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 serves as the proposal of my PhD thesis, supervised by Prof. Philip Wadler and co-supervised by Dr. Brian Campbell and Prof. Aggelos Kiayias.

1 RESEARCH TOPIC

1.1 Background

Blockchain technology has opened a whole array of interesting new applications (e.g. secure multi-party computation [Andrychowicz et al. 2014b], fair protocol design [Bentov and Kumaresan 2014], zero-knowledge proof systems [Goldreich et al. 1991]). Nonetheless, reasoning about the behaviour of such systems is an exceptionally hard task, mainly due to their distributed nature. Moreover, the fiscal nature of the majority of these applications requires a much higher degree of rigor compared to conventional IT applications, hence the need for a more formal account of their behaviour.

The advent of smart contracts (programs that run on the blockchain itself) gave rise to another source of vulnerabilities. One primary example of such a vulnerability caused by the use of smart contracts is the DAO attack[1], where a security flaw in an Ethereum smart contract permitted the exploitation of a venture capital fund worth 150 million dollars at the time. The solution was to create a hard fork of the Ethereum blockchain, *i.e.,* reverse all transactions from the time of the attack, clearly going against the decentralized spirit of cryptocurrencies. Since these (possibly Turing-complete) programs often deal with transactions of significant funds, it is of utmost importance that one can reason and ideally provide formal proofs about their behaviour in a concurrent/distributed setting.

1.2 General Objectives

The primary research goal I propose is to provide a formal model of distributed ledgers, along with a complete mechanization in a mainstream proof assistant. This will enable formal reasoning over the concepts and techniques relevant to distributed ledger technology, giving further confidence to the results produced by the cryptographic community, as well as fostering collaboration

[1] https://en.wikipedia.org/wiki/The_DAO_(organization)

Author's address: Orestis Melkonian, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Edinburgh, UK, orestis.melkonian@ed.ac.uk.

with the fields of Programming Languages. Moreover, I hope to lay the foundations for a much more robust development process for smart contracts, where blockchain developers specify the functionality they wish to implement in a declarative style close to mathematics and then prove logical propositions about the behavioural properties they deem interesting. Ideally, there would be a straightforward way for the developer to encode their smart contract in a proof assistant, which would be equipped with the necessary infrastructure to prove these properties.

First, the formalisation should concern an abstract description of distributed ledgers, in contrast to a full-fledged implementation, so as to make mathematical proofs tractable and modelling decisions feasible in the timespan of a 3-year PhD.

Second, the formalised ledger should support programmable transactional schemes via smart contracts, since these give rise to the concerns about distributed behaviour that I wish to address.

Last but not least, it is crucial that the proofs accompanying the proposed formalisation are of moderate complexity and there is an ergonomic way to prove common properties that developers care about. Hence, I wish to investigate techniques that will render proving behavioural properties of ledgers and smart contracts ergonomic. This will involve, for instance, implementing decision procedures for propositions that frequently appear in such specifications, as well as examining meta-programming techniques that allow for further proof automation.

1.3 Specific Objectives

BitML Mechanization. I aim to formalize the theoretical results presented in [Bartoletti and Zunino 2018], where an idealised process calculus for Bitcoin smart contracts, called the *Bitcoin Modelling Language* (BitML), is introduced. This includes mechanizing BitML's operational semantics and the compilation correctness proof that accompanies the compiler from BitML contracts to Bitcoin transactions. Therefore, one of the most integral parts of my PhD thesis would be the following:

> *Provide a mechanized model of the BitML compiler and its correctness proof.*

Expressiveness of UTxO. The PhD position is taking place while collaborating with IOHK, a blockchain technology company that has close ties with academia and has adopted a scientific process based on peer-reviewing.

Its blockchain platform, *Cardano*, follows an accounting model similar to Bitcoin [Nakamoto 2008], based on *unspent transaction outputs* (UTxO). However, they employ several extensions to the plain UTxO, dubbed Extended UTxO (EUTxO), which allow for more expressive smart contracts.

Another reason to focus on UTxO-based ledgers, instead of account-based ones such as Ethereum [Buterin et al. 2014], is the conceptual complexity associated with stateful behaviours that are difficult to reason about, in comparison to functional specifications that have a straight-forward mathematical equivalent. Hence, the following question naturally arises:

> *What is the relative expressiveness of EUTxO compared to plain UTxO?*

This can again be stated in terms of secure compilation, namely compiling a well-defined mathematical abstraction into an equivalent formalization of EUTxO transactions. In this case though, the full abstraction result we are after is significantly simpler, *i.e.,* proving that a formulation of EUTxO-based ledgers coincide with a particular variant of state machines.

This research avenue eventually leads to the design of a formal framework, in which it would be possible to systematically compare the expressive power of EUTxO-based ledgers against account-based ones, e.g.:

> *What is the relative expressiveness of EUTxO compared to account-based equivalents?*

BitML targeting EUTxO. While the goals of formalizing BitML and investigating the expressiveness of EUTxO seem somewhat disparate, there is actually a sensible point on which they can be combined, namely:

> *Investigate whether compiling BitML contracts to EUTxO transactions simplifies the translation procedure and makes the proof significantly easier.*

Specifically, the BitML compiler is quite involved and, naturally, the accompanying proof complex, due to the limitations imposed by the target language, *i.e.,* the fact that expressing stateful behaviour in Bitcoin smart contracts raise significant issues. This misalignment between the target's lack of support for stateful computation and the inherent automata-based features of BitML's operational semantics, makes it considerably harder to provide relevant correctness proofs.

If we were to target EUTxO-based smart contracts in the compilation, it is expected that the translation becomes much more straightforward and the proofs become simpler to structure and spell out formally.

1.4 Scope of the Dissertation

It will be clear by now that the proposed research topics revolve around two main pillars, namely the proof mechanization of BitML's compilation correctness and the formal expressiveness of EUTxO. While these share the common goal of proving correspondence between a high-level system and a lower-level one, they still constitute quite heterogeneous goals.

An immediate issue that arises is the possibility that my final PhD dissertation will be incoherent, unable to convey a singular contribution to the state-of-the-art in blockchain metatheory. Therefore, it might be safer to focus on a single topic for the sake of coherence.

At this stage, I am still uncertain about the direction I would like to pursue, but it would certainly be valuable to raise this concern during the 1^{st} year review in June and discuss it in front of the review committee.

2 METHODOLOGY

2.1 Proof Mechanization

We are concerned with a highly complex system that requires rigorous investigation. Therefore, we choose to conduct our formal study in a mechanized manner, i.e. using a proof assistant along the way and formalizing all results in Type Theory. Proof mechanization will allow us to discover edge cases and increase the confidence of the model under investigation.

As our proof development vehicle, we choose Agda [Norell 2008], a dependently-typed total functional language similar to Haskell [Hudak et al. 1992]. Agda embraces the *Curry-Howard correspondence*, which states that types are isomorphic to statements in (intuitionistic) logic and their programs correspond to the proofs of these statements [Martin-Löf and Sambin 1984]. Through its unicode-based *mixfix* notational system, one can easily translate a mathematical theorem into a valid Agda type. In absence of such a liberal syntax, one would need to express syntactic forms in a different way (*e.g.*, translating infix definitions into normal prefix form), leading to a wider gap between the informal mathematical text and the mechanized proof. Moreover, programs and proofs share the same structure, e.g. induction in the proof manifests itself as recursion in the program. While Agda has not been proven to be adequate for large software development yet, its flexible notation and elegant design is suitable for rapid prototyping of new ideas and exploratory purposes. We do not expect to hit such problems, since we will stay on a fairly abstract level which postulates cryptographic operations and other implementation details.

This methodology is also in sync with IOHK's rigorous development pipeline; ideas are initially worked on paper by pure academics, then formally verified in a proof assistant for more confidence, resulting in a prototype/reference implementation in Haskell that informs the production codebase (also in Haskell) on the properties that should be tested.

2.2 Proof Automation

Proofs have to be relatively easy to discharge, so as to make user interaction with our formal framework as ergonomic as possible. For instance, a third party interested in blockchain-specific properties should not spend time on proving simple arithmetic lemmas.

