Formalizing BitML Calculus in Agda

ORESTIS MELKONIAN, Utrecht University, The Netherlands

Email: melkon.or@gmail.com, **Advisor**: Wouter Swierstra, **ACM No**: 4094241 **Category**: Graduate (MSc)

1 INTRODUCTION

Blockchain technology has opened a whole array of interesting new applications, such as secure multi-party computation[2], fair protocol design fair [6] and zero-knowledge proof systems [8]. Nonetheless, bugs in *smart contracts* – programs that run on the blockchain – have led to significant financial losses[1], thus it is crucial we can automatically detect them. Moreover, we must detect them statically, since contracts become immutable once deployed. This is exceptionally hard though, due to the concurrent execution inherent in smart contracts, which is why most efforts so far have been on *static analysis* techniques for particular classes of bugs [5, 9, 10].

Recently there has been increased demand for more rigid formal methods in this domain [11] and we believe the field would greatly benefit from a language-based, type-driven approach [15] alongside a mechanized meta-theory. One such example is Scilla, an intermediate language for smart contracts, with a formal semantics based on communicating automata [14]. Scilla follows an *extrinsic* approach to software verification; contracts are written in a simply-typed DSL embedded in Coq [3], and dependent types are used to verify their safety and temporal properties.

In contrast, our work explores a new point in the design space, exploiting the dependent type system of Agda [12] to encode well-formed contracts, whose behaviour is more predictable and easier to reason about. To this end, we formalize an idealistic process calculus for Bitcoin smart contracts, *the Bitcoin Modelling Language* (BitML) [4]. We give an intrinsically-typed model of BitML contracts and a small-step semantics of their execution, as well as a game-theoretic symbolic model that enables reasoning over participant strategies. We have not yet formalized the compiler from BitML contracts to Bitcoin transactions presented in the original paper, but we hope our work paves the way to a fully *certified* compiler.

2 THE BITML CALCULUS

For the sake of brevity, we only give an overview of the major design decisions we made, mostly focusing on the kind signatures of the basic types along with a representative subset of its constructors. The complete formalization is publicly available on Github[2].

Basic Types. First, we parametrize our module with the *abstract data type* of participants, equipped with decidable equality and a non-empty set of *honest* participants *Hon*. Monetary values are represented by natural numbers and a *Deposit* is a *Value* owned by a *Participant*.

Contracts. The type of a contract is indexed by the total monetary value it carries and a set of deposits that guarantee it will not get stuck: **data** *Contract* : *Value* → *List Value* → *Set*. Its constructors comprise the available commands: _⊕_ declares possible branches with equal indices, *split* divides the available funds to multiple contracts (whose values must sum to the initial value), _:_ requires an authorization by a participant to proceed and *after* _ : _ allows further execution of the contract only after some time has passed. Lastly, *withdraw* transfers all remaining funds to a given participant and *put* injects new deposits and secrets to the inner contract:

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

Author's address: Orestis Melkonian, Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands.

```
put _ reveal _ ⇒ _ : (vs : Values) → Secrets → Contract (v + Σ vs) vs′ → Contract v (vs′ ++ vs)
```

A contract is initially made public through an *Advertisement*, denoted ⟨ G ⟩C, which includes a contract C along with some preconditions G that have to be met before it is stipulated.

Small-step Semantics. Our reduction semantics consists of transitions between *configurations*, which are indexed by assets (List A , List A), whose first and second element represent produced and required quantities respectively[3]:

```
data Configuration′ : Asset ∃Advertisement → Asset ∃Contract → Asset Deposit → Set
```

A configuration can hold advertisements ‘_, deposits ⟨ _ , _ ⟩ᵈ, contracts ⟨ _ , _ ⟩ᶜ, secrets _ : _ ♯ _ and action authorizations _ [_]. All asset management occurs when composing configuration with the _ | _ operator; assets required by the right operand can be provided by the left operand. Note that advertisements and contracts are *affine*, but deposits are handled *linearly* (i.e. used only once).

We can now formally define the small-step semantics as a binary relation on *closed* configurations that do not require any assets, i.e. empty in the second position of the tuple. Instead of presenting the entirety of the rules, we choose a representative subset instead:

```
data _⟶_ : Configuration ads cs ds → Configuration ads′ cs′ ds′ → Set where
  D-AuthJoin :                                    D-Join :
      ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | Γ                    ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | A [0↔1] | Γ
    ⟶ ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | A [0↔1] | Γ        ⟶ ⟨ A , v + v′ ⟩ᵈ | Γ
  C-Advertise    : Any (_∈ Hon) (participants (G ad)) → (Γ ⟶ ad | Γ)
  C-AuthCommit   : (secrets A (G ad) ≡ a₀ ... aₙ) × (A ∈ Hon → All (_♯ nothing) aᵢ)
                   → ‘ad | Γ ⟶ ‘ad | Γ | ... ⟨ A : aᵢ ♯ Nᵢ ⟩ ... | A [♯▷ ad]
```

Most rules come in pairs; one rule introduces an authorization of a participant to perform an action and a subsequent rule performs the action. For instance, a participant can join two of her deposits by first authorizing the join action (*D-AuthJoin*) and then actually merging the two deposits (*D-Join*). Other rules are a bit more involved, requiring that certain premises are met before a transition can take place. *C-Advertise* will advertise a contract with at least one honest participant to the current configuration and *C-AuthCommit* authorizes a participant's commitment to all secrets mentioned in the advertisement's precondition, making sure that honest participants only commit to valid secrets.

