Blockchain Semantics in Agda

Laying the foundations for the formal verification of smart contracts

ORESTIS MELKONIAN, School of Informatics, University of Edinburgh, UK

This report presents the progress made during the second year of my PhD studies, supervised by Prof. Philip Wadler and co-supervised by Dr. Brian Campbell and Prof. Aggelos Kiayias.

1 PROGRESS: 2nd YEAR

This section documents what I have accomplished during the second year of my PhD studies. For the sake of brevity, all formalisations are referenced through blue hypelinks inside the text, and submitted papers that resulted from my research appear in green boxes. For reference, I have attached the first-year report and all my paper submissions as separate PDF files.

1.1 UTxO formalisation

Following my work on EUTxO and in particular the bisimulation proof between EUTxO ledgers and *constraint emitting machines (CEMs)* [Chakravarty et al. 2020a], I continued collaboration with the Plutus team at IOHK to extend the model with support for multiple currencies that we achieved through *native custom tokens*, a system we called EUTxO$_{\text{ma}}$.

This is made possible by the generalisation of transferred values from simple scalars to two-level maps that record a balance for each currency (the two leves are needed to support *non-fungible* tokens, as well as a new category of *monetary policy* scripts that control how various tokens can be created and destroyed.

Apart from the financial applications that multi-currency enables, we managed to remedy a significant shortcoming of the previous bisimulation proof, namely that we could neither distinguish between different executions of the same state machine, nor ascertain that a current given state genuinely originates from a valid sequence of steps starting at an initial state. This meant that it is not sound to transfer inductive properties proven valid on the CEM level down to the transaction level, as we could only establish that steps are preserved but there were no initiality guarantees whatsoever.

However, by minting a new currency solely for the purpose of regulating CEM traces and their origin, along with a carefully designed *CEM monetary policy* that enforces all invariants we care about, we were able to solve the aforementioned meta-theoretical problem and provide a more robust bisimulation result. I have formalised all these claims here.

Author's address: Orestis Melkonian, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh, UK.

This resulted in a paper publication at the Reliable Smart Contracts workshop (RSC), co-located with the ISoLA symposium.

> **Native Custom Tokens in the Extended UTXO Model** [3]
> *M.Chakravarty, J.Chapman, K.MacKenzie, O.Melkonian, J.Müller, M.P.Jones P.Vinogradova, P.Wadler*
> `[eutxoma.pdf]`

At the same time, we added the same kind of support in the plain UTxO model, by expressing monetary policies in a simplistic domain-specific language, resulting in a system we dubbed UTxO_{ma}. This makes the approach more accessible to the wider blockchain community, that is easily implementable in most standard settings without too many extensions, but still lacking the nice meta-theoretical properties of EUTxO .

I was again responsible for mechanising UTxO_{ma} on top of my previous UTxO formalisation and another paper was published at the same workshop:

> **UTxO_{ma}: UTXO with Multi-asset Support** [4]
> *M.Chakravarty, J.Chapman, K.MacKenzie, O.Melkonian, J.Müller, M.P.Jones P.Vinogradova, P.Wadler, J.Zahnentferner*
> `[utxoma.pdf]`

1.2 BitML formalisation

Moving on to the BitML part of my thesis, things were far less fortunate; while I already had a prototype definition of coherence by the end of last year, the complete definition seemed to be much more complicated than it first seemed and I still do not have a 100% postulate-free definition.

Progress has been slow due to technicalities mainly arising from the complexity and sheer size of the definition of *coherence*. The definition itself consists of 20 dense inductive cases: the corresponding Agda code encompasses a frightening 2000 lines of code!

Furthermore, there is also the need for systematic naming conventions to keep things manageable. I achieved these using primarily the following features of Agda:

- *typeclasses* for name overloading, *i.e.,* it quickly gets tedious to have different names for similar functions on different types, so we collect all those types in a typeclass and use a single general name that is subsequently resolved to the specific instance implementation automatically via *instance argument resolution*;
- *generalised variables* for all types that we frequently use, *i.e.,* we just declare that certain identifiers/symbols will represent variables of a given type by default and then freely use these without any type annotation;
- *parameterised modules* for enforcing naming conventions systematically, *i.e.,* we can derive definitions systematically from some given parameters, while modules allow us to always give the same name to these derived facts.

Alas, these are relatively new features and their interaction is not that well tested and does not seem to scale very well to the levels that my project requires, leading to the discovery of various bugs in the Agda implementation. Thankfully, I have so far managed to find (dirty) workarounds and proceed with the project, but the process is far from ideal. On the bright side, these woes have resulted in many bug reports and almost all of them have been fixed for future Agda releases.

