Formalizing Extended UTxO and BitML Calculus in Agda

Laying the foundations for the formal verification of smart contracts

Orestis Melkonian

A thesis submitted for the Master of Science degree

Department of Information and Computing Sciences

Utrecht University

July 2019

Supervisors:	Wouter Swierstra	(Utrecht University)
	Manuel M.T. Chakravarty	(Input Output HK)
2nd Examiner:	Gabriele Keller	(Utrecht University)

To Emilia,
for her infinite love, support and devotion.

Abstract

Smart contracts – programs that run on a blockchain – allow for sophisticated transactional schemes, but their concurrent execution makes it difficult to reason about their behaviour and bugs in smart contracts have lead to significant monetary losses (e.g. DAO attack). For that reason, increasingly more attention is given to formal methods, that guarantee that such fatal scenarios are not possible.

We attempt to advance the state-of-the-art for a language-oriented, type-driven account of smart contracts by formalizing two well-established models in Agda and mechanizing the corresponding meta-theory.

The first concerns an abstract model for UTxO-based ledgers, such as Bitcoin, which we further extend to cover features of the Cardano blockchain, namely more expressive scripts and built-in support for user-issued cryptocurrencies.

The second object of study is BitML, a process calculus specifically targeting Bitcoin smart contracts. We present a mechanized semantics of BitML contracts and its small-step semantics, as well as a mechanized account of BitML's symbolic model over participant strategies.

Finally, we sketch the way towards a *certified compiler* from BitML contracts to UTxO transactions, where all behaviours manifesting on BitML's symbolic model can safely be transported to the UTxO level.

Acknowledgements

First, I would like thank my supervisors, Wouter and Manuel. Their constant push for excellence motivated me throughout this thesis and the result would not have been the same without them. Wouter's expertise on dependently-typed models, as well as Manuel's deep understanding of how a blockchain operates were invaluable and significantly shaped the approach taken in this thesis.

Moreover, I would like to thank several researchers from IOHK for helpful discussion while the thesis was still in progress, especially Philip Wadler, James Chapman and Michael Peyton Jones.

Contents

SECTION 1

Introduction

Blockchain technology has opened a whole array of interesting new applications, such as multi-party computation [Andrychowicz et al. 2014], fair protocol design fair [Bentov and Kumaresan 2014] and 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 on the model of Ethereum's scripting language led to 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, 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.

Research Question. The aim of this thesis is to provide a mechanized formal model of an abstract distributed ledger equipped with smart contracts, in which one can begin to formally investigate the expressiveness of the extended UTxO model. Moreover, we hope to lay a foundation for a formal comparison with account-based models used in Ethereum. Put concisely, the broader research question posed is:

How does the UTxO model of smart contracts compare to Ethereum's account-based one?

Overview.

- Section 2 reviews some basic definitions related to blockchain technology and introduces important literature, which will be the main subject of study throughout the development of our reasoning framework. Moreover, we give an overview of related work, putting an emphasis on existing tools based on static analysis.
- Section 3 describes the technology we will use to formally reason about the problem at hand and some key design decisions we set upfront.
- Section 4 describes the formalization of an abstract model for UTxO-based blockchain ledgers.

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

- Section 5 concerns the formalization of our second object of study, the Bitcoin Modelling Language.
- Section 6 gives an overview of relevant previous work, ranging from static analysis tools to type-driven verification approaches.
- Section 7 discusses possible next steps to continue the line of work stemming from this thesis.
- Section 8 concludes with a general overview of our contributions and reflects on the chosen methodology.

SECTION 2

Background

2.1 Distributed Ledger Technology: Blockchain

Cryptocurrencies rely on distributed ledgers, where there is no central authority managing the accounts and keeping track of the history of transactions. One particular instance of distributed ledgers are blockchain systems, where transactions are bundled together in blocks, which are linearly connected with hashes and distributed to all peers. The blockchain system, along with a consensus protocol deciding on which competing fork of the chain is to be included, maintains an immutable distributed ledger (i.e. the history of transactions).

Validity of the transactions is tightly coupled with a consensus protocol, which makes sure peers in the network only validate well-behaved and truthful transactions and are, moreover, properly incentivized to do so.

The absence of a single central authority that has control over all assets of the participants allows for shared control of the evolution of data (in this case transactions) and generally leads to more robust and fair management of assets.

While cryptocurrencies are the major application of blockchain systems, one could easily use them for any kind of valuable asset, or even as general distributed databases.

2.2 Smart Contracts

Most blockchain systems come equipped with a scripting language, where one can write *smart contracts* that dictate how a transaction operates. A smart contract could, for instance, pose restrictions on who can redeem the output funds of a transaction.

One could view smart contracts as a replacement of legal frameworks, providing the means to conduct contractual relationships completely algorithmically.

