ has } (v + v') :: ds) \ni \\
\qquad\quad \Gamma'' \approx \langle\, A \,,\, v \,\rangle^{\mathrm{d}} \mid \langle\, A \,,\, v' \,\rangle^{\mathrm{d}} \mid A\,[0 \leftrightarrow 1] \mid \Gamma \\
\hline
\quad \rightarrow \Gamma' \longrightarrow \Gamma''
\end{array}
$$

Small-step Semantics: Equational Reasoning

$$
\begin{aligned}
&\textbf{data}\ \_ \longrightarrow^{*} \_ \ :\ \mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds} \to \mathit{Configuration}\ \mathit{ads}'\ \mathit{cs}'\ \mathit{ds}' \\
&\qquad\qquad \to \mathit{Set}\ \textbf{where} \\
&\quad \_\ \square : (M : \mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds}) \to M \longrightarrow^{*} M \\
&\quad \_ \longrightarrow\langle\ \_\ \rangle\_ : \forall\ \{L'\ M\ M'\ N\}\ (L : \mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds}) \\
&\qquad \to \{L \approx L' \times M \approx M'\} \\
&\qquad \to L' \longrightarrow M' \\
&\qquad \to M \longrightarrow^{*} N \\
&\qquad\ \overline{\qquad\qquad\qquad\qquad\qquad} \\
&\qquad \to L \longrightarrow^{*} N \\
&\mathit{begin}\ \_ : \forall\ \{M\ N\} \to M \longrightarrow^{*} N \to M \longrightarrow^{*} N
\end{aligned}
$$

Timed-commitment Protocol

A promises to reveal a secret, otherwise loses deposit.

$$
\begin{aligned}
& tc : \mathit{Advertisement}\ 1\ [\,]\ [\,]\ [1\ ,\ 0])\\
& tc = \langle\ A\ !\ 1 \wedge A\#a \wedge B\ !\ 0\ \rangle\\
& \qquad\qquad \mathit{reveal}\ [a] \Rightarrow \mathit{withdraw}\ A \dashv \ldots\\
& \qquad\quad \oplus\ \mathit{after}\ t : \mathit{withdraw}\ B
\end{aligned}
$$

Small-step Semantics: Example (derivation)

$$tc\text{-}semantics : \langle\, A \,,\, 1 \,\rangle^{\mathrm{d}} \longrightarrow^{*} \langle\, A \,,\, 1 \,\rangle^{\mathrm{d}} \mid A : a\#6$$

$$\begin{array}{ll}
tc\text{-}semantics = & \langle\, A \,,\, 1 \,\rangle^{\mathrm{d}} \\
\longrightarrow\langle\, C\text{-}Advertise \,\rangle & `tc \mid \langle\, A \,,\, 1 \,\rangle^{\mathrm{d}} \\
\longrightarrow\langle\, C\text{-}AuthCommit \,\rangle & `tc \mid \langle\, A \,,\, 1 \,\rangle^{\mathrm{d}} \mid \langle\, A : a \,\#\, 6 \,\rangle \mid A\,[\# \triangleright\ \ tc] \\
\longrightarrow\langle\, C\text{-}AuthInit \,\rangle & `tc \mid \langle\, A \,,\, 1 \,\rangle^{\mathrm{d}} \mid \langle\, A : a \,\#\, 6 \,\rangle \mid A\,[\# \triangleright\ \ tc] \mid A\,[tc \triangleright^{s} 0 \\
\longrightarrow\langle\, C\text{-}Init \,\rangle & \langle\, tc \,,\, 1 \,\rangle^{\mathrm{c}} \mid \langle\, A : a \,\#\ inj_1\ 6 \,\rangle \\
\longrightarrow\langle\, C\text{-}AuthRev \,\rangle & \langle\, tc \,,\, 1 \,\rangle^{\mathrm{c}} \mid A : a \,\#\, 6 \\
\longrightarrow\langle\, C\text{-}Control \,\rangle & \langle\, [reveal\ \ \ldots] \,,\, 1 \,\rangle^{\mathrm{c}} \mid A : a \,\#\, 6 \\
\longrightarrow\langle\, C\text{-}PutRev \,\rangle & \langle\, [withdraw\ A] \,,\, 1 \,\rangle^{\mathrm{c}} \mid A : a \,\#\, 6 \\
\longrightarrow\langle\, C\text{-}Withdraw \,\rangle & \langle\, A \,,\, 1 \,\rangle^{\mathrm{d}} \mid A : a \,\#\, 6
\end{array}$$