One way of addressing this issue is to provide decision procedures for logical propositions appearing frequently in the specifications. For instance, UTxO validity conditions require some element to be part of a set, so it is convenient to provide a decision procedure for set membership. This style of proof automation is usually referred to as *proof-by-reflection* [Van Der Walt and Swierstra 2012] and works well for *closed* formulas that do not contain any free variables (e.g. when proving a property of an example ledger with known constituents). Note the overloaded use of the word "reflection", since it does not refer to the typical meta-programming feature of modern programming languages, but rather the proof-theoretic technique of mechanically deriving a theorem's proof from its shape [Van Der Walt 2012].

Proof-by-reflection is not of much use when variables are involved (e.g. when proving a general theorem). To address proof automation in the presence of unspecified variables, one has to resort to meta-programming techniques, where one writes *tactics* that manipulate proof contexts.

Although recent Agda versions come with an experimental reflection mechanism that allows for compile-time meta-programming, it is still far less mature than the tactic languages available in Coq and Isabelle/HOL. Given that Agda's proof automation facilities are still in such an early phase, it might be worthwhile to develop a tactic scripting language from scratch, similar to Coq's *Ltac* language. Apart from discharging more proof obligations automatically (i.e. less manual interaction needed from the user/prover), such a scripting language will also be beneficial to the greater Agda community.

3 WORK PLAN

Year 1. By the end of the first year of my PhD, I estimate to have completely mechanized the results presented in the BitML paper [Bartoletti and Zunino 2018] or, at least, have the general skeleton of the whole proof with minor proof holes. The current status of my formal development suggests I am on schedule, since I have already formalized the source and target systems (i.e.

BitML and Bitcoin transactions), as well as the respective game-theoretic models (i.e. the symbolic and computational models) and the actual compilation process. What remains is the formulation of the correspondence relation between the two models (i.e. *coherence*) and proving that the compiler preserves that relation, which is feasible to cover until the end of the 1^{st} year.

Moreover, I am simultaneously working on the orthogonal topic of formalizing the relative expressive power of UTxO and its extended variant, EUTxO, in collaboration with the Plutus team at IOHK. We have already produced significant theoretical results, namely a bisimulation proof between EUTxO transactions and a novel variant of state machines, called *Constraint Emitting Machines* (CEM). By the end of the 1^{st} year, we aim to have extended the bisimulation proof with multicurrency support.

Year 2. During the 2^{nd}, I plan to finalize BitML's mechanized proof and distill general principles on the design choices involved when considering semantics-preserving compilation with security properties. At this point, possibly there will be insights on how to improve the theory of BitML or the associated proof techniques. Therefore, I would devote time to investigate these improvements and again extract as many principles for structuring secure compilation proofs as possible.

Moreover, I plan to continue working on further theoretical results, pertaining to the expressive power of EUTxO and a formal comparison with account-based ledgers. However, I cannot foresee specific future directions, as this is highly dependent of the general direction of the Plutus team.

Year 3. At the beginning of the 3^{rd} year, I plan to tackle any issues that have not yet reached closure until then. Then, I will devote the rest of the year to write up my PhD thesis, based on skeletal parts I will accumulate throughout my PhD studies.

4 RELATED WORK

In this section, I conduct an extensive review of literature relevant to my research topic. Previous work is drawn from a wide range of scientific fields, from *Blockchain* to *Programming Languages*. The purpose of such an attempt is twofold; first, to thoroughly understand what has already been achieved and, second, to draw inspiration for further directions.

4.1 Blockchain Technology

Although a very recent concept, blockchains have gained major popularity and revealed new research direction in a wide range of fields. While initially introduced only informally, there has been copious amounts of research to formally understand its mechanisms and reason about its behaviour since its

conception in 2009 [Bonneau et al. 2015], a story reminiscent of *BitTorrent* in the context of *peer-to-peer file sharing* [Cohen 2003].

The Dolev-Yao Model. Before we investigate blockchain-specific work, it is worthwhile to notice a prominent technique used for proving that certain security protocols are secure, namely the *Dolev-Yao* model [Dolev and Yao 1983].

One of its main characteristics is that primitive cryptographic operations are postulated to exist with ideal properties, hence allowing for emphasis on the interesting properties currently under scrutiny. Enforcing such separation of concerns — by abstracting away from these details — is a technique that appears throughout our formal development. Keeping a certain amount of abstraction is paramount to efficient prototyping, so as to get sufficient results in a reasonable amount of time.

Another highly influential aspect of the Dolev-Yao model is the consideration of active malicious participants, who are able to impersonate other users, alter transmitted messages and perform other similar actions.

Bitcoin. In 2009, a person/group by the alias of *Satoshi Nakamoto* proposed a decentralized system, *Bitcoin*, able to provide a completely decentralized monetary system [Nakamoto 2008]. The system allows users to exchange currency, without the need for any trusted central entity, by providing a novel consensus algorithm that decides the order of transactions and prevents *double-spending*, i.e., makes sure that there is a certain amount of currency at each point in time and it cannot be spent twice.

Instead of keeping track of explicit accounts, Bitcoin's accounting model is based on *unspent transaction outputs* (UTxO). These are locked transaction outputs carrying some monetary value, which can be spent by anyone who is able to unlock it. The simplest form of locking an output would be using a public key, requiring that the spending transaction is signed with the corresponding private key. In reality, the lock is a *validation script* which expects some arguments; the output can be spent by anyone that provides the arguments that leads script execution to succeed. Therefore, one can design and implement more complicated transactional schemes than the simple, key-based one. Bitcoin offers a a low-level, Forth-like, stack-based language (Script), which has rather limited expressiveness.

For a more thorough description of all integral components of the Bitcoin system, we refer the reader to Princeton's Bitcoin book [Narayanan et al. 2015].

Ethereum. The next most popular blockchain currently is *Ethereum*, which proposes a wholly different approach to conceptualizing the blockchain and

advocates for adopting blockchain technology for general distributed applications (*dApps*), rather than restricting usage to tasks of a fiscal nature [Buterin et al. 2014].

The main difference lies in the underlying accounting model which, in contrast to Bitcoin's UTxO-based approach, consists of a global mutable state of accounts along with the total amount of currency they possess. Transactions that are then submitted to the blockchain change this global state accordingly. This is coupled with a feature-heavy, Javascript-like, Turing-complete scripting language, called *Solidity*, which compiles down to bytecode targeting the *Ethereum Virtual Machine* (EVM). EVM programs are then executed as imperative programs that mutate the blockchain's global state.

Consensus & Relation to Concurrency. There is great similarity between the notion of consensus inherent to how a blockchain operates and classical consensus problems in *Concurrency Theory*, where the question is how to reach agreement on a value in a distributed setting.

First, the consensus algorithm used by both Bitcoin and Ethereum is based on *Proof-of-Work*, requiring blockchain participants to solve hard cryptographic puzzles in order to submit blocks/transactions, thus defending against *denial-of-service* (DoS) attacks. There exist alternative consensus mechanisms, used by other blockchain systems, such as *Proof-of-Stake*, but we refrain from investigating this blockchain component further, as it does not pertain to the central focus of this thesis. This is due to the fact that we are concerned with a component orthogonal to the consensus layer, namely the underlying accounting model.

One notable difference between classical consensus on distributed systems and blockchain-specific consensus is the fiscal nature of blockchain technology, leading to game-theoretic investigations of financial incentives, in addition to reasoning about computational behaviour/resources. Nonetheless, blockchains shed a new light on the consensus problem and have spurred enough interest to reshape the landscape of consensus research [Garay and Kiayias 2018].

Apart from the common background in consensus protocols, the problems one faces when reasoning about the security and behaviour of blockchains bear a striking resemblance with those encountered when reasoning about computer programs that run concurrently or even across different machines, i.e., the main research focus of *Concurrency Theory* and *Distributed Computing*. This particularly pertains to the way programs, running simultaneously on a blockchain, interact with one another [Herlihy 2019].

Formal Blockchain Models. While research is still in its infant steps, there have been considerable efforts to provide formal models of its operations, especially from the cryptographic community [Badertscher et al. 2017; Cachin et al. 2017; Garay et al. 2015; Halpern and Pass 2017]. Alas, the landscape of mechanically verified formal models for blockchain systems is still scarce.

