Formalizing Extended UTxO and BitML Calculus in Agda

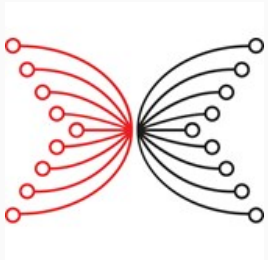
Towards formal verification for smart contracts

Orestis Melkonian

July 8, 2019

Utrecht University, The Netherlands

Supervised by: *Wouter Swierstra* (UU), *Manuel M.T. Chakravarty* (IOHK)
2nd examiner: *Gabriele Keller* (UU)



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

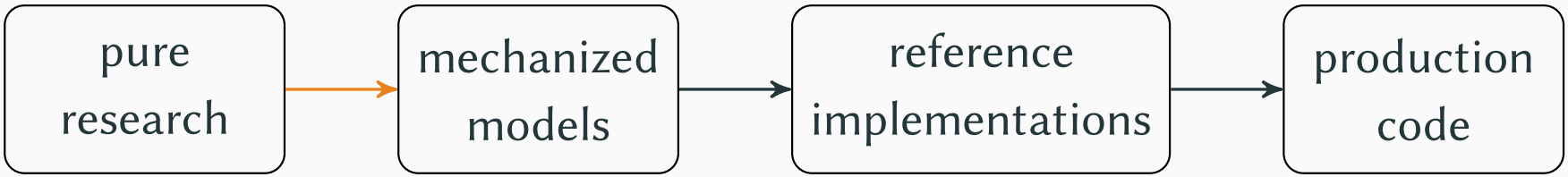
- Based on the notion of accounts
- Smart contracts in (almost) Turing-complete Solidity/EVM

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

```
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

```
validate : PendingTx
         → (i : TxInput)
         → (o : TxOutput)
         → D i ≡ Data o
         → State
         → Bool
validate ptx i o refl st =
  validator i st (value o) ptx (redeemer i st) (dataScript o st)
```

Unspent Outputs

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

where

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

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 ≡ 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:

```
data ValidLedger : Ledger → Set where
  •                : ValidLedger []
  _⊕_⊣_            : ValidLedger l
                   → (tx : Tx)
                   → IsValidTx tx l
                   → ValidLedger (tx :: l)
```

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: $Value = List\ (Hash \times \mathbb{N})$
2. Implement additive group operators (on top of AVL trees):

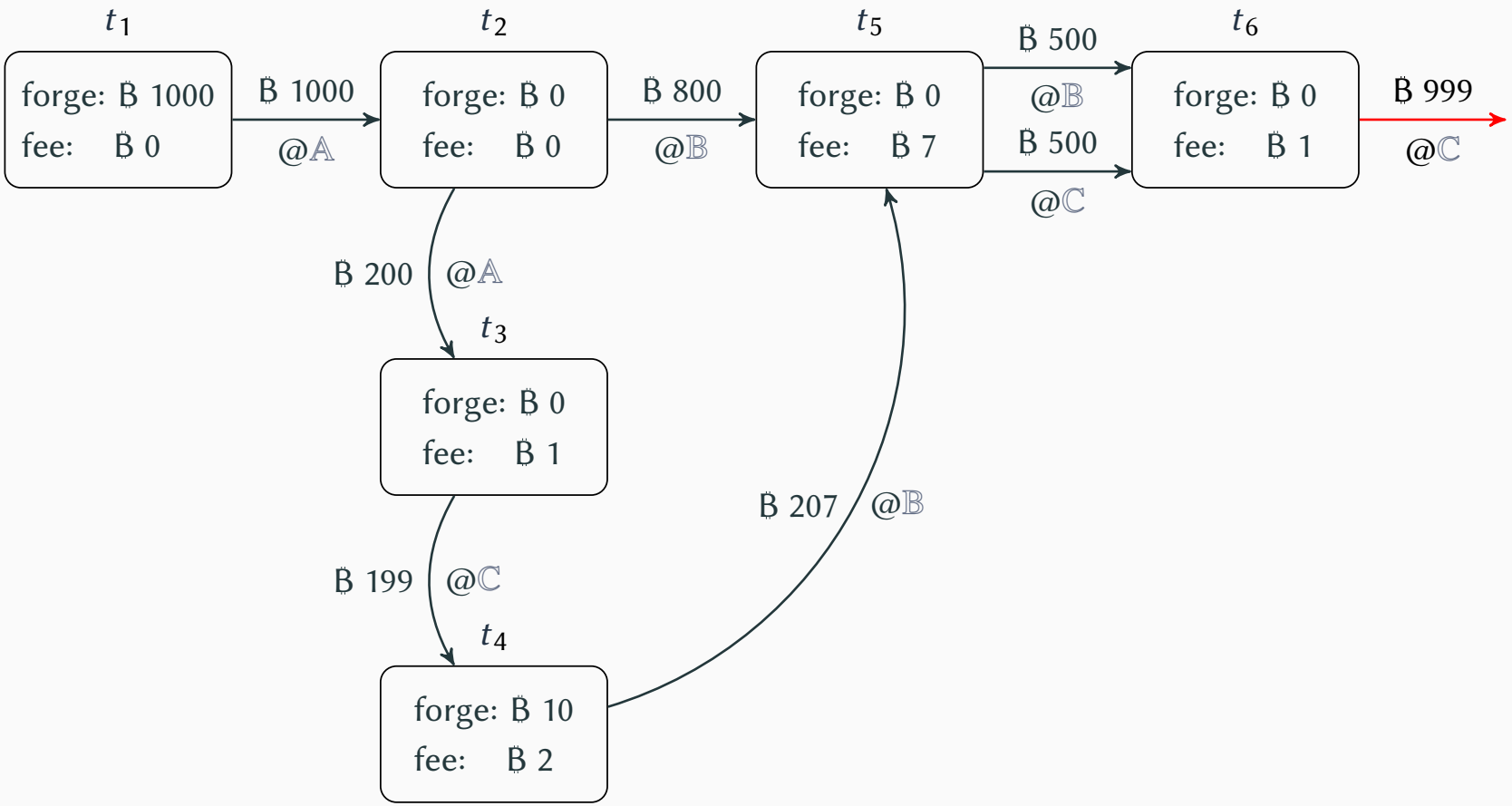
```
open import Data.AVL ℕ-strictTotalOrder

