Formal specification of the Cardano ledger, mechanized in Agda

Andre Knispel, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, Ulf Norell

7 April 2024, FMBC, Luxembourg

Introduction

Motivation

Much (formally-verified) meta-theoretical work on EUTxO

- ...but all on simplified and idealized settings
- ...none cover the full feature set of a realistic blockchain like Cardano

Motivation

Much (formally-verified) meta-theoretical work on EUTxO

- ...but all on simplified and idealized settings
- ...none cover the full feature set of a realistic blockchain like Cardano

Previous iterations of Cardano's ledger specification written informally on paper

- Lack the rigor of a mechanized formal artifact
- Are not executable and thus require a separate prototype to be implemented

Methodology

Separation of Concerns

- Networking: deals with sending messages across the internet
- Consensus: establishes a common order of valid blocks
- Ledger: decides whether a sequence of blocks is valid

Methodology

Separation of Concerns

- Networking: deals with sending messages across the internet
- Consensus: establishes a common order of valid blocks
- Ledger: decides whether a sequence of blocks is valid

Stay as close to the previous (LaTeX) formulation as possible

- Use of set theory
- System evolution formulated as state machines

Methodology

Separation of Concerns

- Networking: deals with sending messages across the internet
- Consensus: establishes a common order of valid blocks
- Ledger: decides whether a sequence of blocks is valid

Stay as close to the previous (LaTeX) formulation as possible

- Use of set theory
- System evolution formulated as state machines

Use the Agda proof assistant to produce a readable mechanized specification.

Agda Preliminaries

Axiomatic Set theory

Easy to define common operations:

```
_⊆_ _≡ᵉ_ : {A : Type} → ℙ A → ℙ A → Type
X ⊆  Y = ∀ {x} → x ∈ X → x ∈ Y
X ≡ᵉ Y = X ⊆ Y × Y ⊆ X
```

Axiomatic Set theory

Easy to define common operations:

```
_⊆_ _≡ᵉ_ : {A : Type} → ℙ A → ℙ A → Type
X ⊆  Y = ∀ {x} → x ∈ X → x ∈ Y
X ≡ᵉ Y = X ⊆ Y × Y ⊆ X
```

Finite maps as set of tuples:

```
_⇀_ : Type → Type → Type
A ⇀ B = ∃ λ (ℛ : ℙ (A × B)) →
  ∀ {a b b'} → (a , b) ∈ ℛ → (a , b') ∈ ℛ → b ≡ b'
```

Type of transitions

$$\Gamma \vdash s \xrightarrow[X]{b} s'$$

Type of transitions

$$\Gamma \vdash s \xrightarrow[X]{b} s'$$

```
_⊢_→⦇_⦈_ : Env → State → Signal → State → Type
```

Depicting transitions: Triptychs

Environments (Signals)	States

Possible transitions

Sequencing transitions: Reflexive-transitive closure

```
data _⊢_→⦅_⦆*_ : Env → State → List Signal → State → Type where
                                        step :
  base :                                  • Γ ⊢ s  →⦅ b      ⦆  s'
                                          • Γ ⊢ s' →⦅ bs     ⦆* s''
       ─────────────                        ──────────────────────────
       Γ ⊢ s →⦅ [] ⦆* s                     Γ ⊢ s  →⦅ b :: bs ⦆* s''
```

Formalization

Basic entities / assumptions

- Crypto:

```
VKey Sig Ser : Type
isSigned : VKey → Ser → Sig → Type
```

Basic entities / assumptions

- Crypto:

```
VKey Sig Ser : Type
isSigned : VKey → Ser → Sig → Type
```

- Addresses:

```
Credential = KeyHash ⊎ ScriptHash
```

```
record BaseAddr : Type where
  pay   : Credential
  stake : Credential
```

```
record RwdAddr : Type where
  stake : Credential
```

```
Addr = BaseAddr ⊎ …
```

Basic entities / assumptions

- Crypto:

```
VKey Sig Ser : Type
isSigned : VKey → Ser → Sig → Type
```

- Addresses:

```
Credential = KeyHash ⊎ ScriptHash
```

```
record BaseAddr : Type where
  pay   : Credential
  stake : Credential
```

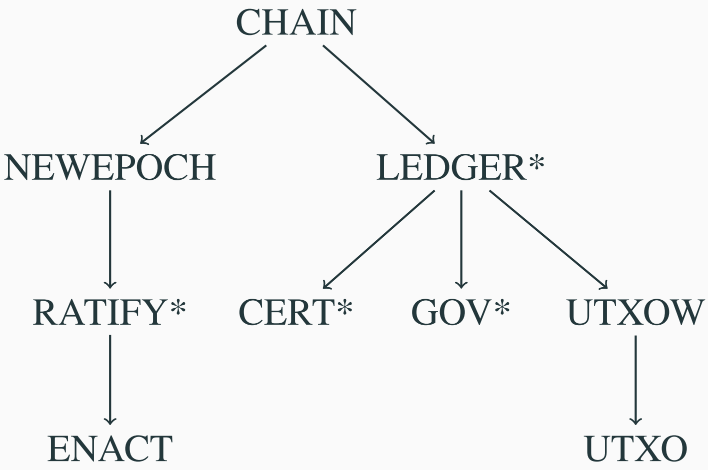
```
record RwdAddr : Type where
  stake : Credential
```

```
Addr = BaseAddr ⊎ …
```