In all of the rules above, configuration elements of interest always appear on the left of a composition, relying on the fact that (Configuration , _|_) forms a *commutative monoid*. In a machine-checked setting this is not enough; we have to somehow reorder the input and output configurations. We first define an *equivalence* _ ≈ _, relating configurations that are equal up to permutation. We then factor out the equivalence relation in the reflexive transitive closure of the step relation, which will eventually constitute our equational reasoning device:

```
data _⟶*_ : Configuration ads cs ds → Configuration ads′ cs′ ds′ → Set where
  _ ⟶ ⟨ _ ⟩_ : (L : Configuration ads cs ds) { _ : L ≈ L′ × M ≈ M′ }
              → (L′ ⟶ M′) → (M ⟶* N) → (L ⟶* N)
```

Example. Let us give a mechanized derivation for a contract implementing the *timed-commitment protocol* [7], where a participant commits to revealing a valid secret *a* to another participant, but loses her deposit of ₿ 1 if she does not meet a certain deadline *t*:

[3] We prepend an ∃ to the name of a type to denote that we existentially pack its indices.

```
tc-deriv : ⟨ A , 1 ⟩ᵈ ⟶* ⟨ A , 1 ⟩ᵈ | A : a ♯ 6
tc-deriv = let tc = ⟨ A ! 1 ∧ A ♯ a ⟩ reveal [ a ] ⇒ withdraw A ⊕ after t : withdraw B in
  ⟨ A , 1 ⟩ᵈ                    ⟶⟨ C-Advertise ⟩
  ‘tc | ⟨ A , 1 ⟩ᵈ               ...
  ⟨ withdraw A , 1 ⟩ᶜ | A : a ♯ 6  ⟶⟨ C-Withdraw ⟩
  ⟨ A , 1 ⟩ᵈ | A : a ♯ 6        □
```

First, A holds a deposit of ₿ 1, as required by the advertised contract's precondition (*C-Advertise*). The contract is stipulated after the prerequisites are satisfied and the first branch is picked when A reveals her secret. Finally, A retrieves the deposit back (*C-Withdraw*).

Symbolic model. Moving on to the definition of BitML's symbolic model, we associate a label to each inference rule and extend the step relation to emit labels, thus defining a *labelled transition system*. A multi-step derivation $\_ \longrightarrow^* \_$ now accumulates a list of labels and essentially models possible traces of the execution. We can now define *participant strategies* as functions that, given a current trace[4], select a number of possible next moves that are admissible by the semantics. Since only a certain class of strategies is considered valid (e.g. the participant cannot authorize actions by others), we model strategies as dependent record types:

```
record HonestStrategy (A : Participant) where
  field strategy : Trace → Label
        valid    : (A ∈ Hon) × (∀ R α → α ∈ strategy (R ∗) → authorizers α ⊆ [A]) × ...
record AdversaryStrategy (Adv : Participant) where
  field strategy : Trace → (∀ A → A ∈ Hon → HonestStrategy A) → Label
        valid    : (Adv ∉ Hon) × ...
```

The final choice out of all moves submitted by the honest participants is made by a single adversary, whose strategy additionally takes the honest strategies as input and the chosen action is subject to another set of conditions (e.g. the adversary cannot delay time for an arbitrary amount of time). We can now formulate when a trace *conforms* to a set of strategies, namely when each step in the derivation is a (valid) adversarial choice over the available honest moves. Lastly, we prove several meta-theoretical lemmas, e.g. that derivations in the small-step semantics are preserved even if we strip out sensitive information: $\forall\, R'\ \to (R \xrightarrow{\alpha} R') \to (R * \xrightarrow{\alpha} R'\ *)$.

Towards certified compilation. In contrast to the compiler proposed in the original BitML paper, we aim to give a compiler to a more abstract accounting model for ledgers based on *unspent output transactions* (UTxO) [16] and mechanize a similar proof for *compilation correctness*, stating that attacks in the compiled contracts can always be observed in the symbolic model. We already have an Agda formalization for such ledgers[5], which statically enforces the validity of their transactions (e.g. all referenced addresses exist).

Our formalization actually covers extensions to the basic UTxO model of Bitcoin, as employed by the Cardano blockchain [1]. Since these extensions allow for more expressive power in the scripts residing in transactions[6], we expect the translation to be more straightforward, much like how the financial DSL *Marlowe* is implemented on top of an extended-UTxO ledger [13]. Most importantly, compilation down to our dependently-typed ledgers will guarantee that we only ever get valid ledgers.

[4] Before we give an execution trace as input to a strategy, we traverse the derivation and strip out all secrets using $\_*$.
[5] https://github.com/omelkonian/formal-utxo
[6] For instance, the addition of *data scripts* in transaction outputs makes stateful behaviour possible.

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] 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.

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. *The Coq proof assistant reference manual: Version 6.1*. Ph.D. Dissertation. Inria.

[4] Massimo Bartoletti and Roberto Zunino. 2018. *BitML: a calculus for Bitcoin smart contracts*. Technical Report. Cryptology ePrint Archive, Report 2018/122.

[5] Massimo Bartoletti and Roberto Zunino. 2018. Verifying liquidity of Bitcoin contracts. *IACR eprint* (2018).

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

[7] Dan Boneh and Moni Naor. 2000. Timed commitments. In *Annual International Cryptology Conference*. Springer, 236–254.

[8] 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.

[9] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 116.

[10] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 48.

[11] 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.

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

[13] Pablo Lamela Seijas and Simon Thompson. 2018. Marlowe: Financial contracts on blockchain. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 356–375.

[14] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687* (2018).

[15] Tim Sheard, Aaron Stump, and Stephanie Weirich. 2010. Language-based verification will change the world. (2010).

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