An exception is Anton Setzer's formal model of Bitcoin in Agda [Setzer 2018], which provides a mechanically-verified, executable specification of how Bitcoin operates. In order to make such a mechanization effort tractable, cryptographic primitives and other irrelevant components are dealt with abstractly, by postulating such functionality and the corresponding desired properties; a technique we also employ throughout our formal development. While this work is a wonderful exercise in dependently-typed programming and uses advanced modelling techniques provided by Agda, such as *induction-recursion*, no further meta-theory is formalized and the feasibility of such a static model for proving any kind of non-trivial property of the blockchain remains questionable.

Another noteworthy attempt is [Marmsoler 2019], where a tool for specifying the behaviour of *dynamic architectures*, FACTum, is used to model blockchains and automatically generate code for the Isabelle/HOL proof assistant [Nipkow et al. 2002]. Then, it is possible to interactively prove desired properties of the system, such as persistence of blockchain entries, using temporal logic to reason about traces.

Lastly, [Hirai 2018] is an interesting work in this direction, which formally models blockchains as state-transition systems and reasons about possible properties using logical formulas in temporal-epistemic terms, drawing inspiration from the fields of *modal logic* and its interpretation in terms of *Kripke structures*. More specifically, the modality of these logical formulas allow for expressing and analyzing the *atomicity* of a cross-chain swap.

4.2 Smart Contracts

Smart contracts are computer programs that reside on the blockchain itself and, consequently, are executed on-chain. Their name is derived from their fiscal nature (i.e., they resemble legal/financial contracts) and the fact that they are self-enforced without requiring a trusted intermediary. As was briefly mentioned in Subsection 4.1, Bitcoin only provides simple script templates, while Ethereum users have access to a full-blown imperative language.

Attacks & Analysis Tools. Since smart contracts handle monetary transactions, they provide a tempting battleground for malicious attackers to exploit

undiscovered vulnerabilities, sometimes leading to a tremendous loss of capital (*e.g.,* the infamous DAO attack[2]). This is exactly the reason why a lot of research effort has been put on validating contracts, either by analyzing their source code or developing appropriate formal methods for reasoning about the behaviour and security properties.

[Sergey and Hobor 2017] examine different attacks on Ethereum smart contracts and recast them as classical problems found in the settings of concurrent program execution, in order to advocate for a *contracts-as-concurrent-objects* analogy. While it seems surprising that deterministic programs written in Solidity exhibit similar behaviour to concurrent programs with shared memory, this becomes clear when one considers the races in terms of transaction submission (i.e., one cannot decide on the exact order transactions will get incorporated).

A taxonomy of security vulnerabilities of Ethereum smart contracts is systematized in [Atzei et al. 2017], encompassing issues introduced at the levels of Solidity, EVM and the blockchain mechanism itself. These vulnerabilities concern Solidity's function-calling, error-handling and resource-monitoring mechanisms, as well the inherent immutability of EVM bytecode and the limitations of blockchain's inner workings. Finally, a collection of smart contracts are presented, which exhibit issues that fall into one of the investigated categories.

Another categorization of attacks possible on Ethereum is provided in [Luu et al. 2016], although the kind of attacks are more focused on timestamp dependencies, improper handling of exception and a particular form of function callbacks, called *re-entrancy*, which led to the aforementioned DAO attack. Having identified how these attacks are possible through examples, an operational semantics for Ethereum is introduced and suggestions for improvement and tighter security are recommended. Alas, these extensions require all network clients to upgrade; something not realistically feasible. To remedy this limitation, the authors introduce an analysis tool based on symbolic execution, OYENTE, that can automatically detect flaws in individual contracts, helping *developers* to program safer contracts and *users* to avoid interacting with problematic ones. The tool is accompanied by experimental evaluation, which demonstrates that OYENTE is a pragmatic tool for analyzing smart contracts in the wild.

As expected from the fact that certain vulnerabilities have been identified and led to significant financial losses, there is currently a plethora of analysis tools, which check smart contracts against well-known vulnerabilities and "code smells". These can be divided between tools that statically

[2] https://en.wikipedia.org/wiki/The_DAO_(organization)

analyze the program under question, *e.g.,* using the Datalog logic programming language [Grech et al. 2018; Tsankov 2018], and the ones that dynamically monitor contracts for violations and possibly provide recovery mechanisms [Colombo et al. 2018].

Formal Methods & Verification. While analysis tools are a pragmatic approach to detecting known vulnerabilities, they prove inadequate for security issues that are yet to appear. Most importantly, once smart contracts are deployed on the blockchain, they are immutable. Therefore, developing appropriate formal methods to reason about contract behaviour and safeguard against future attacks is of paramount importance, as evidenced by the amount of surveys available in this domain [Bartoletti and Zunino 2019a; Harz and Knottenbelt 2018; Miller et al. 2018]. One approach that proves quite attractive for these purposes is *language-based verification*, where issues are avoided by carefully designing the semantics of the language itself [Sheard et al. 2010].

A notable example of a language specifically designed to accommodate formal verification is Scilla, an intermediate language for Ethereum-like smart contracts [Sergey et al. 2018]. Scilla's semantic model is based on *communicating automata*, which provide a clean separation between the language's purely functional fragment and its asynchronous messaging mechanism. Its development comprises of a *shallow embedding* in Coq [Barras et al. 1997], where one can formally verify properties of contracts. The properties under question are of a temporal nature and are subsequently divided in two categories: *safety* properties that hold throughout a contract's execution and *consistency* predicates that hold under certain assumptions, which mandate examining a contract's *execution traces* and interactions with other contracts. Scilla is used in the Zilliqa blockchain, is accompanied by well-documented operational semantics and integrated static analyses to aid contract developers and has been used to verify crucial properties of several example contracts [Sergey et al. 2019].

Another framework for certifying smart contracts in Coq is *ConCert* [Annenkov et al. 2020; Annenkov and Spitters 2019; Nielsen and Spitters 2019]. Although it specifically targets the Oak programming language for smart contracts, the techniques used are applicable to any *functional* smart-contract language. The problem addressed here is the semantic gap between a *deep* and a *shallow* embedding of a language, which are independently useful in different context; one would use a deep embedding to reason about a language's *meta-theory*, while the shallow embedding is convenient for validating concrete, individual contracts. This is solved by a clever use of recently introduced meta-programming facilities of the Coq ecosystem, which allows for a more principled way to connect the semantics of the two alternative embeddings.

A Coq framework is also being developed in the context of Michelson, a low-level, stack-based scripting language for the Tezos blockchain [Goodman 2014]. Again, the language is designed with formal verification in mind. *Mi-Cho-Coq* is the Coq embedding of Michelson, which provides facilities for formally verifying the functional correctness of Michelson programs [Bernardo et al. 2019]. To make this possible, the original Michelson interpreter written in OCaml was ported/embedded in Coq and then contract properties are proven using Djikstra's famous *weakest precondition calculus* [Dijkstra 1975a]. In contrast to ConCert though, there is no semantic connection between the Coq embedding and the actual Michelson interpreter. In contrast to Scilla, the contract properties one can verify are still quite limited, e.g., do not include an adversarial model.

A different point in the design space of formal frameworks for smart contracts is investigated in [Chen et al. 2018]. It is based on the $\mathbb{K}$ framework, where semantics are defined in a *language-independent* fashion and several language components (e.g. parser, interpreter) are automatically derived in a *correct-by-construction* manner. To that end, a formal semantics of EVM are defined in the $\mathbb{K}$ framework, dubbed *KEVM*, but no reasoning mechanisms/logics have been formulated yet.

Lastly, an alternative approach that circumvents the use of a proof assistant are *SMT solvers*, such as Microsoft's Z3 [De Moura and Bjørner 2008]. These allow for automatically deciding first-order formulas (in a suitable decidable theory), rather than providing a complete proof/derivation interactively. The VeriSol verifier is one such example, used in Microsoft's Azure blockchain [Wang et al. 2018]. There, Azure contracts are annotated with function pre-/post-conditions, which get checked automatically by an external SMT solver. A rather similar approach is employed in the context of Ethereum, where Solidity programs are statically analyzed to derive an SMT-encoding of the same program, whose `assert` statements can then be automatically verified by an SMT solver [Alt and Reitwießner 2018]. However, there is an inherent limitation to the kind of properties that can be automatically verified by an SMT solver, a process commonly referred to as *push-button verification*. For instance, loop invariants in imperative programs are typically inserted manually by humans to aid the SMT solver, although there exist techniques for automatically inferring these in some cases, such as *monomial predicate abstraction* [Lahiri and Qadeer 2009].