_ +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

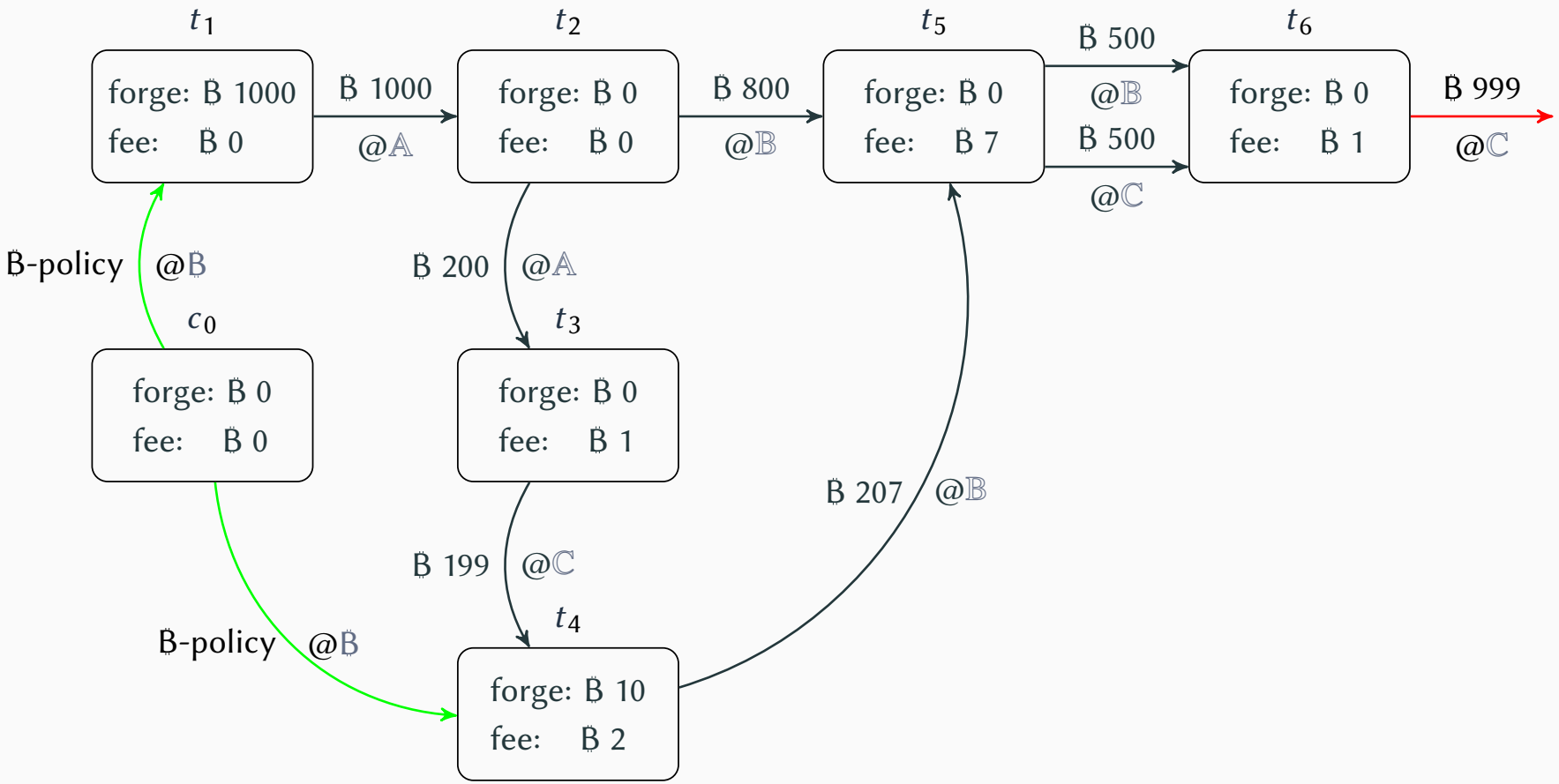
We now need to enforce monetary policies on forging transactions:

```
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

$$Address = \mathbb{N}$$

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

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

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

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

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

$$\mathbb{B}\text{-}validator : State \to \ldots \to Bool$$

$$\mathbb{B}\text{-}validator\ (\textbf{record}\ \{height = h\})\ \ldots = (h \equiv^{b} 1) \vee (h \equiv^{b} 4)$$

Example: Smart Constructors

```
mkValidator : TxOutputRef → (... → TxOutputRef → ... → Bool)
mkValidator o ... o' ... = o ≡ᵇ o'

₿ _ : ℕ → Value
₿ v = [(₿-validator# , v)]

withScripts : TxOutputRef → TxInput
withScripts o = record {outputRef = o
                        ; redeemer  = λ _ → o
                        ; validator = mkValidator tin}

withPolicy : TxOutputRef → TxInput
withPolicy tin = record {outputRef = tin
                         ; redeemer = λ _ → tt
                         ; validator = ₿-validator}

_ @ _ : Value → Index addresses → TxOutput
v @ addr = record {value = v; address = addr; dataScript = λ _ → tt}
```

Example: Definitions of Transactions

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

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

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

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

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:

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

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

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

BitML

Basic Types

```
module BitML (Participant : Set)
             (_ ≟ₚ _ : Decidable {A = Participant} _ ≡ _)
             (Honest : List⁺ Participant) where
```

$$
\begin{aligned}
\mathit{Time} &= \mathbb{N} \\
\mathit{Value} &= \mathbb{N} \\
\mathit{Secret} &= \mathit{String} \\
\mathit{Deposit} &= \mathit{Participant} \times \mathit{Value}
\end{aligned}
$$

Contract Preconditions

