Human and machine-readable models of state machines for the Cardano ledger

Andre Knispel
IOG
James Chapman
IOG
Orestis Melkonian
IOG & University of Edinburgh, UK
Polina Vinogradova
IOG

Abstract

Cardano is a third generation crypto currency developed by IOG whose nodes consist of a network layer, a consensus layer, and a ledger layer. The ledger tracks and validates financial transactions.

The ledger team at IOG has been successful in using a combination of an abstract specification of the ledger, modeled as a small-step operational semantics and written in LaTeX, pen-and-paper proofs, and property based testing using QuickCheck to support the implementation of this critical component of the system. The specification serves as a design document and reference for the team, and also other members of the Cardano ecosystem. However, LaTeX provides no scope or type checking of the model, and there is a tendency for the spec to get out of sync with the rapidly changing implementation. To mitigate both of these problems, and improve on what we already have, we are developing a specification in Agda which is both human and machine readable. This will provide higher assurance and easier maintenance than the current specification via scope and type checking of the current specification. Additionally, we derive a reference implementation from this model via meta-programming, which can be used for conformance testing against the implementation. Last but not least, we can perform machine checked proofs of key properties.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases blockchain, UTXO, ledger, Agda, meta-programming, formal verification

Funding This work was supported by Input Output (iohk.io) through their funding of the Edinburgh Blockchain Technology Lab.

1 Introduction

The Cardano ledger is a large state machine, specified by (at the time of writing) four documents totaling over 200 pages describing its semantics [2]. One document for the initial version of the Shelley ledger, and one for each set of changes that were introduced in various hard forks. These specifications are implemented as pure Haskell functions, in an 'executable specification': a Haskell implementation of the formal specification(s) that focuses on readability and comparability with its formal counterpart. The original intent was to also produce a separate implementation that focuses on performance, sacrificing comparability to the formal specification, and testing the implementation against the executable specification.

This goal has not yet been reached. The executable specification was sufficiently practical to be used in production, but further practical improvements caused a gap to develop between the formal specification and what has become the production implementation. Additionally, maintaining the same semantics in the formal specification and the executable specification/implementation has been challenging in practice, and there have been several instances in which changes to one were missing from the other for extended periods of time.

To remedy these issues we are currently working on a formal model of the Cardano ledger in literate Agda[3] that can generate both the formal specification and the executable specification from a single source (using Agda's LaTeX and MAlonzo or agda2hs backends), as proposed in [1]. This eliminates any possibility for differing semantics between the formal and executable specifications, closing the gap completely. Furthermore, we can conformance

test the implementation against the executable specification using property based testing, ensuring that the specification and implementation remain in sync. The model should be as close to the original formal specifications as possible in terms of the generated LaTeX document, while of course matching their semantics exactly. The formal specifications are written using small-step operational semantics and set theory, which in our experience works well in practice. This does lead to some friction with Agda's type theoretic foundations. For example, the relations in the formal specifications are not computable a-priori. We are using Agda's reflection mechanism to derive computable functions from these relations, together with proofs of their correctness.

2 General framework & reflection mechanisms

The semantics of the ledger and its parts are given by 4-ary relations of the form

$$\mathrm{STS} \subseteq \mathrm{C} \times \mathrm{S} \times \mathrm{Sig} \times \mathrm{S}$$

where $\mathrm{C}, \mathrm{S}, \mathrm{Sig}$ are the sets of contexts, states, and signals respectively. As an example, the context could hold some fee-related parameters, the state could hold the set of unspent transaction outputs or accounts, and the signal could be blocks or transactions. These relations are usually composed with other relations (in various ways) to ultimately form the CHAIN relation, which models the semantics of block application to the ledger state.