$\square$

Symbolic Model: Labelled step relation

$$
\begin{aligned}
\textbf{data}\ \_ \longrightarrow[\![\ \_\ ]\!]\ \_\ \ :\ \ & \mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds} \\
\rightarrow\ & \mathit{Label} \\
\rightarrow\ & \mathit{Configuration}\ \mathit{ads}'\,\mathit{cs}'\,\mathit{ds}' \\
\rightarrow\ & \mathit{Set}\ \textbf{where}
\end{aligned}
$$

$\vdots$

$$
\begin{aligned}
& \mathit{DEP\text{-}AuthJoin} : \\
& \quad \langle\, A\,,\, v \,\rangle^{\mathrm{d}} \mid \langle\, A\,,\, v' \,\rangle^{\mathrm{d}} \mid \Gamma \\
& \longrightarrow[\![\ \mathit{auth\text{-}join}\ [A\,,\, 0 \leftrightarrow 1]\ ]\!] \\
& \quad \langle\, A\,,\, v \,\rangle^{\mathrm{d}} \mid \langle\, A\,,\, v' \,\rangle^{\mathrm{d}} \mid A\ [0 \leftrightarrow 1] \mid \Gamma
\end{aligned}
$$

$\vdots$

$$
\begin{array}{l}
\textbf{data}\ \mathit{Trace} : \mathit{Set}\ \textbf{where} \\
\quad \_\bullet \qquad\qquad\quad : \exists \mathit{TimedConfiguration} \to \mathit{Trace} \\
\quad \_::[\![\_]\!]\_ : \exists \mathit{TimedConfiguration} \to \mathit{Label} \to \mathit{Trace} \to \mathit{Trace}
\end{array}
$$

$$
\begin{array}{l}
\_ \rightarrowtail [\![\_]\!]\_ : \mathit{Trace} \to \mathit{Label} \to \exists \mathit{TimedConfiguration} \to \mathit{Set} \\
R \rightarrowtail [\![\ \alpha\ ]\!]\ (\_, \_, \_, tc') \\
\quad = \mathit{proj}_2\ (\mathit{proj}_2\ (\mathit{proj}_2\ (\mathit{lastCfg}\ R))) \longrightarrow [\![\ \alpha\ ]\!]\ tc'
\end{array}
$$

Symbolic Model: Strategies (honest participant)

```
record HonestStrategy (A : Participant) : Set where
  field
    strategy : Trace → List Label
    valid    : A ∈ Hon
             × (∀ R α → α ∈ strategy R ∗ →
                  ∃[R′] (R ↣⟦ α ⟧ R′))
             × (∀ R α → α ∈ strategy R ∗ →
                  All (_ ≡ A) (authDecoration α))
             ⋮

HonestStrategies = ∀ {A} → A ∈ Hon → HonestStrategy A
```

Symbolic Model: Strategies (adversary)

$$
\begin{array}{l}
\textbf{record}\ \mathit{AdversarialStrategy}\ (\mathit{Adv} : \mathit{Participant}) : \mathit{Set}\ \textbf{where} \\
\quad \textbf{field} \\
\qquad \mathit{strategy} \ : \ \mathit{Trace} \to \mathit{List}\ (\mathit{Participant} \times \mathit{List}\ \mathit{Label}) \to \mathit{Label} \\
\qquad \mathit{valid} \quad\ : \ \mathit{Adv} \notin \mathit{Hon} \\
\qquad\qquad \times\ \forall\ \{R : \mathit{Trace}\}\ \{\mathit{moves} : \mathit{List}\ (\mathit{Participant} \times \mathit{List}\ \mathit{Label})\} \to \\
\qquad\qquad\quad \textbf{let}\ \alpha = \mathit{strategy}\ R * \ \mathit{moves}\ \textbf{in} \\
\qquad\qquad\quad (\ \ \exists[A]\ \ (\ A \in \mathit{Hon} \\
\qquad\qquad\qquad\qquad \times\ \mathit{authDecoration}\ \alpha \equiv \mathit{just}\ A \\
\qquad\qquad\qquad\qquad \times\ \alpha \in \mathit{concatMap}\ \mathit{proj}_2\ \mathit{moves}\ ) \\
\qquad\qquad\quad \uplus\ (\ \ \mathit{authDecoration}\ \alpha \equiv \mathit{nothing} \\
\qquad\qquad\qquad \times\ (\forall\ \delta \to \alpha \not\equiv \mathit{delay}\ [\delta]) \\
\qquad\qquad\qquad \times\ \exists[R']\ (R \rightarrowtail\llbracket\ \alpha\ \rrbracket\ R')) \\
\qquad\qquad\quad \vdots \\
\qquad\qquad\quad)
\end{array}
$$