```
data Precondition : Values -- volatile deposits
                  → Values -- 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′)
```

Contracts I

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

  -- collect deposits and secrets
  put _ reveal _ if _ ⇒ _ ⊣ _ :
        (vs : Values) → (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
```

Advertisements

```
record Advertisement (v : Value) (vs^c vs^v vs^p : Values) : Set where
  constructor _⟨ _ ⟩⊣ _
  field G     : Precondition vs^v vs^p
        C     : Contracts v vs^c
        valid : length vs^c ⩽ length vs^v
              × participants^g G ⨣ participants^c C
                  ⊆
                participant ⟨$⟩ persistentDeposits G
              × v ≡ sum vs^p
```

Example Advertisement

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

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

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

Small-step Semantics: Actions I

$$AdvertisedContracts = List\ (\exists[\, v\,, \ldots, vs^{p}\,]\ Advertisement\ v\ \ldots\ vs^{p})$$
$$ActiveContracts\ \ = List\ (\exists[\, v\,, vs]\ Contracts\ v\ vs)$$

```
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

$$
\begin{aligned}
&\textit{-- join two deposits}\\
&\_ \leftrightarrow \_ : \forall\ \{vs\} \to (i : \mathit{Index}\ vs) \to (j : \mathit{Index}\ vs)\\
&\qquad \to \mathit{Action}\ p\ [\,]\ [\,]\ vs\ (p\ \mathit{has}\_\ \langle\$\rangle\ \mathit{merge}\ i\ j\ vs)\\
&\textit{-- commit secrets to stipulate an advertisement}\\
&\#\triangleright\ \_ : (ad : \mathit{Advertisement}\ v\ vs^{c}\ vs^{v}\ vs^{p})\\
&\qquad \to \mathit{Action}\ p\ [v\,, vs^{c}\,, vs^{v}\,, vs^{p}\,, ad]\ [\,]\ [\,]\ [\,]\\
&\textit{-- spend x to stipulate an advertisement}\\
&\_ \triangleright^{s} \_ : (ad : \mathit{Advertisement}\ v\ vs^{c}\ vs^{v}\ vs^{p}) \to (i : \mathit{Index}\ vs^{p})\\
&\qquad \to \mathit{Action}\ p\ [v\,, vs^{c}\,, vs^{v}\,, vs^{p}\,, ad]\ [\,]\ [vs^{p}\ !!\ i]\ [\,]\\
&\textit{-- pick a branch}\\
&\_ \triangleright^{b} \_ : (c : \mathit{Contracts}\ v\ vs) \to (i : \mathit{Index}\ c)\\
&\qquad \to \mathit{Action}\ p\ [\,]\ [v\,, vs\,, c]\ [\,]\ [\,]\\
&\vdots
\end{aligned}
$$

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
```