4.3 State Machines

As has become quite obvious by now, opinions are converging towards an automata-based interpretation of smart contracts.

Smart Contracts. First and foremost, the operational semantics of the aforementioned smart-contract language Scilla is based on a particular form of communicating automata, called *Communicating State Transition Systems* [Sergey et al. 2018]. This was initially employed for specifying arbitrary resource protocols to reason about concurrent resources, combining the compositionality of *Concurrent Separation Logic* (CSL) and the fine-grained resource control of *Rely-Guarantee* (RG) [Nanevski et al. 2014].

Another automata-based approach to specifying Ethereum smart contracts is employed in the *FSolidM* tool, where the user can design automata in a graphical interface, which can then be automatically translated to Solidity contracts [Mavridou and Laszka 2018]. The authors have also identified common security vulnerabilities and specified corresponding *design patterns*, allowing a user to effortlessly integrate it in their state machine in the form of an *extension plugin*. The easy-to-use graphical interface allows for easier adoption by the community, while the plugin mechanism aids in the development of more secure contracts. For instance, a 'locking' plugin is provided to "foolproof" contracts against reentrancy attacks, by excluding mutually recursive calls for all functions inside the contract.

In the context of Bitcoin, [Andrychowicz et al. 2014a] utilizes the notion of *Timed Automata* to model Bitcoin smart contracts, which are specified in the Uppaal model checker [Larsen et al. 1997]. This provides a pragmatic way to verify temporal properties of concrete smart contracts; Uppaal can verify properties written as *timed computation tree logic* (TCTL) formulas. Alas, the authors provide no formal claim that this class of automata actually corresponds to Bitcoin smart contracts, hence there is still a significant semantic gap that hinders further formalization.

Mealy Machines. One particular form of state machines that might prove useful in the context of understanding smart contract behaviour, are *Mealy machines*, an extension of standard state machines to allow transitions to additionally produce output [Mealy 1955].

This formulation proves attractive, as it is well-established, has been thoroughly studied in the past, and has even been proved isomorphic to a coalgebraic logic, i.e. every finite Mealy machine corresponds to a finite formula of this logic [Bonsangue et al. 2008].

Although initially conceived to specify sequential digital circuits, the added ability of emitting outputs seems appropriate in the context of a blockchain ledger, where transaction submissions (i.e., transitions) might have side-effects on the blockchain (i.e., outputs). Note that this is a major inspiration

for our formulation of *Constraint Emitting Machines*, which we employ to reason about the expressiveness of a UTxO-based ledger with extended scripting capabilities (see Subsection 5.4).

4.4 Process Calculi

It is generally agreed that the λ-calculus provides a *canonical* model for purely functional computation, Unfortunately, there is no established equivalent for concurrent computation, but there has been a rich body of research towards establishing such a core calculus for concurrency [Pierce 1997]. These formal models for concurrency are typically called *process calculi*, while the scientific area that emerged through their study is sometimes referred to as *process algebra* [Baeten 2005]. Here, we give a brief overview of some important attempts in this direction and contrast that with one of our main topic of interest in this thesis, namely the BitML calculus.

CCS. One of the most influential theoretical models towards a core calculus for concurrency is the *Calculus of Communicating Systems* (CCS), which introduces constructs for variable renaming/restriction, parallel composition and non-deterministic choice, among others [Milner 1980].

Its main innovation is the abandoning the idea of programs as pure functions from input to output, which is achieved by introducing an alternative *semantic domain* for processes, based on *labelled transition systems*. In fact, this led to the related development of *structural operational semantics* [Plotkin 1981], a highly influential technique that is employed by the state-of-the-art in the semantics of programming languages up to this day.

π-calculus. Following a series of publication by Milner [Milne and Milner 1979; Milner 1979a,b], whose purpose was to refine the theory and capabilities of CCS, a further extension of CCS was later introduced to add support for *mobility*, i.e., the ability for processes to give explicit names to communication channels and communicate the names themselves across channels, called the π-calculus [Milner 1989]. There, a formal model of π-calculus is introduced, along with the notion of *bisimulation* that identifies equivalent processes when they behave the same in any context (more on this in Subsection 4.5).

CSP. Around the same time as CCS, another process calculus for concurrency appears by the name of *Communicating Sequential Processes* (CSP) [Hoare 1978]. CSP is based on Djikstra's *guarded commands* [Dijkstra 1975b], which are commands prefixed by a boolean predicate; sequential composition can proceed only when the predicate expression evaluates to `true`. Furthermore, CSP employs synchronized communication, which in addition

to guarded commands, proves to be sufficient to express commonly used programming constructs, such as co-routines, procedures and functions, as well as common concurrent primitives, such as monitors and conditional critical regions.

In contrast to previous approaches that use global variables to express communication, CSP introduces a paradigm shift to *message passing*. Surprisingly, the paradigm of message passing introduced in CSP was not apparent in the initial form of CCS, but inspired Milner to incorporate it in its revised versions and the π-calculus [Baeten 2005].

Although not initially accompanied by a proper semantics and lacking rigorous techniques for proving correctness, further investigations led to what came to be known as *Theoretical CSP* [Hoare 1980; Hoare et al. 1981], which is based on *trace theory* and *failure pairs* preserving deadlock behaviour.

ACP. A more axiomatic approach is taken in the *Algebra of Communicating Processes* (ACP), where models for concurrent processes are specified as axiomatic systems, consisting of equational rules [Bergstra and Klop 1987].

There, a more algebraic treatment of processes is investigated, given by a series of axiomatic systems with gradually increasing complexity. First, a simple axiom system, called *Basic Process Algebra*, provides the core of all subsequent refinements and contains no communication constructs whatsoever (although it contains parallel composition, which freely interleaves processes). Then, communication is added to arrive at the definition of the axiomatic system, called ACP, which is then extended with *abstraction* to yield ACP_τ, allowing for more scalability and modular proof techniques.

In contrast to CCS that provides communication tangled with abstraction, and CSP that merges messaging with restriction, ACP employs a more general communication scheme by introducing all these features independently [Baeten 2005].

BitML. A process calculus specifically designed for blockchain smart contracts is the *Bitcoin Modelling Language* (BitML) [Bartoletti and Zunino 2018], which constitutes one of the integral research topics of this thesis.

BitML contains constructs for fundamental smart contract operations, such as withdrawing funds, specifying deadlines and revealing secrets. By combining these core constructs, one gets a highly flexible, yet minimal, calculus, proven to be adequate for expressing a diverse set of contract examples [Bartoletti et al. 2018; Lande and Zunino 2018].

BitML's operational semantics is defined as a *labelled transition system* (LTS) between configurations, which indicate the funds of each participant and action authorizations among others. Then, a *symbolic model* is defined over the

execution traces allowed by the operational semantics, upon which a game-theoretic notion of honest and adversarial strategies is defined. Briefly, an honest strategy outputs a set of possible next moves, given the current execution trace, while the adversary has the final call of which of all possible moves by all honest participant is actually realised.

The authors also provide a compilation scheme from BitML to 'standard' Bitcoin contracts, accompanied by a compilation correctness proof, stating that any attack possible on the Bitcoin level is also reflected in BitML's symbolic level. The aforementioned translation targets a formal model of Bitcoin transactions that precisely defines transactions and their consistency with respect to a given ledger [Atzei et al. 2018]. A similar game-theoretic model is defined on the low level of Bitcoin transactions, dubbed *computational model*. Compilation correctness is then formulated as a correspondence between symbolic and computational runs.

Although BitML does not contain constructors for *explicit message passing*, it can nevertheless be accurately described as a process calculus, due to the non-determinism inherent in the rules of its operational semantics. In fact, when one starts reasoning about actions different participants may employ, it quickly becomes clear that such strategies are defined via inter-participant communication. In other words, it turns out that BitML shares messaging capabilities with more typical process calculi, although they arise implicitly through BitML's game-theoretic symbolic model.

