Formal Investigation of the Extended UTxO Model (Extended Abstract)

Orestis Melkonian
Information and Computing Sciences
Utrecht University
melkon.or@gmail.com

Wouter Swierstra
Information and Computing Sciences
Utrecht University
w.s.swierstra@uu.nl

Manuel M.T. Chakravarty
Input Output HK
manuel.chakravarty@iohk.io

1 Introduction

Blockchain technology has seen a plethora of applications during the past few years [3, 5, 7], but has also unveiled a new source of vulnerabilities[1] arising from the distributed execution of *smart contracts* (programs that run on the blockchain). Since many of these applications deal with transactions of significant funds, it is crucial that we can formally reason about their (concurrent) behaviour.

This opens up a lot of opportunities for novel applications of formal verification techniques. In particular, we believe that a language-based, type-driven approach to contract development constitutes an effective way to make their execution more predictable. To this end, we attempt to lay the foundations for a mechanized formal framework, where one can verify that certain undesirable scenarios are impossible.

We formulate an accounting model for ledgers based on *unspent transaction outputs* (UTxO), the ledger model underlying Bitcoin [9] and many other blockchains. We conduct our study in Agda [10], exploiting its expressive dependent type system to mechanically enforce desired properties statically. An executable specification of our formal development is available on Github[2].

2 Formal Model

Our formalization closely follows the abstract accounting model for UTxO-based cryptocurrencies presented in [13], which leaves out details of other technical components of the blockchain such as cryptographic operations. We further extend the original formulation to cover the extensions employed by the Cardano blockchain platform [1]. Cardano extends Bitcoin's UTxO model with data scripts on transaction outputs, in an effort to bring it on par with Ethereum's expressive account-based scripting model [6], as well as support for multiple cryptocurrencies on the same ledger [2].

Transactions & Ledgers. For simplicity, we model monetary quantities and hashes as natural numbers. We treat the type of addresses as an abstract module parameter equipped with an injective hash function. Transactions consist of a list of outputs, transferring a monetary value to an address, and a list of inputs referring to previous outputs:

```
module UTxO (Address : Set) (_♯ : Address → ℕ) where
  record OutputRef : Set where
    field id    : ℕ -- hash of the transaction
          index : ℕ -- index in the list of outputs
  record Input : Set where
    field outRef    : OutputRef
          R D       : Set
          redeemer  : State → R
          validator : State → R → D → Bool
  record Output : Set where
    field value   : Value
          address : Address
          D       : Set
          data    : State → D
  record Tx : Set where
    field inps  : List Input
          outs  : List Output
          forge : Value
          fee   : Value
```

Both inputs and outputs carry authorization scripts; for a transaction to consume an unspent output, the result of the validator script has to evaluate to *true*, given the current state of the ledger and additional information provided by the redeemer and data scripts[3]:

```
authorize :: Input → List Tx → Bool
authorize i l = let s = getState l in
  validator i s (redeemer i s) (data (lookup l (outRef i)) s)
```

A ledger consists of a list of transactions, whose *unspent transaction outputs* we can recursively compute:

```
utxo : List Tx → List OutputRef
utxo []       = ∅
utxo (tx :: l) = (utxo l \ map outRef (ins tx)) ∪ outs tx
```

[1] https://en.wikipedia.org/wiki/The_DAO_(organization)
[2] https://github.com/omelkonian/formal-utxo/

[3] Note that redeemers and data scripts can have an arbitrary result type (R and D, respectively).

TyDe'19, August 18, 2019, Berlin, Germany
2019.

Validity. There are still invariants of a well-formed ledger that are not captured by the current typing as $List\ Tx$. To remedy this, we encode the validity of a transaction with respect to a given ledger as a dependent data type. For the sake of brevity, we present only two such conditions, namely that inputs refer to existing unspent outputs and all authorizations succeed:

```
record IsValidTx (tx : Tx) (l : List Tx) : Set where
  field validOutputRefs :
          ∀i → i ∈ ins tx → outRef i ∈ utxo l
        allInputsValidate :
          ∀i → i ∈ ins tx → authorize i l ≡ true
        ...
```

Other validity conditions include that no output is spent twice ($Unique\ (map\ outRef\ (ins\ tx))$) and transactions preserve total values ($forge + \sum_{in} \equiv fee + \sum_{out}$).

It is now possible to characterize a well-formed $Ledger$, by requiring a validity proof along with each insertion to the list of transactions. Exposing only this type-safe interface to the user will ensure one can only construct valid ledgers.

3 Meta-theory

Apart from being able to define correct-by-construction ledgers, we can prove further meta-theoretical results over our existing formulation.