Small-step Semantics: Configurations I

```
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

$$
\begin{aligned}
&\textit{-- parallel composition}\\
&\_ \mid \_ \;:\; \textit{Configuration}'\ (ads^{\mathrm{l}}\,, rads^{\mathrm{l}})\ (cs^{\mathrm{l}}\,, rcs^{\mathrm{l}})\ (ds^{\mathrm{l}}\,, rds^{\mathrm{l}})\\
&\qquad \to \textit{Configuration}'\ (ads^{\mathrm{r}}\,, rads^{\mathrm{r}})\ (cs^{\mathrm{r}}\,, rcs^{\mathrm{r}})\ (ds^{\mathrm{r}}\,, rds^{\mathrm{r}})\\
&\qquad \to \textit{Configuration}'\ (ads^{\mathrm{l}} +\!\!+ ads^{\mathrm{r}} \quad\quad\ , rads^{\mathrm{l}} +\!\!+ (rads^{\mathrm{r}} \setminus ads^{\mathrm{l}}))\\
&\qquad\qquad\qquad\quad\ (cs^{\mathrm{l}} +\!\!+ cs^{\mathrm{r}} \quad\quad\ , rcs^{\mathrm{l}} +\!\!+ (rcs^{\mathrm{r}} \setminus cs^{\mathrm{l}}))\\
&\qquad\qquad\qquad\quad\ ((ds^{\mathrm{l}} \setminus rds^{\mathrm{r}}) +\!\!+ ds^{\mathrm{r}}\,, rds^{\mathrm{l}} +\!\!+ (rds^{\mathrm{r}} \setminus ds^{\mathrm{l}}))
\end{aligned}
$$

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

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

$$\mathit{DEP\text{-}AuthJoin}: \langle\, A\,, v\,\rangle^{\mathrm{d}} \mid \langle\, A\,, v'\,\rangle^{\mathrm{d}} \mid \Gamma \longrightarrow \langle\, A\,, v\,\rangle^{\mathrm{d}} \mid \langle\, A\,, v'\,\rangle^{\mathrm{d}} \mid A\,[0 \leftrightarrow 1] \mid \Gamma$$

$$\mathit{DEP\text{-}Join}: \langle\, A\,, v\,\rangle^{\mathrm{d}} \mid \langle\, A\,, v'\,\rangle^{\mathrm{d}} \mid A\,[0 \leftrightarrow 1] \mid \Gamma \longrightarrow \langle\, A\,, v + v'\,\rangle^{\mathrm{d}} \mid \Gamma$$

$$\mathit{C\text{-}Advertise}: \forall\ \{\Gamma\ \mathit{ad}\} \to \exists[\, p \in \mathit{participants}^{\mathrm{g}}\ (G\ \mathit{ad})\,]\ p \in \mathit{Hon} \to \Gamma \longrightarrow \text{`}\mathit{ad} \mid \Gamma$$

Small-step Semantics: Inference Rules II

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

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

$\vdots$

Small-step Semantics: Timed Inference Rules I

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

data _ ⟶_t _ : Configuration^t ads cs ds → Configuration^t 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

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

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

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

Small-step Semantics: Equational Reasoning

```
data _ ⟶* _ : Configuration ads cs ds → Configuration ads′ cs′ ds′
                → Set where

  _ □ : (M : Configuration ads cs ds) → M ⟶* M

  _ ⟶⟨ _ ⟩_ : ∀ {L′ M M′ N} (L : Configuration ads cs ds)
    → {L ≈ L′ × M ≈ M′}
    → L′ ⟶ M′
    → M ⟶* N
      ---------------------------
    → L ⟶* N