Apart from the technical side, there is also a conceptual issue that hindered my overall progress: the explanation text in the original paper leaves too many things implicit, which makes it necessary for me to devise several invariants or assumptions myself. Mistakes I might do then or corner cases I might have missed potentially requires refactoring the whole codebase from scratch. As I started accumulating copious amount of such implicit information, I contacted the BitML authors to make sure this is what they informally had in mind but did not spell out in the paper: they agreed on all my points, phew!

One example of this issue occurs in the indexed maps that the coherence relation keeps track of at each step of its inductive definition; these maps concern resources we have available for each participant/variable/etc. There is a lot of handwaving in the BitML paper on how these maps formally get updated, but the Agda formalisation has to be absolutely precise about that. This leads to additional proof obligations for each case, where we need to reason about the domains/co-domains of these maps, which is far from trivial and nowhere mentioned in the text. At least we get the benefit of additional sanity-checking on our rules, since these additional constraints have to always been proven for each case and hence safeguard against semantic errors.

Another subltle ambiguity concerned the notion of the *subterms* of a contract, namely that in some cases these refers to sub-structures in the contract that correspond to Bitcoin transactions, while in other cases it referred to parts of the script we have to sign. The problem is that these two do not coinside, therefore I had to go back and disambiguate all such usages in my BitML work so far.

While all the aforementioned issues consumed a good part of this year of my PhD, without producing any significant research breakthrough or something worthy of publication, the situation now is much clearer with very few things left to attend to, so I am quite optimistic I will have a completely mechanised definition of coherence real soon.

1.3 Agda Infrastucture

1.3.1 ***AGDA2HS.*** As an advanced user of Agda for a few years, I started attending the Agda Implementors' Meeting (AIM), where people get together for a week to present Agda-related work they have recently done or conduct code sprints (usually working on Agda itself).

I was planning to co-organise AIM XXXI with Wen Koeke, but COVID-19 happened and we had to cancel our plans. After that, I attended the next meeting (AIM XXXII) as a bystander, and during AIM XXXIII I chose the lonesome road, as I found the projects quite intimidating for me to join along, and instead developed the `setup-agda` tool for Github (see Section Sect. 1.3.2).

During AIM XXXIV, I joined a code sprint with Ulf Norell, Jesper Cockx and James Chapman, where we worked on a new backend for Agda, AGDA2HS, that would compile a non-dependently typed Agda program to an almost identical Haskell program, erasing proofs and fancy type-level features along the way. While the general design decisions stemmed from group discussions during the meeting, my personal contributions to the actual development of the backend were the following:

- **continuous integration** for the Haskell development environment, making sure everything still works after each commit,
- supporting more of Haskell's **builtin types** (`Word/Float/Char`),
- translating Agda's if_then_else_ to Haskell's builtin `if..then..else..`,
- automatically generating the necessary Haskell **imports** when certain types/operations are used on the Agda side,
- compiling Agda's where clauses to Haskell's `where` clauses, which was suprisingly tricky as Agda defines a separate module for each where block and one needs to chase these definitions down to insert them at this point,
- supporting **default** methods in typeclasses with **minimal complete definitions** (*i.e.,* when an instance of a class can be given just by giving a subset of the required methods, and the rest of the fields get derived from them), which implements a particularly complex translation procedure (described in the paper) which is necessary since Agda has no support whatsoever for these features.
- and, last but not least, **reviewing** contributions from the other members of the team.

Later this year, Jesper led a group of undergraduate students at TU Delft that conducted an experiment on AGDA2HS by porting several data structures from the popular `containers` Haskell package to Agda and proving important properties about them such as the laws appearing in the web documentaion of each data structure.

This led to a paper submission to the prestigious workshop on Certified Programs and Proofs (CPP), co-located this year with POPL, with the hope of disseminating this novel approach for program verification. Acceptance is still pending; notifications are on November 22nd.

> **Reasonable Agda is Correct Haskell: Instrinsic Program Verification using AGDA2HS** [5]
> *Jesper Cockx, Orestis Melkonian, James Chapman, Ulf Norell, et al.*
> [`agda2hs.pdf`]

I have confidentialy attached our submission (`agda2hs.pdf`); please do not share with anyone else. As for which parts I have personally contributed, I was mostly responsible for writing the part of the paper describing the actual backend, and had little involvement with the sections that describe the use cases.