The same authors investigated the property of *liquidity* in BitML smart contracts, i.e., that the funds stored within a smart contract do not remain frozen indefinitely [Bartoletti and Zunino 2019b]. The result is several alternative definitions of liquidity, all proven to be *decidable*, thus allowing *model-checking* to automatically decide these properties for an arbitrary contract. Remarkably, one can specify arbitrary temporal properties in *linear temporal logic*, in addition to liquidity, which can then be automatically checked by the model-checker. This constitutes an addition to the list of static analysis tools presented in Subsection 4.2, arguably increasing confidence during the development of secure contracts [Atzei et al. 2019].

4.5 Programming Language Theory

In this section, we give an overview of results in the theory of programming languages, which are closely related to our main research questions. Although the body of work in this area is vast, we provide a coarse stratification of relevant sub-fields and provide pointers for further investigation.

Semantics of Programming Languages. Since most of the work discussed in this section pertains to questions regarding the semantics of programming languages, it would be helpful to review the prominent styles of specifying such semantics.

Denotational semantics study programming constructs as mathematical objects in some *semantic domain D*, where each type of the language is mapped to its denotation via a function $[\![\_]\!] : Type \to D$ [Scott 1970]. This approach is *compositional*, since the denotation of a large program can be derived from the denotations of its constituents, e.g., the previous definition naturally extends to *contexts* and *typing derivations*:

$$[\![\Gamma \vdash t : \sigma]\!] = [\![\Gamma]\!] \to [\![\sigma]\!]$$

In contrast to the abstract nature of denotational semantics, *operational semantics* try to capture a more concrete description of a programs execution. Here, we are particularly interested in the so-called *small-step* or *structural* operational semantics [Plotkin 1981], where a program's meaning corresponds to a series of individual computational steps, rather than describing the overall result of computation as in *big-step* or *natural* operational semantics. An important advantage of such an approach is that it is closer to the intuitive computational behaviour and allows a *syntax-directed*, *inductive* definition, which is amenable to rigorous verification. While big-step semantics are also structural, it abstracts away evaluation details that we might want to consider, in order to have an accurate understanding of a program's execution. *Inference rules* are given as a set of transitions from initial to target configuration, which are typically modelled by a (labelled) transition system, a reoccurring phenomenon throughout our own work.

Last but not least, a more recent development was the formulation of a language's semantics in game-theoretic terms, known as *game semantics* [Abramsky 1997]. This is of interest to us, since it inspired BitML's semantics and, moreover, provided solutions to long-standing problems on *full abstraction* of several programming languages (more on this later on).

Bisimulation. In [Pierce 1997], two foundational calculi are presented: the λ-calculus for purely functional computations and the π-calculus for concurrent systems. There are many possible definitions of equivalence between different terms/processes.

Denotational equivalence states that two terms are equivalent if their semantic counterpart is equivalent:

$$M = N \iff [\![M]\!] = [\![N]\!]$$

This definition is not always satisfactory, since the semantic domain may contain *too many* values, hence differentiating between terms/processes that behave the same. As an artificial example, consider the case where we denote terms as the sequence of the syntactic elements; then $1 + 2$ and $2 + 1$ would not be denotationally equivalent, although they behave the same.

To remedy this, the notion of *observational equivalence* was introduced, in which two terms are considered equivalent if they behave exactly the same based on the observations we can extract from them. The most common form of this type of equivalence is *contextual equivalence*, where we consider two terms equivalent, only when their impact is the same on any surrounding context:

$$M = N \iff \forall C[M] \downarrow iff C[N] \downarrow$$

While this notion of equivalence captures the semantics more strictly and is closer to general intuition, it may prove difficult to prove due to the universal quantification of arbitrary contexts.

An alternative formulation, based on a language's operational semantics, is *bisimulation*, where two systems are considered equivalent if they execute corresponding steps that start and finish on *bisimilar* states. Thus, proving bisimulation entails coming up with some relation to connect system states, as well as a proof that steps allowed by the operational semantics move between states that are related in this way.

In its original form, *strong bisimulation*, the two systems must run in "lock-step", i.e., there is an one-to-one correspondence between their steps. A more common form of bisimulation is the so-called *weak bisimulation*, where systems are allowed to perform an arbitrary number of internal steps, *i.e.,* a single step in the source might correspond to multiple steps in the target.

The main advantage of bisimulation over contextual equivalence is the absence of the aforementioned universal quantification, therefore lending itself to a pragmatic proof technique, *coinduction* [Sangiorgi 2011], where one can assume bisimilarity and, if that leads to no contradiction, we get the desired proof of bisimulation. In contrast, an inductive proof would make it harder to identify all the possible way such non-deterministic systems behave the same, although their internal structure might be completely different. For instance, this kind of proof technique is used to study process equivalence in the context of the π-calculus [Sangiorgi 1996].

Lastly, a language-agnostic technique for relating different state-based system, reminiscent of bisimulation, is presented in the seminal work of [Abadi and Lamport 1988]. Specifically, the goal is to prove that a high-level specification is correctly implemented by a lower-level one. This can be achieved by providing a *refinement mapping* from high-level states/steps to low-level

ones. The authors continue to show that there exist certain techniques to augment the low-level definition (e.g. by introducing auxiliary variables), so as to guarantee the existence of such a mapping.

Full Abstraction for Denotational Semantics. It is commonplace to define both denotational and operational semantics for a language, given that each has complementary advantages. An issue that arises, however, is whether these two different semantics coincide; denotationally equivalent terms should also be observationally equivalent.

This question was introduced in the context of the *Programming Computable Functions* (PCF) programming language, an extension of the simply-typed λ-calculus, where a semantics is dubbed *fully-abstract* when both equivalences coincide; observational equivalence corresponds to denotational equivalence [Plotkin 1977]. [3]. Around the same time, Milner discussed full abstraction for the more general setting of typed λ-calculi [Milner 1977]. Another interesting use of *full abstraction* concerns the comparison of different evaluation strategies, as exemplified in [Riecke 1993], where call-by-value, call-by-name and lazy evaluations are considered.

There have been alternative definitions that generalize the notion of equivalence, such as *logical full abstraction* [Longley and Plotkin 2000]. Game semantics have proven quite flexible in tackling the problem of full abstraction, as seen by the diverse range of programming constructs approached this way, which include subtyping, probabilistic choice, references, non-determinism and concurrency [Curien 2007].

Other examples of full abstraction results include a fully-abstract *trace semantics* for a λ-calculus extended with *references* [Laird 2007], as well as a fully-abstract semantics for classical processes [Kokke et al. 2019].

Full Abstraction for Expressiveness. It turns out that finding such fully-abstract translations is a notoriously hard process, witnessed by the sheer amount of publications on full abstraction for PCF [Abramsky et al. 2013], sometimes not even being satisfiable [Parrow 2016]. More surprisingly, it is debatable whether full abstraction actually is a good global criterion for the systematic investigation of the relative expressive power of programming languages, as shown by demonstrative counterexamples in [Gorla and Nestmann 2016].

Therefore, a central problem in the theory of programming languages is systematically comparing the expressiveness of different languages. Several

[3] A very similar notion is *computational adequacy*, which states that observationally distinct terms have distinct denotations.

theoretical frameworks have been proposed to tackle this question, although there is still no consensus towards a universally accepted solution.

A framework suited to formally study the expressiveness of different language extensions appears in [Felleisen 1991], where different extensions to the basic λ-calculus are shown and compared with the core language. Some are proven to strictly raise the expressive level of the language, hence rendering a local embedding impossible.

A more generic framework is provided in [Shapiro 1991], where the languages compared need not have a common semantic basis. In this case, several concurrent languages are compared, though the method applies to sequential ones as well.

In [Mitchell 1993], another technique is introduced based on *abstraction-preserving reductions*, where several examples and counterexamples are shown.

Certified Compilation. Proving compilers correct has had a long history in computer science, starting from the 1960s [McCarthy and Painter 1967]. The primary purpose is to give formal guarantees that compiling from a high-level language to a lower-level ones does not alter the source semantics.