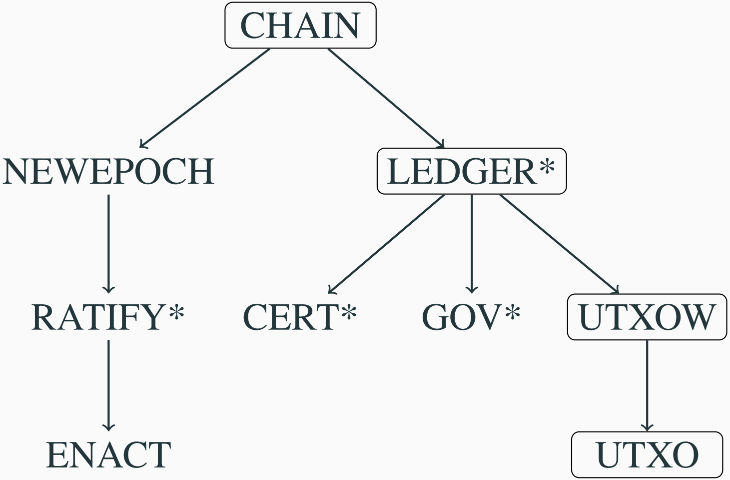
- Tunable parameters:

```
record PParams : Type where
  maxBlockSize maxTxSize a b : ℕ
```

The hierarchy of transitions



The hierarchy of transitions



CHAIN: processing blocks

```
record Block : Type where
  field ts   : List Tx
        slot : Slot
```

```
record NewEpochState : Type where
  field lastEpoch : Epoch; acnt : Acnt
        ls : LState; es : EnactState
        fut : RatifyState

record ChainState : Type where
  field newEpochState : NewEpochState
```

```
CHAIN :
  • mkNewEpochEnv s ⊢ s .newEpochState ⇀⦇ epoch slot ,NEWEPOCH⦈ nes
  • ⟦ slot ⊗ constitution .proj₁ .proj₂ ⊗ pparams .proj₁ ⊗ es ⟧
    ⊢ nes .ls ⇀⦇ ts ,LEDGER*⦈ ls'
  ────────────────────────────────────────
  _ ⊢ s ⇀⦇ b ,CHAIN⦈ updateChainState s nes
```

LEDGER: processing transactions

```
record LEnv : Type where
  slot        : Slot
  ppolicy     : Maybe ScriptHash
  pparams     : PParams
  enactState  : EnactState
```

```
record LState : Type where
  utxoSt    : UTxOState
  govSt     : GovState
  certState : CertState
```

```
LEDGER :
  • mkUTxOEnv Γ ⊢ utxoSt ⇀⦇ tx ,UTXOW⦈ utxoSt'
  • ⟦ epoch slot ⊗ pparams ⊗ txvote ⊗ txwdrls ⟧ ⊢ certState ⇀⦇ txcerts ,CERT*⦈ certState'
  • ⟦ txid ⊗ epoch slot ⊗ pparams ⊗ enactState ⟧ ⊢ govSt ⇀⦇ txgov txb ,GOV*⦈ govSt'
    ────────────────────────────────────────
    Γ ⊢ s ⇀⦇ tx ,LEDGER⦈ ⟦ utxoSt' ⊗ govSt' ⊗ certState' ⟧
```

LEDGER: The transaction type

```
Ix TxId : Type
TxIn  = TxId × Ix
TxOut = Addr × Value × Maybe DataHash
UTxO  = TxIn ⇀ TxOut
```

```
record TxBody : Type where
  txins  : ℙ TxIn
  txouts : Ix ⇀ TxOut
  txfee  : Coin
  txvote : List GovVote
  txprop : List GovProposal
  txsize : ℕ
  txid   : TxId
```

```
record TxWitnesses : Type where
  vkSigs  : VKey ⇀ Sig
  scripts : ℙ Script
```

```
record Tx : Type where
  body : TxBody
  wits : TxWitnesses
```

UTXOW: witnesses & scripts

```
UTXOW-inductive :
  • witsVKeyNeeded ppolicy utxo txb ⊆ witsKeyHashes
  • scriptsNeeded ppolicy utxo txb ≡ᵉ witsScriptHashes
  • ∀[ (vk , σ) ∈ vkSigs ] isSigned vk (txidBytes txid) σ
  • ∀[ s ∈ scriptsP1 ] validP1Script witsKeyHashes txvldt s
  • Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'
    ────────────────────────────────
    Γ ⊢ s ⇀⦇ tx ,UTXOW⦈ s'
```