While this project seems totally irrelevant to the central topic of my thesis, this is not entirely true; as pointed by Prof.Kiayias in my first-year PhD review, my work would benefit tremendously from a practical result in addition to the (mechanised)

meta-theoretical proofs. AGDA2HS indeed provides such an avenue, since all the dependently typed programs that are included in my formalisations might now potentially be extracted to readable Haskell code that interfaces with an existing blockchain like Cardano.

1.3.2 ***Continuous Integration:*** `setup-agda`. As the size and complexity of my BitML formalisation grew, encompassing several different packages, dozens of modules, and thousands lines of code, I saw the need for continuous integration alongside the version control of my repositories on Github. Alas, no one had yet bothered to bundle an Agda-specific plugin as available in more mainstream languages like Javascript and Haskell.

Therefore, I had to write my own Github Action plugin that would automatically compose the sequence of shell commands one needs to run, given just a few declarative indications of which versions to install, which extra libraries to download, whether to publish an HTML version of the code, and where the entrypoint of the project is.

The final result was that now one does not need to copy customized shell commands each time they create an Agda project that they wish to keep under continuous integration. The user just invokes the plugin from the Github CI configuration file and `setup-agda` will take care of the rest. To clarify with an example, this is what the CI configuration file for the BitML compilation proofs looks like:

```yaml
name: CI
on: push: {branches: master}
jobs:
  build-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2.3.1
      - uses: omelkonian/setup-agda@v0.1
        with:
          agda-version: 2.6.1.3
          stdlib-version: 1.6
          libraries: |
            omelkonian/formal-prelude#92ef
            omelkonian/formal-bitcoin#0341
            omelkonian/formal-bitml#4382
          main: Main
          token: ${{ secrets.GITHUB_TOKEN }}
```

This project depends on my personal Agda prelude (formal-prelude), my formalisation of the BitML calculus (formal-bitml) and my formalisation of Bitcoin transactions (formal-bitcoin), where these are similarly configured and published using `setup-agda`.

This has made it possible to effortlessly evolve code on separate repositories, whilst making sure everything I commit typechecks (*i.e.,* all proofs are valid).

As for the implementation, since Agda is itself written in Haskell, I had to first package the existing `setup-haskell` tool as a Typescript library, which has been merged in the official repository, and I also added *caching* support, but this extension is still pending approval. Then I developed a custom Typescript plugin for Github Actions that runs the Agda-specific commands.

1.3.3 ***Reflection-based automatic solvers: Prelude.Solvers.*** Another minor sidetask I have started investigating is an extensible framework for defining automatic solvers in Agda. This was primarily motivated by the increased complexity of my BitML formalisation, where I found myself writing proofs that followed an annoyingly repeating proof pattern.

I based my design around Agda's *reflection* feature and ease of extensibility: solvers should be provided modularly in separate files that do not depend on one another. So far, I have implemented prototype solvers just for *list membership* and *propositional equality*, but I hope to continue developing others as needed without too much overhead. The plan is to identify lemmas about a given type of propositions that is repeatedly used in a certain mechanical pattern, and use them in the corresponding solver(s) of the types involved as we go.

A *solver* defines a view function that extracts information from the hole's type but may fail for types that are not handled by the current solver, and a solve function that uses this information to solve the hole by also having access to a recursive call to the general solving procedure (*i.e.,* that can call other solvers as well).

Once we have several of these solvers, we tie the recursive knot by importing all of them and then defining the *general solver* as the global disjunction, *i.e.,* we try out all sub-solvers until one succeeds and fail only when all of them fail.

Finally, the user can simply invoke the solve macro or solveWith ⋯ to further provide extra facts that are not available in the current local context, instead of writing an explicit proof term. Simple toy cases are now instantly discharged, but my current solvers need significant extensions to deal with the complex proofs I work with in the BitML project.