begin _ : ∀ {M N} → M ⟶* N → M ⟶* N
```

Small-step Semantics: Example (contract)

Timed-commitment Protocol

A promises to reveal a secret, otherwise loses deposit.

$$tc : \mathit{Advertisement}\ 1\ [\,]\ [\,]\ [1\,,0])$$

$$tc = \langle\ A\ !\ 1 \wedge A\#a \wedge B\ !\ 0\ \rangle$$

$$\mathit{reveal}\ [a] \Rightarrow \mathit{withdraw}\ A \dashv \ldots$$

$$\oplus\ \mathit{after}\ t : \mathit{withdraw}\ B$$

Small-step Semantics: Example (derivation)

$$
\begin{aligned}
&\mathit{tc\text{-}semantics} : \langle\, A\,,\,1\,\rangle^{\mathrm{d}} \longrightarrow^{*} \langle\, A\,,\,1\,\rangle^{\mathrm{d}} \mid A : a\#6 \\
&\mathit{tc\text{-}semantics} = \quad \langle\, A\,,\,1\,\rangle^{\mathrm{d}} \\
&\longrightarrow \langle\, \mathit{C\text{-}Advertise}\,\rangle \quad {}^{\backprime}\mathit{tc} \mid \langle\, A\,,\,1\,\rangle^{\mathrm{d}} \\
&\longrightarrow \langle\, \mathit{C\text{-}AuthCommit}\,\rangle {}^{\backprime}\mathit{tc} \mid \langle\, A\,,\,1\,\rangle^{\mathrm{d}} \mid \langle\, A : a \# 6\,\rangle \mid A\,[\#\triangleright \mathit{tc}] \\
&\longrightarrow \langle\, \mathit{C\text{-}AuthInit}\,\rangle \quad {}^{\backprime}\mathit{tc} \mid \langle\, A\,,\,1\,\rangle^{\mathrm{d}} \mid \langle\, A : a \# 6\,\rangle \mid A\,[\#\triangleright \mathit{tc}] \mid A\,[\mathit{tc} \triangleright^{s} 0] \\
&\longrightarrow \langle\, \mathit{C\text{-}Init}\,\rangle \quad \langle\, \mathit{tc}\,,\,1\,\rangle^{\mathrm{c}} \mid \langle\, A : a \# \mathit{inj}_1\ 6\,\rangle \\
&\longrightarrow \langle\, \mathit{C\text{-}AuthRev}\,\rangle \quad \langle\, \mathit{tc}\,,\,1\,\rangle^{\mathrm{c}} \mid A : a \# 6 \\
&\longrightarrow \langle\, \mathit{C\text{-}Control}\,\rangle \quad \langle\, [\mathit{reveal}\ \ldots]\,,\,1\,\rangle^{\mathrm{c}} \mid A : a \# 6 \\
&\longrightarrow \langle\, \mathit{C\text{-}PutRev}\,\rangle \quad \langle\, [\mathit{withdraw}\ A]\,,\,1\,\rangle^{\mathrm{c}} \mid A : a \# 6 \\
&\longrightarrow \langle\, \mathit{C\text{-}Withdraw}\,\rangle \quad \langle\, A\,,\,1\,\rangle^{\mathrm{d}} \mid A : a \# 6
\end{aligned}
$$

□

Symbolic Model: Labelled step relation

$$
\begin{aligned}
&\textbf{data}\ \_ \longrightarrow\!\!\llbracket\ \_\ \rrbracket\ \_\ :\ \mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds} \\
&\qquad\qquad \rightarrow \mathit{Label} \\
&\qquad\qquad \rightarrow \mathit{Configuration}\ \mathit{ads}'\ \mathit{cs}'\ \mathit{ds}' \\
&\qquad\qquad \rightarrow \mathit{Set}\ \textbf{where} \\
&\quad \vdots \\
&\quad \mathit{DEP\text{-}AuthJoin} : \\
&\qquad \langle\ A\ ,\ v\ \rangle^{\mathrm{d}}\ |\ \langle\ A\ ,\ v'\ \rangle^{\mathrm{d}}\ |\ \Gamma \\
&\quad \longrightarrow\!\!\llbracket\ \mathit{auth\text{-}join}\ [A\ ,\ 0 \leftrightarrow 1]\ \rrbracket \\
&\qquad \langle\ A\ ,\ v\ \rangle^{\mathrm{d}}\ |\ \langle\ A\ ,\ v'\ \rangle^{\mathrm{d}}\ |\ A\ [0 \leftrightarrow 1]\ |\ \Gamma \\
&\quad \vdots
\end{aligned}
$$

Symbolic Model: Traces

$$\textbf{data}\ \mathit{Trace} : \mathit{Set}\ \textbf{where}$$

$$\_^{\bullet} \quad : \exists \mathit{TimedConfiguration} \to \mathit{Trace}$$

$$\_ :: [\![\ \_\ ]\!]\ \_ : \exists \mathit{TimedConfiguration} \to \mathit{Label} \to \mathit{Trace} \to \mathit{Trace}$$

$$\_ \rightarrowtail [\![\ \_\ ]\!]\ \_ : \mathit{Trace} \to \mathit{Label} \to \exists \mathit{TimedConfiguration} \to \mathit{Set}$$

$$R \rightarrowtail [\![\ \alpha\ ]\!]\ (\_ , \_ , \_ , tc') = \mathit{proj}_2\ (\mathit{proj}_2\ (\mathit{proj}_2\ (\mathit{lastCfg}\ R))) \longrightarrow [\![\ \alpha\ ]\!]\ tc'$$

Symbolic Model: Strategies (honest participant)

$$
\begin{aligned}
&\textbf{record}\ \mathit{HonestStrategy}\ (A : \mathit{Participant}) : \mathit{Set}\ \textbf{where}\\
&\quad \textbf{field}\\
&\qquad \mathit{strategy} \;:\; \mathit{Trace} \to \mathit{List}\ \mathit{Label}\\
&\qquad \mathit{valid} \;\;\;:\; A \in \mathit{Hon}\\
&\qquad\qquad \times\ (\forall\ R\ \alpha \to \alpha \in \mathit{strategy}\ R\ast \to\\
&\qquad\qquad\qquad \exists[\,R'\,]\ (R \rightarrowtail\!\!\!\!\!\!\rightarrow[\![\ \alpha\ ]\!]\ R'))\\
&\qquad\qquad \times\ (\forall\ R\ \alpha \to \alpha \in \mathit{strategy}\ R\ast \to\\
&\qquad\qquad\qquad \mathit{All}\ (\_ \equiv A)\ (\mathit{authDecoration}\ \alpha))\\
&\qquad\qquad \vdots
\end{aligned}
$$

$$
\mathit{HonestStrategies} = \forall\ \{A\} \to A \in \mathit{Hon} \to \mathit{HonestStrategy}\ A
$$

Symbolic Model: Strategies (adversary)