A renewed interest in correct compilation arose in recent years by CompCert, a compiler for a large subset of C (called Clight), targeting all common instruction sets [Leroy 2009].[4] Entirely programmed in the Coq proof assistant, the compiler itself is programmed in Coq's purely-functional subset Gallina and its proofs of correctness are expressed with *dependent types* and proven using *tactics* [Barras et al. 1997]. CompCert demonstrated that full compiler verification is *feasible* and gives a lot of benefits, since the compiler backend is highly optimized, consisting of multiple phases, each proven to preserve the desired safety properties via bisimulation and the resulting performance is on par with GCC [Leroy 2006].

An alternative approach for certified compilation, which does not require that the compiler is written in a dependently-typed language, is *type-preserving* compilation. The seminal paper of [Morrisett et al. 1999] describes the design of an optimizing compiler from System F to a typed assembly language TAL, where high-level abstractions (e.g. closures) are enforced on the type level. Therefore, preserving the types through compilation guarantees that no such violations arise in the produced low-level code. Of course, as we require more intricate properties to be preserved through compilation, we will need ever more expressive type systems, which is by itself a fruitful direction for future research in secure compilation.

[4] CompCert initially only compiled down to PowerPC, although it supports x86, ARM and RISC-V currently.

Certifying compilers, on the other hand, opt out of verifying the whole compilation process and, instead, verify the validity of each individual output through a program outside the compiler, called the *certifier*. Note that this technique is sometimes also referred to as *translation validation*. A primary example of this is the Touchstone compiler from a type-safe subset of C to the DEC Alpha assembly language [Necula and Lee 1998]. Although the compiler cannot be said to be *bug-free*, it will nonetheless report any incorrect compiler outputs and is, arguably, much easier to prove and amenable to future compiler extensions (e.g. new optimization phases), since the certifier can be developed independently.

Secure Compilation. If we would also like to preserve security properties through compilation, we arrive back to the notion of full abstraction, since it is the technique most often employed in the context of *secure compilation* [Patrignani et al. 2019].

The seminal work of [Abadi 1999] uses full abstraction in two different compilation issues; first, preserving security properties of Java classes arising from access modifiers (e.g., public/private) down to the generated intermediate bytecode and, secondly, implementing private channel communication on top of the π-calculus via the use of cryptographic primitives.

Other notable examples include a fully-abstract complier from an ML-like subset of F* [Swamy et al. 2011] to Javascript, enabling developers to reason about their programs on the high level of ML, without having to thoroughly understand the intricacies of Javascript [Fournet et al. 2013]. The full abstraction proof of the compiler is mechanized in F* and utilizes the notion of bisimulation discussed previously.

Alas, full abstraction does not always capture security properties we would like to enforce or could even be impossible to find such translations [Parrow 2016; Patrignani et al. 2019]. Hence the need to explore a more open space of secure compilation criteria that can be preserved against any adversarial context. In [Abate et al. 2019], different properties are investigated, ranging from simple *trace properties* (e.g. safety) to *hyperproperties* (e.g. non-interference) and *relational hyperproperties* (e.g. trace equivalence). Results indicate that most are easier to prove than full abstraction and provide strictly stronger security guarantees.

Another limitation of full abstraction manifests when the source language cannot express some constructs of the target language, thus creating issues in *back-translation*; a crucial step in the full abstraction proof, where target-level contexts are translated to equivalent source-level ones. A solution is examined in [Devriese et al. 2016], where back-translation is only considered *approximately*, up to a number of reduction steps. Furthermore, this work employs

commonly-used techniques to overcome the limitation of contextual equivalence, such as *cross-language logical relations* and *step-indexing*.

Kripke logical relations (KLR) are a particular form of logical relations, taking a form similar to Kripke semantics in intuitionistic logic. They have been extensively used to solve long-standing full abstraction problems, such as the aforementioned obstacle in the context of PCF [Ohearn and Riecke 1995]. When dealing with languages that support recursion, *step-indexing* restricts recursive operations to approximations up to a certain number of iterations. Step-indexing, along with other properties such as bi-orthogonality and realizability [Benton et al. 2010], has been successfully used to prove compilation correctness from a simply-typed functional language to a variant SECD machine [Benton and Hur 2009], as well as from an impure ML-like λ-calculus to idealized assembly [Hur and Dreyer 2011].

However, bisimulation and Kripke logical relations have complementary advantages, leading to an increased interest in combining them to get a more flexible and modular proof technique. [Hur et al. 2012] identifies the limitations of each technique and arrives at a novel formulation of a *relation transition system* (RTS). Specifically, RTSs marry the ability of bisimulation to reason about recursive structure coninductively, while at the same time supporting local-state reasoning via the transition semantics of KLRs.

Similarly, [Neis et al. 2015] introduces the notion of *parametric interlanguage simulations* (PILS), in order to overcome limitations that manifest when considering *compositional* compilation correctness, rather than whole-program compilation. All this is done in the context of a compiler from a higher-order imperative language to assembly with multiple optimization passes, extending the previous work of [Hur and Dreyer 2011] and [Hur et al. 2012].

5 WORK SO FAR

In this section, I will present the results I have formalized thus far. I will provide links to the Github repositories for each formalization effort and, whenever there is need to point to particular sub-components, I will provide links to HTML pages, where one can navigate interactively through the rendered Agda source code.

For a more detailed description of the work discussed here, I refer the reader to the publications attached separately. These include[5]:

- **[tyde.pdf]**
  An extended abstract on formalizing UTxO ledgers in Agda, presented at the workshop for *Type-Driven Development* (TyDe) of ICFP'19.

[5] The first two were published as part of my MSc thesis at the University of Utrecht.

- **[src.pdf]**
  An extended abstract on the initial setup of the BitML formalization, presented at the *Student Research Competition* of ICFP'19.
- **[eutxo.pdf]**
  A full paper on the extended UTxO model and its correspondence to state machines, presented at the *Workshop for Trusted Smart Contracts* (WTSC) of FC'20.

5.1 Formalizing Bitcoin Ledgers

We start with mechanizing the abstract model of Bitcoin transactions described in [Atzei et al. 2018], exploiting Agda's dependent type system for more expressivity.[6]

The formalization includes a dependently-typed encoding of *Bitcoin scripts*, where scripts are indexed by their result type to disallow invalid program combinations statically Moreover, we employ a DeBruijn encoding of script parameters — scripts are indexed by their context — in order to statically guarantee that only valid arguments are used. A *denotational semantics* of script execution under a particular environment is also given.

Moreover, a model of intrinsically-typed *Bitcoin transactions* is formalized, where type indices keep track of input/output sizes and ensure that a transaction is well-formed (e.g. all inputs have a corresponding witness).

Finally, we formulate the *consistency property* on ledgers, which is satisfied exactly when all submitted transaction are *valid* w.r.t. the so-far constructed ledger.

Note that most cryptographic primitives are postulated, following the paradigm set out by the Dolev-Yao model [Dolev and Yao 1983]. Specifically, we *postulate* ideal hashing functions, as well as public-key cryptographic primitives for signing and verifying signatures, but *implement* the exact procedures for signing transactions and m-of-n multi-signature schemes.

5.2 Formalizing the BitML Calculus

We continue by formalising the syntax and semantics of BitML contracts[7], as presented in Section 4 of [Bartoletti and Zunino 2018].

Initially, we tried out an instrinsically typed encoding of BitML contracts, but it turned out to overly complicate the rest of the development, so we opted out for a *simply-typed approach* without enforcing any additional static invariants *intrinsically*. Instead, desired properties are enforced *extrinsically*, by carrying the relevant proofs where necessary.

[6] https://github.com/omelkonian/formal-bitcoin
[7] https://github.com/omelkonian/formal-bitml

Then, a *small-step operational semantics* is given, formulated as a *labelled transition system* between *configurations*. These are presented in the form of inference rules, and we additionally prove that the associated proofs needed as hypotheses are *decidable*; using *proof-by-reflection* allows us to get these proofs automatically generated when considering *closed* terms (i.e. without any free variables). Furthermore, the inference rules are structured in such a way, so as to enjoy *equational reasoning*, where one can prove the *reflexive transitive closure* of the reduction relation by providing a linear series of individual steps.