1.4 Towards a Separation Logic for UTxO-based blockchains

As an aside project, in collaboration with Wouter Swierstra (Utrecht University) and James Chapman (IOHK), we started thinking about whether a reasonable Hoare-style logic could be defined for UTxO-based ledgers. The main motivation behind this is the realisation that accounts with balances resemble memory locations with values assigned to them; advances in the theory of (concurrent) memory access may be transferred to the theory of blockchain. In particular, *Hoare Logic (HL)* is useful for reasoning about imperative programs reading/writing to memory, which can then be extended to *Separation Logic (SL)* to allow for modular reasoning of such programs that operate on disjoint parts of the memory, and finally work on *Concurrent*

Separation Logic (CSL) provides reasoning tools for concurrency. Therefore it is sensible to ask:

> *Can we transfer the concepts of HL/SL/CSL to the UTxO setting?*

Wouter initially drafted a document exploring the idea in more formal mathematical notation. Starting from the simple definition of a ledger as a list of transfers of a specific monetary value between two participants, we can describe execution in terms of ledger states that map participants to their current balance, both in *operational* as well as *denotational* style. Then, a simple Hoare logic can be constructed, essentially providing an *axiomatic* semantics than can be proven sound with respect to the previous semantics. If we, furthermore, augment the logical formulas appearing in Hoare triples with *separating conjunction* $P * Q$, we can formulate the *FRAME* inference rule of SL and hopefully prove it!

Then, I went from there to formalising these concepts in Agda, which proved to be suprisingly tricky due to the multitude of options one can choose to model them in a type-theoretic setting. The formalisation is readily available here.

First, a *Ledger* is defined as a transfer list between an abstract type of participants, and both *denotational* and *operational* semantics are defined where ledger states are maps from participants to their current balance, *e.g.,* the denotation of a transaction is a function from a state to the next (updated) state, and we prove that these semantics are *equivalent*.

Here comes the first major design decision we have to make; how should we represent such maps? One example of **shallow embedding** would be maps as functions from keys to values, while a **deep embedding** could explicitly keep a list of key-value pairs. Another choice between shallow and deep embedding is also manifest for the logical formulas that will appear in Hoare triples, where the dilemma takes the form of whether to follow a *higher-order abstract syntax (HOAS)* approach or not. There, it turns out, the deep approach is dictated by the need to be able to enumerate all references variables, since with HOAS we would lose the ability to reflect on lambdas.

Crucially, it does not matter which approach we use as this is an implementation detail. What we need is only a precise interface of the data structure, along with formal properties that it should satisfy, and then the library author can pick any implementation that conforms to these. Most importantly, we need to avoid *leaky abstractions* at all costs; it would be unfortunate if implementation details were allowed to appear in the meta-theoretical proofs for Hoare-style ledgers, since any change in the implementation could potentially render our proofs invalid! To remedy this issue, I devised a **systematic way to define abstract data structures in Agda**, where one can pick-and-choose different implementations depending on the proof ergonomics of each specific case. To cut a long story short, one defines the interface as a record containing operations/laws/etc and implementations are given in separate files as modules marked with abstract to avoid leaks to the outside world. Finally, we use Agda's *records-as-modules* syntax to open the interface record R using an

implementation that we imported as a module M (*i.e.,* open R (record { M }) public) to make the whole interface accessible but its implementation hidden to the user.

For my needs thus far, I have defined an interface for *Sets*, and one for *Maps*. I have implemented sets for some type A *as lists* of type A paired with a proof that no duplicates exist and *as predicates* of type $A \longrightarrow$ Set, that is much easier to work with but does not enjoy decidability. The choices for maps include implementations *as partial functions* from keys to values or total ones.

After this short digression, let us move on to present a Hoare-style *axiomatic semantics*, based on a *deep embedding of propositional formulas* appearing in pre-/post-conditions, since we will need to somehow refer to the set of participants that are involved when we later cover features from separation logic.

I have proven that this semantics is **sound** with respect to the previously defined *denotational semantics* and, by transitivity, to the *operational* one, as well as *commutativity* and *associativity* of separating conjunction which naturally follows from the aforementioned *map laws*.

If we also define what it means for a ledger to be disjoint , we can formulate and prove the *[FRAME]* inference rule of SL:

$$\frac{l \# R \quad \{P\} l \{Q\}}{\{P * R\} l \{Q * R\}} \text{ [FRAME]}$$

where P, Q, R are propositional formulas, l is a ledger, $\#$ denotes disjointness of the underlying participant sets, and the Hoare triple $\{P\}l\{Q\}$ indicates that if we start from a state that satisfies P and execute all transactions in l we arrive at a new state satisfying Q. The rule effectively allows us to inject facts about a separate ledger (R) in our local conditions.