```
record AdversarialStrategy (Adv : Participant) : Set where
  field
    strategy : Trace → List (Participant × List Label) → Label

    valid    : Adv ∉ Hon
             × ∀ {R : Trace} {moves : List (Participant × List Label)} →
               let α = strategy R ∗ moves in
               (  ∃[A]  ( A ∈ Hon
                        × authDecoration α ≡ just A
                        × α ∈ concatMap proj₂ moves )
                ⊎ (  authDecoration α ≡ nothing
                   × (∀ δ → α ≢ delay [δ])
                   × ∃[R'] (R ↣⟦ α ⟧ R')))
               ⋮
               )
```

Symbolic Model: Adversary makes final choice

```
runAdversary : Strategies → Trace → Label
runAdversary (S† , S) R = strategy S† R ∗  honestMoves
  where
    honestMoves = mapWith∈ Hon (λ {A} p → A , strategy (S p) R∗)
```

Symbolic Model: Conformance

```
data _ -conforms-to- _ : Trace → Strategies → Set where

  base : ∀ {Γ : Configuration ads cs ds} {SS : Strategies}
    → Initial Γ
      ---------------------------------------
    → (ads , cs , ds , Γ @ 0)• -conforms-to- SS

  step : ∀ {R : Trace} {T′ : ∃TimedConfiguration} {SS : Strategies}
    → R -conforms-to- SS
    → R ↣⟦ runAdversary SS R ⟧ T′
      ---------------------------------------
    → (T′ :: ⟦ runAdversary SS R ⟧ R) -conforms-to- SS
```

Symbolic Model: Meta-theory

- *strip-preserves-semantics* :

$$
\begin{aligned}
&(\forall\, A\ s \rightarrow \alpha \not\equiv \mathit{auth\text{-}rev}\,[A\,,\,s]) \rightarrow \\
&(\forall\, A\ ad\ \Delta \rightarrow \alpha \not\equiv \mathit{auth\text{-}commit}\,[A\,,\,ad\,,\,\Delta]) \\
&\rightarrow (\forall\, T' \rightarrow \frac{R \rightarrowtail [\![\ \alpha\ ]\!]\ T'}{\rightarrow R{*} \rightarrowtail [\![\ \alpha\ ]\!]\ T'{*})} \\
&\times\ (\forall\, T' \rightarrow \frac{R{*} \rightarrowtail [\![\ \alpha\ ]\!]\ T'}{\rightarrow \exists[\,T''\,]\ (R \rightarrowtail [\![\ \alpha\ ]\!]\ T'') \times (T'{*} \equiv T''{*})}
\end{aligned}
$$

- *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?