Lessons learned from building a large, real-world Agda project

Andre Knispel

IOG

November 25, 2024

About the project

- A full specification of the Cardano ledger
  - It is a pure state machine, describing validity of blocks
- FMBC 2024 paper [2]

Formal Specification of the Cardano Blockchain Ledger, Mechanized in Agda

Andre Knispel
Input Output, Berlin, Germany

Orestis Melkonian
Input Output, Kirkwall, UK

James Chapman
Input Output, Glasgow, UK

Alasdair Hill
Input Output, Bristol, UK

Joosep Jääger
Input Output, Tartu, Estonia

William DeMeo
Input Output, Boulder, US

Ulf Norell
QuviQ, Göteborg, Sweden

Abstract

Blockchain systems comprise critical software that handle substantial monetary funds, rendering them excellent candidates for *formal verification*. One of their core components is the underlying

- Started in Jul 2021
- 474 commits

External constraints

- Precedent (existing LaTeX specifications)
- Computability (executable specification, conformance testing)
- Readability (want a document for internal and external consumption)

Relational specification

- The specification consists of lots of not so small state machines.
- To reason about state machines: (inductive) Relations $>$ Functions.
  - Covers partiality automatically.
  - We don't have non-determinism.

Executing relational state machines

- How do we make it executable? `Computational` type class!
  - Old version:

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

  - New Version

```
record Computational (_⊢_⇀⦇_⦈_ : C → S → Sig → S → Type) Err : Type₁ where
  constructor MkComputational
  field
    computeProof : (c : C) (s : S) (sig : Sig) → ComputationResult Err (∃[ s' ] c ⊢ s ⇀⦇ sig ⦈ s')

  compute : C → S → Sig → ComputationResult Err S
  compute c s sig = map proj₁ $ computeProof c s sig

  field
    completeness : (c : C) (s : S) (sig : Sig) (s' : S)
      → STS c s sig s' → compute c s sig ≡ success s'
```

- We have a recipe for writing instances, if the constructors of the relation have the right properties it is straightforward.

Executing relational state machines

```
data _⊢_⇀⦇_,UTXOS⦈_ : UTxOEnv → UTxOState → Tx → UTxOState → Type where
  Scripts-Yes :
    ∀ {Γ} {s} {tx}
    → let open Tx tx renaming (body to txb); open TxBody txb
          open UTxOEnv Γ renaming (pparams to pp)
          open UTxOState s
          sLst = collectPhaseTwoScriptInputs pp tx utxo
      in
        • evalScripts tx sLst ≡ isValid
        • isValid ≡ true
          ────────────────────────────────
          Γ ⊢ s ⇀⦇ tx ,UTXOS⦈  ⟦ (utxo ∣ txins ᶜ) ∪ˡ (outs txb)
                              , fees + txfee
                              , deposits
                              , donations + txdonation
                              ⟧ᵘ

  Scripts-No :
    ∀ {Γ} {s} {tx}
    → let open Tx tx renaming (body to txb); open TxBody txb
          open UTxOEnv Γ renaming (pparams to pp)
          open UTxOState s
          sLst = collectPhaseTwoScriptInputs pp tx utxo
      in
        • evalScripts tx sLst ≡ isValid
        • isValid ≡ false
          ────────────────────────────────
          Γ ⊢ s ⇀⦇ tx ,UTXOS⦈  ⟦ utxo ∣ collateral ᶜ
                              , fees + cbalance (utxo ∣ collateral)
                              , deposits
                              , donations
                              ⟧ᵘ
```

Executing relational state machines

```
instance
  Computational-UTXOS : Computational _⊢_⇀⦇_,UTXOS⦈_ String
  Computational-UTXOS = record {go} where
    module go Γ s tx
      (let H-Yes , ⁇ H-Yes? = Scripts-Yes-premises {Γ} {s} {tx})
      (let H-No  , ⁇ H-No?  = Scripts-No-premises {Γ} {s} {tx}) where
      open Tx tx renaming (body to txb); open TxBody txb
      open UTxOEnv Γ renaming (pparams to pp)
      open UTxOState s
      sLst = collectPhaseTwoScriptInputs pp tx utxo

      computeProof =
        case H-Yes? ,′ H-No? of λ where
          (yes p , no _ ) → success (_ , (Scripts-Yes p))
          (no _  , yes p) → success (_ , (Scripts-No p))
          (_     , _    ) → failure "isValid check failed"

      completeness : ∀ s' → Γ ⊢ s ⇀⦇ tx ,UTXOS⦈ s' → map proj₁ computeProof ≡ success s'
      completeness _ (Scripts-Yes p) with H-No? | H-Yes?
      ... | yes (_ , refl) | _     = case proj₂ p of λ ()
      ... | no _           | yes _ = refl
      ... | no _           | no ¬p = case ¬p p of λ ()
      completeness _ (Scripts-No p) with H-Yes? | H-No?
      ... | yes (_ , refl) | _     = case proj₂ p of λ ()
      ... | no _           | yes _ = refl
      ... | no _           | no ¬p = case ¬p p of λ ()
```

Executing relational state machines

- Lesson: It's a lot easier to prove equivalences between functions and relations than you might think. So if you have a function that is a bit too annoying in your proofs:
  - Make a relation that relates the inputs and outputs of the function.
  - Prove that the two are equivalent.
  - Then prove things about the relation, not the function.