But what about the concurrent case, where transactions can be executed in parallel? It turns out that the *[PARALLEL]* inference rule of CSL can be effortlessly transferred over from concurrency theory, bearing the gift of *modular reasoning*:

$$\frac{l_1 \parallel l_2 = l \quad \{P_1\} l_1 \{Q_1\} \quad \{P_2\} l_2 \{Q_2\} \quad l_1 \# P_2 \quad l_2 \# P_1}{\{P_1 * P_2\} l \{Q_1 * Q_2\}} \text{ [PARALLEL]}$$

where we prove some facts about disjoint parts of the ledger l_1, l_2 and then combine them to produce a proof about any interleaving l. Finally, we demonstrate the benefits of said modularity through an *example* of four transactions that can be split into two disjoint pairs of two, which essentially mirrors what has already been witnessed in the case of concurrent imperative programming. Success!

In retrospect, the main motivation for pursuing this idea might have been to have a pleasant break from the technically heavy work on BitML; it is certainly the case that Agda is more satisfying to work with in small-scale projects/prototypes rather than medium-/large-scale ones. However, many problems that arise in this setting are strikingly similar with the ones that appeared on my investigation of EUTxO ; in particular, reasoning about temporal properties of a contract in the high level of CEMs (*c.f.,* Sect. 1.1) and transferring these proofs to the low level of EUTxO transactions. Thus, I believe this line of work nicely fits under the general theme of my PhD topic,

and there might be useful insights that we can gain for the more central questions of my thesis. If time allows until the end of my PhD studies, that is whether I finish the BitML formalisation promptly, I wish to continue exploring the obvious next step in this research direction: extending the formalisation from simple linear transaction ledgers to actual UTxO-based blockchains and writing a paper on that.

1.5 Academic Experience

Teaching. I have provided teaching support in various courses that the University offers, as a marker, tutor, and/or teaching assistant:

- **TSPL:** Types and Semantics for Programming Languages
- **INF1A:** Introduction to Computation: Functional Programming
- **INF2D:** Reasoning and Agents

Paper reviews. I was also invited to contribute my first ever reviews for two blockchain-related submissions to the following venues:

- **APLAS:** Asian Symposium on Programming Languages and Systems
- **MSCS:** Mathematical Structures in Computer Science (special edition in memory of Martin Hofmann)

2 TIMEPLAN: 3^{rd} YEAR

Let me finally outline my plans for the final year of my PhD studies, aimed at producing a thesis disseration at the end of the year.

Compared to my initial plan, the completion of BitML has been delayed from the end of the second year to (hopefully) the middle of the third year. This essentially means that a few ambitious research directions I initially had in mind and was planning to execute will most probably not happen, rather compromising on just having a complete formalisation of BitML.

2.1 OCT'21 - MAR'22 | Finish BitML

I plan to spend half of the last year of my PhD in completing my mechanisation of the BitML complilation correctness proof. This should be doable as there are only a couple of pages in the original paper that I have not yet covered, namely the game-theoretic translation from high-level (BitML) strategies to low-level (Bitcoin) strategies and then the final proof of *computational soundness* assuring us that we can safely reason on the simpler BitML model without missing possible attacks on the Bitcoin level.

As for a contigency plan in case of unexpected turbulence, I will resort to making some additional simplifying assumptions to at least produce a complete end-to-end compilation correctness proof.

Last, I am confident that I have gathered enough material throughout my experimentation with BitML to produce a substantial publication at the end of this year.

2.2 MAR'22 - AUG'22 | Thesis write-up

The second half of the third year will be devoted to writing up my thesis, as I have accrued enough diverse material on a common theme that can easily fill up a dissertation.

However, given that it would be impossible for me to write text full-time on a daily basis, I will use a small portion of my time to develop the other half-baked projects I have started and tying up loose ends that may still dangle from the first half of the year.

REFERENCES

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*. 83–100.

Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. 2020a. The Extended UTXO Model. In *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers (Lecture Notes in Computer Science)*, Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala (Eds.), Vol. 12063. Springer, 525–539. https://doi.org/10.1007/978-3-030-54455-3_37

Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. 2020b. Native Custom Tokens in the Extended UTXO Model. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 12478. Springer, 89–111. https://doi.org/10.1007/978-3-030-61467-6_7

Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. 2020c. UTXO_{ma}: UTXO with Multi-asset Support. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 12478. Springer, 112–130. https://doi.org/10.1007/978-3-030-61467-6_8

Jesper Cockx, Orestis Melkonian, James Chapman, and Ulf Norell et al. 2021. Reasonable Agda is Correct Haskell: Instrinsic Program Verification using AGDA2HS. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, United States, January 16-22, 2022*. ACM. Paper submitted, still under review.

Wen Kokke and Wouter Swierstra. 2015. Auto in agda. In *International Conference on Mathematics of Program Construction*. Springer, 276–301.

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