UTXO: the “core” transition

```
record UTxOEnv : Type where
  slot    : Slot
  pparams : PParams
Deposits = DepositPurpose ⇀ Coin
```

```
record UTxOState : Type where
  utxo           : UTxO
  deposits       : Deposits
  fees donations : Coin
```

```
UTXO-inductive :
  • txins ≢ ∅                                  • txins ⊆ dom utxo
  • minfee pp tx ≤ txfee                       • txsize ≤ maxTxSize pp
  • consumed pp s txb ≡ produced pp s txb      • coin mint ≡ 0
    ────────────────────────────────────────────
                         ⟦ (utxo ∣ txins ᶜ) ∪ˡ outs txb
                         , updateDeposits pp txb deposits
    Γ ⊢ s ⇀⦇ tx ,UTXO⦈  , fees + txfee
                         , donations + txdonation ⟧
```

UTXO: the property of Value Preservation

Property (Value preservation)

- $tx\ .body\ .txid \notin map\ proj_1\ (dom\ (s\ .utxo))$
- $\Gamma \vdash s \rightharpoonup ⦇ tx\ , UTXO ⦈\ s'$

$$\frac{}{getCoin\ s \equiv getCoin\ s'}$$

Compiling to executable Haskell

Proving transitions are computational

```
record Computational (_⊢_⇀⦇_,X⦈_ : C → S → Sig → S → Type) : Type where
  field compute         : C → S → Sig → Maybe S
        compute-correct : compute Γ s b ≡ just s' ⇔ Γ ⊢ s ⇀⦇ b ,X⦈ s'
```

Example: compiling the UTXOW transition (Agda source)

```
Computational-UTXOW : Computational _⊢_⇀⦇_,UTXOW⦈_
Computational-UTXOW = let H , ⁇ H? = UTXOW-inductive-premises {tx}{s}{Γ} where
    computeProof : Maybe $ ∃ (Γ ⊢ s ⇀⦇ tx ,UTXOW⦈_)
    computeProof = case H? of λ where
      (yes (p₁ , p₂ , p₃ , p₄ , p₅)) →
        map₂′ (UTXOW-inductive⋯ p₁ p₂ p₃ p₄ p₅) <$> computeProof' Γ s tx
      (no _) → nothing

    completeness : ∀ s' → Γ ⊢ s ⇀⦇ tx ,UTXOW⦈ s' → M.map proj₁ computeProof ≡ just s'
    completeness s' (UTXOW-inductive⋯ p₁ p₂ p₃ p₄ p₅ h)
      rewrite dec-yes H? (p₁ , p₂ , p₃ , p₄ , p₅) .proj₂
      with computeProof' Γ s tx | completeness' _ _ _ _ h
    ... | just _ | refl = refl
```

```
utxowStep : UTxOEnv → UTxOState → Tx → Maybe UTxOState
utxowStep = compute Computational-UTXOW
{-# COMPILE GHC utxowStep #-}
```

Example: running the UTXOW transition (Haskell target)

```haskell
import Lib (utxowStep)

utxowSteps :: UTxOEnv -> UTxOState -> [Tx] -> Maybe UTxOState
utxowSteps = foldM . utxowStep
```

Example: running the UTXOW transition (Haskell target)

```haskell
spec :: Spec
spec = describe "utxowSteps" $ it "results in the expected state" $
  utxowSteps initEnv initState [testTx1, testTx2] @?= Just (MkUTxOState
    { utxo = [ (1,0) .-> (a0, (890, Nothing))
             , (2,0) .-> (a2, (10,  Nothing))
             , (2,1) .-> (a1, (80,  Nothing)) ]
    , fees = 20 })
  where
  testTx1, testTx2 :: Tx

  initEnv :: UTxOEnv
  initEnv = MkUTxOEnv {slot = 0, pparams = ...}

  initUTxO :: UTxO
  initUTxO = [ (0, 0) .-> (a0, (1000, Nothing)) ]

  initState :: UTxOState
  initState = MkUTxOState {utxo = initUTxO, fees = 0}
```

Conclusion

Future Work

Compilation issues:

- Automate away boilerplate
- Finalize conformance-testing integration
- Randomly test (proven) Agda statements by translating to Quickcheck properties
- Optimizations in implementation → refinements in formalization

Future Work

Compilation issues:

- Automate away boilerplate
- Finalize conformance-testing integration
- Randomly test (proven) Agda statements by translating to Quickcheck properties
- Optimizations in implementation → refinements in formalization

Expand the scope of the formalization

- Prove more interesting meta-theoretical properties
- Cover previous eras: “keeping up with the past”
- Towards verifying smart contracts

Questions?

`https://intersectmbo.github.io/formal-ledger-specifications/`