Weakening. Given a suitable injection on addresses, we prove a weakening lemma, stating that a valid ledger parametrized over some addresses will remain valid even if more addresses become available:

$$weakening : (f : A \hookrightarrow B) \rightarrow Ledger\ l \rightarrow Ledger\ (weaken\ f\ l)$$

Weakening consists of traversing the ledger's outputs and transporting all addresses via the supplied injection; in order to keep references intact, the injection has to also preserve the original hashes[4].

Combining. Ideally, one would wish for a modular reasoning process, where it is possible to examine subsets of unrelated transactions in a compositional manner.

We provide a ledger combinator that interleaves two *separate* ledgers. Due to lack of space, we eschew from giving the formal definition of the separation connective $\_*\_\approx\_$. Briefly, two ledgers are separate if they do not share any common transaction and the produced interleaving does not break previous validator scripts (since they will now execute on a different ledger state). These conditions are necessary to transfer the validity of the two sub-ledgers to a proof of validity of the merged ledger:

$$\_\leftrightarrow\_\dashv\_ : Ledger\ l \rightarrow Ledger\ l' \rightarrow l * l' \approx l'' \rightarrow Ledger\ l''$$

[4] A practical case of such weakening is migrating from a 32-bit word address space to a 64-bit one.

The notion of weakening we previously defined proves rather useful here, as it allows merging two ledgers acting on different addresses.

4 Discussion

Proof Automation. Although we have made it possible to express desired ledger properties in the type system, users still need to manually discharge tedious proof obligations. In order to make the proof process more ergonomic, we can prove that the involved propositions are *decidable*, thus defining a decision procedure for closed formulas that do not contain any free variables [12]; we have already proven decidability of the validity conditions[5] and wish to also cover the propositions appearing in weakening and combining.

Comparison with Ethereum. It would be interesting to conduct a more formal comparison between UTxO-based and account-based ledgers, relying on previous work on *chimeric ledgers* [14] that gives a translation between these two approaches. Note that implementing this translation on our inherently-typed representation would guarantee that we only produce **valid** UTxO ledgers.

Towards verification of smart contracts. Although our framework gives a formal foundation for UTxO-based ledgers, reasoning about the high-level behaviour of smart contracts is still out of reach. The quest for a mathematical model that captures the subtleties of contract behaviour and is amendable to mechanized verification is still an open problem, but there seems to be a consensus that formal methods lead to the most promising direction [8].

Scilla, an intermediate-level language for smart contracts, has a formal semantics based on communicating automata that has proven adequate to mechanically verify *safety* and *liveness* properties [11].

The Bitcoin Modelling Language (BitML), an idealistic process calculus for Bitcoin contracts, is accompanied by a small-step reduction semantics and a symbolic model of participant strategies that is intuitive to work with [4]. The authors also provide a compiler from high-level BitML contracts to low-level Bitcoin transactions, along with a compilation correctness theorem: *computational attacks on compiled contracts are also observable in the symbolic model.* We are, in fact, currently formalizing the BitML calculus and its symbolic model in Agda[6] and plan to mechanize compilation down to our formal UTxO model instead.

Acknowledgments

We would like to thank Philip Wadler and Michael Peyton Jones for helpful discussion and IOHK for financial support.

[5] There is an example construction of a valid ledger in the code repository, where our decision procedure automatically discharges all required proofs.
[6] https://github.com/omelkonian/formal-bitml

References

[1] 2019. The Extended UTxO Model. Retrieved 5/2019 from https://github.com/input-output-hk/plutus/blob/master/docs/extended-utxo/README.md

[2] 2019. Multi-Currency. Retrieved 5/2019 from https://github.com/input-output-hk/plutus/blob/master/docs/multi-currency/multi-currency.md

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 443–458.

[4] Massimo Bartoletti and Roberto Zunino. 2018. BitML: a calculus for Bitcoin smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 83–100.

[5] Iddo Bentov and Ranjit Kumaresan. 2014. How to use bitcoin to design fair protocols. In *International Cryptology Conference*. Springer, 421–439.

[6] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).

[7] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM (JACM)* 38, 3 (1991), 690–728.

[8] Andrew Miller, Zhicheng Cai, and Somesh Jha. 2018. Smart contracts and opportunities for formal methods. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 280–299.

[9] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[10] Ulf Norell. 2008. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*. Springer, 230–266.

[11] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR* abs/1801.00687 (2018). arXiv:1801.00687 http://arxiv.org/abs/1801.00687

[12] Paul Van Der Walt and Wouter Swierstra. 2012. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*. Springer, 157–173.

[13] Joachim Zahnentferner. 2018. An Abstract Model of UTxO-based Cryptocurrencies with Scripts. *IACR Cryptology ePrint Archive* 2018 (2018), 469.

[14] Joachim Zahnentferner. 2018. Chimeric Ledgers: Translating and Unifying UTXO-based and Account-based Cryptocurrencies. *IACR Cryptology ePrint Archive* 2018 (2018), 262.