Symbolic Model: Adversary makes final choice

$$
\begin{array}{l}
\mathit{runAdversary} : \mathit{Strategies} \to \mathit{Trace} \to \mathit{Label} \\
\mathit{runAdversary}\ (S\dagger\ ,\ S)\ R = \mathit{strategy}\ S\dagger\ R \ast\ \ \mathit{honestMoves} \\
\quad \textbf{where} \\
\qquad \mathit{honestMoves} = \mathit{mapWith}{\in}\ \mathit{Hon}\ (\lambda\ \{A\}\ p \to A\ ,\ \mathit{strategy}\ (S\ p)\ R{\ast})
\end{array}
$$

$$\textbf{data}\ \_\text{-conforms-to-}\_ : \mathit{Trace} \to \mathit{Strategies} \to \mathit{Set}\ \textbf{where}$$

$$\mathit{base} : \forall\ \{\Gamma : \mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds}\}\ \{\mathit{SS} : \mathit{Strategies}\}$$
$$\to \mathit{Initial}\ \Gamma$$
$$\overline{}$$
$$\to (\mathit{ads}\ ,\ \mathit{cs}\ ,\ \mathit{ds}\ ,\ \Gamma\ @\ 0)^{\bullet}\ \text{-conforms-to-}\ \mathit{SS}$$

$$\mathit{step} : \forall\ \{R : \mathit{Trace}\}\ \{T' : \exists\mathit{TimedConfiguration}\}\ \{\mathit{SS} : \mathit{Strategies}\}$$
$$\to R\ \text{-conforms-to-}\ \mathit{SS}$$
$$\to R \rightarrowtail [\![\ \mathit{runAdversary}\ \mathit{SS}\ R\ ]\!]\ T'$$
$$\overline{}$$
$$\to (T' :: [\![\ \mathit{runAdversary}\ \mathit{SS}\ R\ ]\!]\ R)\ \text{-conforms-to-}\ \mathit{SS}$$

Symbolic Model: Meta-theory

- *strip-preserves-semantics* :

$$
\begin{array}{l}
(\forall\ A\ s \quad\ \to \alpha \not\equiv \mathit{auth\text{-}rev}\ [A\ ,\ s]) \to \\
(\forall\ A\ ad\ \Delta \to \alpha \not\equiv \mathit{auth\text{-}commit}\ [A\ ,\ ad\ ,\ \Delta]) \\
\ \to (\forall\ T' \ \ \to \dfrac{R \rightarrowtail [\![\ \alpha\ ]\!]\ T'}{\to R{*} \rightarrowtail [\![\ \alpha\ ]\!]\ T'{*})} \\
\ \times\ (\forall\ T' \ \ \to \dfrac{R{*} \rightarrowtail [\![\ \alpha\ ]\!]\ T'}{\to \exists [\, T'' \,]\ (R \rightarrowtail [\![\ \alpha\ ]\!]\ T'') \times (T'{*} \equiv T''{*})}
\end{array}
$$

- *adversarial-move-is-semantic* :

$$
\exists [\, T' \,]\ (R \rightarrowtail [\![\ \mathit{runAdversary}\ (S\dagger\ ,\ S)\ R\ ]\!]\ T')
$$

BitML Paper Fixes

Discrepancies in inference rules

e.g. forgetting surrounding context Γ

Non-linear derivations

If one of the hypothesis is another step, we lose equational-style linearity.
Solution: Move result state of the hypothesis to the result of the rule.

Missed assumptions

The original formulation of the *strip-preserves-semantics* lemma required only that the action does not reveal secrets (*C-AuthRev*), but it should not commit secrets either (*C-AuthCommit*).

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: BitML

1. A lot of proof obligations associated with most datatypes
   - Implement decision procedures for them, just like we did for UTxO
2. Computational model
   - Formulation very similar to the symbolic model we already have, but a lot of additional details to handle
3. Compilation correctness: *Symbolic Model* $\approx$ *Computational Model*
   - Compile to abstract UTxO model instead of concrete Bitcoin transactions?
   - Already successfully employed by Marlowe
   - Data scripts stateful capabilities fit well for state transition systems!

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

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?