We demonstrate that examining concrete contracts in this formal model is feasible, by giving a possible execution of the timed-commitment protocol as an *equational proof*, where the participant actually reveals the secret and collects her deposit back.

5.3 Formalizing Compilation from Bitml to Bitcoin

Having formalised both the high level model of BitML and the low level of Bitcoin transactions, we are ready to combine the two, so as to begin formulating the compilation correctness proof.[8]

Symbolic Model. Continuing from the intuition that proofs using the inference rules of the operational semantics correspond to possible executions, we mechanize a game-theoretical description of participant interaction, dubbed the *symbolic model*, as described in Section 5 of [Bartoletti and Zunino 2018].

In such a model, we can examine possible strategies and prove the (im)possibility of certain attacks, by providing a set of honest and adversary strategies that exhibit such behaviour.

We also prove *auxiliary lemmas* pertaining to the validity of strategies, which will be needed in the final proof of compilation correctness.

Computational Model. We define a similar game-theoretic model for the low level of Bitcoin transactions, dubbed the *computational model*, as described in Section 6 of [Bartoletti and Zunino 2018].

Participant moves now include submitting transactions and broadcasting messages to other participants, among others. Contrary to the symbolic model, we have not yet proved the relevant auxilliary lemmas; only the *proposition* has been formulated.

It is worth noting that it is at this point that we first discern the non-deterministic features of Bitcoin smart contracts, arising from the message-passing paradigm, as can commonly be encountered in the context of process calculi.

[8] https://github.com/omelkonian/formal-bitml-to-bitcoin

BitML Compiler. Finally, we are ready to implement the *compiler* from BitML contracts to Bitcoin transactions, based in Section 7 of [Bartoletti and Zunino 2018].

There, the compiler was originally presented as a set of inference rules, mostly for uniformity with the rest of the paper, but we actually implement the compiler as a function from *contract advertisements* (i.e. BitML contracts, coupled with certain preconditions that have to be met before stipulation) to an unordered set of transactions, hence performing the translation *deterministically*.

However, BitML's non-determinism is recovered when considering the order that the generated transactions will be submitted. In other words, the compiler generates all transactions that may be needed in some possible scenario. As a sanity check, we demonstrate that our mechanized compiler generates the expected Bitcoin transactions in the timed-commitment *example*, by checking against the results shown in the paper.

5.4 Formalizing UTxO-based Ledgers

Apart from the Bitcoin-specific model of transactions, we have also formalized generic UTxO-based transactions[9], based on the abstract model formulated in [Zahnentferner 2018].

Specifically, we have extended the UTxO model to include extra functionality available on the Cardano blockchain. These include additional information made available to the validator, which allows for more complicated transactional schemes (e.g. stateful contracts), as well as support for multiple currencies in a form similar to the one presented in [Zahnentferner 2019].

The formalization includes a simply-typed encoding of UTxO *transactions and ledgers*, as well as a rigid formulation of a transaction's *validity* w.r.t. a ledger.[10] Again, to enable the technique of *proof-by-reflection* we prove *decidability* of all validity conditions, which allows us to define *correct-by-construction* ledgers.

In order to be able to accommodate different UTxO-based blockchains, we do not fix the on-chain scripting language. Rather, we replace such scripts with their denotations, i.e. Agda functions.

In order to formalize the additional expressive power of the extended validators, we formulate a specific variant of state machines, dubbed *Constraint Emitting Machines (CEM)*. These resemble Mealy machines, as discussed in Subsection 4.3, however in our case the output is blockchain-specific, namely

[9] https://github.com/omelkonian/formal-utxo

[10] In the Bitcoin model, validity is referred to as consistency.

first-order constraints on the fields of the transaction that corresponds to the current CEM transition.

Moreover, we demonstrate an example contract defined as such a state machine, namely a *guessing game* where players submit guesses of a secret string.

Finally, we provide a mechanized proof of *bisimulation* between such a state machine and a UTxO-based ledger, such that CEM states correspond to data, attached in unspent transaction outputs. The proof can be conceptually divided into two directions.

Soundness states that, given a CEM transition, we can always find a corresponding transaction to extend the ledger corresponding to the source CEM state.

Completeness is the inverse, where we translate valid transaction submissions into CEM states, if the transaction actually refers to the state machine; otherwise, the CEM state remains the previous one. This direction of the bisimulation is *weak*, since the ledger could potentially contain irrelevant transactions to the current state machine, i.e., the transition takes an arbitrary number of *internal* steps until it reaches a *bisimilar* state.

6 FUTURE DIRECTIONS

After I have completed the mechanization of BitML's compilation correctness proof, I plan to investigate several different avenues, subject to time constraints.

BitML, In Retrospect. Formalizing a mathematical model does not only verify that claimed results hold mechanically, but also provides us with a more thoroughly described view of the model. Therefore, it might be the case that several improvements may become clear during the formal development. For instance, compilation correctness relies on a *coherence* relation between symbolic and computational executions, but it would be worthwhile to investigate whether other techniques used in non-blockchain PL, such as *back-translation*, would make the proof easier.

Integration with Plutus. In our UTxO formalization, we employed an abstract view of scripts, modelling them as functions. However, we lose important properties, such as decidable equality between scripts. This can be recovered if we instantiate scripts to a particular syntax, which would be Plutus in the case of Cardano. There is already a mechanized version of Plutus metatheory[11], also written in Agda, thus it would be interesting to integrate Plutus in our own formal development.

[11] https://github.com/input-output-hk/plutus/tree/master/metatheory

Verification of Concrete Contracts. In addition to validating a model's metatheory, one would also desire to validate concrete smart contracts. In the context of our BitML formalization, this would consist of writing BitML programs embedded in Agda and proving that certain properties are met. In the context of our UTxO formalization, we would hope for transferring proved properties of a state machine to equivalent properties on the transaction level.

PLFA Section on Secure Compilation. It is our hope that, through the formalization of the BitML compiler, we will be able to distill general principles on structuring secure compilation proofs. A valuable contribution, in the context of this thesis, would be to write a corresponding chapter for the *Programming Language Foundations in Agda* (PLFA) book [Wadler and Kokke 2019], as a way to efficiently disseminate these principles.

Formal Comparison with Ethereum. Continuing the UTxO bisimulation relation, it might be worthwhile to examine a correspondence with a model of Ethereum as well. [Zahnentferner and HK 2018] already provides a translation between the abstract UTxO model we have formalized and an abstract account-based blockchain, which we can try to integrate in the current formal development.

Kripke Relations for Blockchains. [Hirai 2018] has already used Kripke logical relations to model blockchain systems and, as discussed in Subsection 4.5, they have been extensively used in the theory of programming languages. Naturally, one should be tempted to investigate whether a deeper connection exists between KLRs and smart contract semantics.

REFERENCES

Martín Abadi. 1999. Protection in programming-language translations. In *Secure Internet programming*. Springer, 19–34.

Martín Abadi and Leslie Lamport. 1988. The existence of refinement mappings. (1988).

Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 256–25615.

Samson Abramsky. 1997. Game semantics for programming languages. In *MFCS*, Vol. 97. 25–29.

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2013. Full abstraction for PCF. *arXiv preprint arXiv:1311.6125* (2013).

Leonardo Alt and Christian Reitwießner. 2018. Smt-based verification of solidity smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 376–388.

Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. 2014a. Modeling bitcoin contracts by timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 7–22.

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

Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: a smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 215–228.

Danil Annenkov and Bas Spitters. 2019. Towards a smart contract verification framework in Coq. *arXiv preprint arXiv:1907.10674* (2019).

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*. Springer, 164–186.

Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. 2019. Developing secure Bitcoin contracts with BitML. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1124–1128.

Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. 2018. A formal model of Bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*. Springer, 541–560.

Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a transaction ledger: A composable treatment. In *Annual International Cryptology Conference*. Springer, 324–356.

Jos CM Baeten. 2005. A brief history of process algebra. *Theoretical Computer Science* 335, 2-3 (2005), 131–146.

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.

Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. 2018. Fun with Bitcoin smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 432–449.

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.

Massimo Bartoletti and Roberto Zunino. 2019a. Formal models of Bitcoin contracts: a survey. *Frontiers in Blockchain* 2 (2019), 8.