To generate an executable specification from these relations, we require a computable function that produces a new state (or an error) given a context, signal and initial state, such that the function maps its inputs to a non-error output state if and only if the given relation holds on these four values. In Agda, we express this using the following record:

```
record Computational (STS : C → S → Sig → S → Set) : Set where
  field compute : C → S → Sig → Maybe S
        correct : compute c s sig ≡ just s ⇔ STS c s sig s
```

Essentially, a member of Computational STS is a pair of such a compute function, together with a proof of its correctness. Given that such a relation STS is Computational, we prove three key properties:

- STS is right-unique, i.e. given its first three arguments, there is at most one fourth argument making the relation hold.
- Any other correct implementation is (extensionally) equal to compute.
- If equality for states is decidable, the entire relation STS is decidable.

The second property is particularly important for the executable specification: it means that for the semantics of an executable specification, it does not matter how compute, or Computational STS were defined, only that there is a definition for it.

As an example, we could define the following relation[1]:

```
data _⊢_⇀⦇_,UTXO⦈_ : ℕ → UTxO × ℕ → Tx → UTxO × ℕ → Set where
  UTXO-inductive : let f = txfee tx in
    · txins tx ≠ ∅ · txins tx ⊆ dom utxo
    · minFee ≤ f  · balance (txins tx ◁ utxo) ≡ balance (outs tx) + f
    ──────────────────────────────────────────────────────────────
    minFee ⊢ (utxo , fees) ⇀⦇ tx ,UTXO⦈ ((txins tx ⋪ utxo) ∪ outs tx , fees + f)
```

[1] The horizontal line, as well as the dots ·, simply denote function arrows.

This is a simple model of a UTxO ledger with a fixed minimum fee per transaction that is paid into a fee pot. If all the properties above the line hold, the relation below the line is defined to hold. This suggests a general method of implementing compute: match the inputs to the function to the corresponding patterns given below the line. Then, check whether all the properties above the line hold, and if so return the new state as given below the line, otherwise return an error.

This assumes that there is only a single possible derivation for the relation, and that all variables appearing in the derivation already appear in the first three arguments of the conclusion. The latter is always satisfied in our formal specifications, but not the former. If there are multiple possible derivations, one can simply try all of them in some order until one of them succeeds, or fail otherwise. However, the correctness proof of this function then needs to show that the order in which the branches are tried does not matter.

This approach has been mechanized, and to derive a proof that the previous example is Computational one can simply write the following line of code:

```
unquoteDecl Computational-UTXO = deriveComp (quote _⊢_⇀⦇_,UTXO⦈_) Computational-UTXO
```

Compiling this to Haskell yields an executable model for conformance testing.

3 Problems & future work

This is still early work. Currently, our model only contains a rule for UTxO accounting similar to the above example, and a simple rule for witnessing, which only form small parts of the existing ledger. It also contains a proof that the total value in the system is conserved, and some automation that generates an executable specification.

There is some friction resulting from our use of set theory. We need a substancial amount of constructions and facts about finite sets, but expressivity is also an issue. Not all constructions we use in functions in the formal specifications are automatically computable, and Agda's syntax mechanism is not powerful enough to express some of the set comprehensions we use in the formal specification. This forces us to either find notations that are reasonably close to the original ones that do work in Agda, or to introduce differences in how things look in the code and LaTeX, which then has the potential to introduce semantic differences.

Another issue that is out of reach in Agda is vertical vectors. In the example UTXO transition above horizontal vectors are still fine, but there are many examples that are more complicated and difficult to read without vertical notation. This means we will need to use pre-processing or some other method to render the vectors vertically in the LaTeX output.

References

1 Philipp Kant, Kevin Hammond, Duncan Coutts, James Chapman, Nicholas Clarke, Jared Corduan, Neil Davies, Javier Díaz, Matthias Güdemann, Wolfgang Jeltsch, Marcin Szamotulski, and Polina Vinogradova. Flexible formality practical experience with agile formal methods. In *Trends in Functional Programming - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers*, volume 12222 of *Lecture Notes in Computer Science*, pages 94–120. Springer, 2020. doi:10.1007/978-3-030-57761-2_5.

2 IOHK Formal Methods Team. Cardano ledger. URL: https://github.com/input-output-hk/cardano-ledger.

3 IOHK Formal Methods Team. Formal ledger specifications. URL: https://github.com/input-output-hk/formal-ledger-specifications.