Code organization

- We want to abstract irrelevant details (mostly types and instances) as much as possible.
- Also want `--safe`, so module parameters are our abstraction tool of choice.
- We use bundled module parameters. This makes imports easier and probably has better performance (?).

Code organization

```
record GovStructure : Type₁ where
  field TxId DocHash : Type
        ⦃ DecEq-TxId ⦄ : DecEq TxId

  field crypto : _
  open Crypto crypto public

  field epochStructure : _
  open EpochStructure epochStructure public

  field scriptStructure : ScriptStructure crypto epochStructure
  open ScriptStructure scriptStructure public

  open Ledger.PParams crypto epochStructure scriptStructure public

  field govParams : GovParams
  open GovParams govParams public

  field globalConstants : _
  open GlobalConstants globalConstants public

  open import Ledger.Address Network KeyHash ScriptHash public
```

Set theory

- Gave a dedicated talk about this at AIM XXXI.
- Used to use a list-based approach [1], but this was slow & hard to work with.
- Long story short:
  - Want to use finite sets as a data structure without being bothered by implementation details.
  - The library is split into an abstract axiomatic part and some models.
  - To use it, assume a generic model with the right properties and instantiate it later.[1]

[1] Technically, we actually do fix a model ahead of time, but we keep it all in an abstract block. This could probably be refactored away into the module parameters.

FFI

- Use a mix of `Coercible` from the standard library and a custom `Convertible` class (which is really the same as `Equivalence`).

```
data Coercible (A : Set a) (B : Set b) : Set where
  TrustMe : Coercible A B

{-# FOREIGN GHC data AgdaCoercible l1 l2 a b = TrustMe #-}
{-# COMPILE GHC Coercible = data AgdaCoercible (TrustMe) #-}

-- Once we get our hands on a proof that `Coercible A B` we postulate
-- that it is safe to convert an `A` into a `B`. This is done under the
-- hood by using `unsafeCoerce`.

postulate coerce : {{_ : Coercible A B}} → A → B

{-# FOREIGN GHC import Unsafe.Coerce #-}
{-# COMPILE GHC coerce = \ _ _ _ _ _ -> unsafeCoerce #-}
```

- The aforementioned module parameters are instantiated by concrete types and functions.

FFI

- This approach comes at some runtime cost, but FFI concerns are kept completely separate from everything else.
  - This means we can (and sometimes do) use unsafe code in FFI.
- We recently added a method for asking downstream (Haskell) code for implementations without adding a dependency.
  - This allows the Haskell implementation to supply things such as cryptographic primitives without complicating our build.

FFI

```
module ExternalStructures (externalFunctions : ExternalFunctions) where
  open ExternalFunctions externalFunctions

  HSPKKScheme : PKKScheme
  HSPKKScheme = record
    { Implementation
    ; isSigned         = λ a b m → extIsSigned a b m ≡ true
    ; isSigned-correct = error "isSigned-correct evaluated"
    ; Dec-isSigned     = λ {vk} {ser} {sig} →
      ⁇ (extIsSigned vk ser sig because error "Dec-isSigned evaluated")
    }
```

- Since almost all of our types are defined in modules that are parametrized over these things, we sometimes need to `unsafeCoerce` between types of the form `X ext` and `X dummy`.

Metaprogramming

- Difficult, but lets us reduce boilerplate in various situations:
  - Deriving `DecEq` (agda-stdlib-meta).
  - Deriving MAlonzo FFI datatype bindings (Ulf).

    ```
    instance
      HsTy-UTxOEnv = autoHsType UTxOEnv ⊣ withConstructor "MkUTxOEnv"
                                        • fieldPrefix "ue"
      Conv-UTxOEnv = autoConvert UTxOEnv
    ```

  - Automatically extracting some pieces of step relations, reduces code duplication (Orestis).
  - Basic rewriting for set theory.
- We used to be able to derive `Computational`, but it was too slow, brittle, and certain features we needed were too difficult to implement. So it's abandoned for now, but maybe we can revive it some day.

Performance

- Generally, type checking is only slow in modules containing proofs.
- The biggest source of slowness seems to be meta programs.
  - Even really easy ones are way too slow sometimes.
- Run-time performance has only been an issue when the code was hideously unoptimized.

References

Set theory talk:
`https://www.youtube.com/watch?v=MNouehcOICA`

- Firsov, D., Uustalu, T.: Dependently typed programming with finite sets. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming. pp. 33–44 (2015)

- Knispel, A., Melkonian, O., Chapman, J., Hill, A., Jääger, J., DeMeo, W., Norell, U.: Formal specification of the cardano blockchain ledger, mechanized in agda. In: 5th International Workshop on Formal Methods for Blockchains (FMBC 2024). Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2024)