Massimo Bartoletti and Roberto Zunino. 2019b. Verifying liquidity of Bitcoin contracts. In *International Conference on Principles of Security and Trust*. Springer, 222–247.

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. *ACM Sigplan Notices* 44, 9 (2009), 97–108.

Nick Benton, Chung-Kil Hur, and Cambridge Paris. 2010. Realizability and compositional compiler correctness for a polymorphic language. *Technical Report MSR-TR-2010–62, Microsoft Research, Tech. Rep.* (2010).

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

Jan A Bergstra and Jan Willem Klop. 1987. ACP τ a universal axiom system for process specification. In *Workshop on Algebraic Methods*. Springer, 445–463.

Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. 2019. Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. *arXiv preprint arXiv:1909.08671* (2019).

Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. 2015. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 104–121.

Marcello M Bonsangue, Jan Rutten, and Alexandra Silva. 2008. Coalgebraic logic and synthesis of Mealy machines. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 231–245.

Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).

Christian Cachin, Angelo De Caro, Pedro Moreno-Sanchez, Björn Tackmann, and Marko Vukolic. 2017. The Transaction Graph for Modeling Blockchain Semantics. *IACR Cryptology ePrint Archive* 2017 (2017), 1070.

Xiaohong Chen, Daejun Park, and Grigore Roşu. 2018. A language-independent approach to smart contract verification. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 405–413.

Bram Cohen. 2003. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, Vol. 6. 68–72.

Christian Colombo, Joshua Ellul, and Gordon J Pace. 2018. Contracts over smart contracts: Recovering from violations dynamically. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 300–315.

Pierre-Louis Curien. 2007. Definability and full abstraction. *Electronic Notes in Theoretical Computer Science* 172 (2007), 301–310.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 164–177.

Edsger W Dijkstra. 1975a. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

Edsger W. Dijkstra. 1975b. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Danny Dolev and Andrew Yao. 1983. On the security of public key protocols. *IEEE Transactions on information theory* 29, 2 (1983), 198–208.

Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of computer programming* 17, 1-3 (1991), 35–75.

Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully abstract compilation to JavaScript. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 371–384.

Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 281–310.

Juan A Garay and Aggelos Kiayias. 2018. SoK: A Consensus Taxonomy in the Blockchain Era. *IACR Cryptology ePrint Archive* 2018 (2018), 754.

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.

LM Goodman. 2014. Tezos—a self-amending crypto-ledger White paper. *URL: https://www. tezos. com/static/papers/white_paper. pdf* (2014).

Daniele Gorla and Uwe Nestmann. 2016. Full abstraction for expressiveness: History, myths and facts. *Mathematical Structures in Computer Science* 26, 4 (2016), 639–654.

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), 1–27.

Joseph Y Halpern and Rafael Pass. 2017. A knowledge-based analysis of the blockchain protocol. *arXiv preprint arXiv:1707.08751* (2017).

Dominik Harz and William Knottenbelt. 2018. Towards safer smart contracts: A survey of languages and verification methods. *arXiv preprint arXiv:1809.09805* (2018).

Maurice Herlihy. 2019. Blockchains from a distributed computing perspective. *Commun. ACM* 62, 2 (2019), 78–85.

Yoichi Hirai. 2018. Blockchains as Kripke Models: An Analysis of Atomic Cross-Chain Swap. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 389–404.

Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

Charles Antony Richard Hoare. 1980. A model for communicating sequential process. (1980).

Charles Antony Richard Hoare, Stephen D Brookes, and Andrew William Roscoe. 1981. *A theory of communicating sequential processes*. Oxford University Computing Laboratory, Programming Research Group.

Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. 1992. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices* 27, 5 (1992), 1–164.

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 133–146.

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. *ACM SIGPLAN Notices* 47, 1 (2012), 59–72.

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better late than never: a fully-abstract semantics for classical processes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

Shuvendu K Lahiri and Shaz Qadeer. 2009. Complexity and algorithms for monomial and clausal predicate abstraction. In *International Conference on Automated Deduction*. Springer, 214–229.

James Laird. 2007. A fully abstract trace semantics for general references. In *International Colloquium on Automata, Languages, and Programming*. Springer, 667–679.

Stefano Lande and Roberto Zunino. 2018. SoK: unraveling Bitcoin smart contracts. *Principles of Security and Trust LNCS 10804* (2018), 217.

Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 42–54.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

John R Longley and Gordon Plotkin. 2000. Logical full abstraction and PCF. (2000).

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and*

communications security. 254–269.

Diego Marmsoler. 2019. Towards Verified Blockchain Architectures: A Case Study on Interactive Architecture Verification. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 204–223.

Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples.

Anastasia Mavridou and Aron Laszka. 2018. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*. Springer, 523–540.

John McCarthy and James Painter. 1967. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science* 1 (1967).

George H Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (1955), 1045–1079.

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.

George Milne and Robin Milner. 1979. Concurrent processes and their syntax. *Journal of the ACM (JACM)* 26, 2 (1979), 302–321.

Robin Milner. 1977. Fully abstract models of typed λ-calculi. *Theoretical Computer Science* 4, 1 (1977), 1–22.

Robin Milner. 1979a. An algebraic theory for synchronization. In *Theoretical Computer Science 4th GI Conference*. Springer, 27–35.

Robin Milner. 1979b. Flowgraphs and flow algebras. *Journal of the ACM (JACM)* 26, 4 (1979), 794–818.

Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer. https://doi.org/10.1007/3-540-10235-3

Robin Milner. 1989. A calculus of mobile process. *LFCS Report, Dept. of Computer Science, Univ. of Edinbough* (1989).

John C Mitchell. 1993. On abstraction and the expressive power of programming languages. *Science of Computer Programming* 21, 2 (1993), 141–163.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568.

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

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2015. Bitcoin and cryptocurrency technologies. *Curso elaborado pela* (2015).

George C Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices* 33, 5 (1998), 333–344.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 166–178.

Jakob Botsch Nielsen and Bas Spitters. 2019. Smart Contract Interactions in Coq. *arXiv preprint arXiv:1911.04732* (2019).

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

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

Peter W Ohearn and Jon G Riecke. 1995. Kripke logical relations and PCF. *Information and Computation* 120, 1 (1995), 107–116.

Joachim Parrow. 2016. General conditions for full abstraction. *Mathematical Structures in Computer Science* 26, 4 (2016), 655–657.

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

Benjamin C Pierce. 1997. Foundational Calculi for Programming Languages. *The Computer Science and Engineering Handbook* 1997 (1997), 2190–2207.

Gordon D Plotkin. 1977. LCF considered as a programming language. *Theoretical computer science* 5, 3 (1977), 223–255.

Gordon D Plotkin. 1981. A structural approach to operational semantics. (1981).

Jon G Riecke. 1993. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science* 3, 4 (1993), 387–415.

Davide Sangiorgi. 1996. A theory of bisimulation for the π-calculus. *Acta informatica* 33, 1 (1996), 69–97.

Davide Sangiorgi. 2011. *Introduction to bisimulation and coinduction*. Cambridge University Press.

Dana Scott. 1970. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford.

Ilya Sergey and Aquinas Hobor. 2017. A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security*. Springer, 478–493.

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

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Anton Setzer. 2018. Modelling Bitcoin in Agda. *arXiv preprint arXiv:1804.06398* (2018).

Ehud Shapiro. 1991. Separating concurrent languages with categories of language embeddings. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 198–208.

Tim Sheard, Aaron Stump, and Stephanie Weirich. 2010. Language-based verification will change the world. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 343–348.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. *ACM SIGPLAN Notices* 46, 9 (2011), 266–278.

Petar Tsankov. 2018. Security analysis of smart contracts in datalog. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 316–322.

PD Van Der Walt. 2012. *Reflection in agda*. Master's thesis.

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.

Philip Wadler and Wen Kokke. 2019. *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/.

Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2018. Formal specification and verification of smart contracts for Azure blockchain. *arXiv preprint arXiv:1812.08829* (2018).

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

Joachim Zahnentferner. 2019. Multi-Currency Ledgers. (2019), To Appear.

Joachim Zahnentferner and Input Output HK. 2018. *Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies*. Technical Report. Cryptology ePrint Archive, Report 2018/262, 2018. https://epri nt. iacr. org ….