While previous work on writing financial contracts [Peyton Jones et al. 2000] suggests it is fairly straightforward to write such programs embedded in a general-purpose language (in this case Haskell) and to reason about them with *equational reasoning*, it is restricted in the centralized setting and, therefore, does not suffice for our needs.

Things become much more complicated when we move to the distributed setting of a blockchain [Buterin et al. 2014; Nakamoto 2008]. Hence, there is a growing need for methods and tools that will enable tractable and precise reasoning about such systems.

Numerous scripting languages have appeared recently [Seijas et al. 2016], spanning a wide spectrum of expressiveness and complexity. While language design can impose restrictions on what a language can express, most of these restrictions are inherited from the accounting model to which the underlying system adheres.

In the next section, we will discuss the two main forms of accounting models:

(1) **UTxO-based**: stateless models based on *unspent transaction outputs*
(2) **Account-based**: stateful models that explicitly model interaction between *user accounts*

2.3 UTxO-based: Bitcoin

The primary example of a UTxO-based blockchain is Bitcoin [Nakamoto 2008]. Its blockchain is a linear sequence of *blocks* of transactions, starting from the initial *genesis* block. Essentially, the blockchain acts as a public log of all transactions that have taken place, where each transaction refers to outputs of previous transactions, except for the initial *coinbase* transaction of each block. Coinbase transactions have no inputs, create new currency and reward the miner of that block with a fixed amount. Bitcoin also provides a cryptographic protocol to make sure no adversary can tamper with the transactional history, e.g. by making the creation of new blocks computationally hard and invalidating the "truthful" chain statistically impossible.

A crucial aspect of Bitcoin's design is that there are no explicit addresses included in the transactions. Rather, transaction outputs are actually program scripts, which allow someone to claim the funds by giving the proper inputs to the validator script (i.e. arguments that make the script return *true*)[2]. Thus, although there are no explicit user accounts in transactions, the effective available funds of a user are all the *unspent transaction outputs* (UTxO) that he can claim (e.g. by providing a digital signature).

2.3.1 Script

In order to write such scripts in the outputs of a transaction, Bitcoin provides a low-level, Forth-like, stack-based scripting language, called Script. Script is intentionally not Turing-complete (e.g. it does not provide looping structures), in order to have more predictable behaviour. Moreover, only a very restricted set of "template" programs are considered standard, i.e. allowed to be relayed from node to node.

Script Notation. Programs in script are a linear sequence of either data values (e.g. numbers, hashes) or built-in operations (distinguished by their OP_ prefix).

The stack is initially considered empty and we start reading inputs from left to right. When we encounter a data item, we simply push it to the top of the stack. On encountering an operation, we pop the necessary number of arguments from the stack, apply the operation and push the result back. The evaluation function $[\![\_]\!]$ executes the given program and returns the final result at the top of the stack. For instance, adding two numbers looks like this:

$$[\![1\ 2\ \mathrm{OP\_ADD}]\!] = 3$$

P2PKH. The most frequent example of a "standard" program in Script is the *pay-to-pubkey-hash* (P2PKH) type of scripts. Given a hash of a public key $<\mathrm{pub}\#>$, a P2PKH output carries the

[2] When access to a transaction output is restricted via a validator script, we sometimes say that the output is *locked* by the script.

following script:

$$\text{OP\_DUP OP\_HASH <pub\#> OP\_EQ OP\_CHECKSIG}$$

where OP_DUP duplicates the top element of the stack, OP_HASH replaces the top element with its hash, OP_EQ checks that the top two elements are equal, OP_CHECKSIG verifies that the top two elements are a valid pair of a digital signature of the transaction data and a public key hash.

The full script will be run when the output is claimed (i.e. used as input in a future transaction) and consists of the P2PKH script, preceded by the digital signature of the transaction by its owner and a hash of his public key. Given a digital signature <sig> and a public key hash <pub>, a transaction is valid when the execution of the script below evaluates to True.

$$\text{<sig> <pub> OP\_DUP OP\_HASH <pub\#> OP\_EQ OP\_CHECKSIG}$$

To clarify, assume a scenario where Alice want to pay Bob ₿ 10. Bob provides Alice with the cryptographic hash of his public key <pub#> and Alice can submit a transaction of ₿ 10 with the following output script:

$$\text{OP\_DUP OP\_HASH <pub\#> OP\_EQ OP\_CHECKSIG}$$

After that, Bob can submit another transaction that uses this output by providing the digital signature of the transaction <sig> (signed with his private key) and his public key <pub>. It is easy to see that the resulting script evaluates to True.

P2SH. A more complicated script type is *pay-to-script-hash* (P2SH), where output scripts simply authenticate against a hash of a *redeemer* script <red#>:

$$\text{OP\_HASH <red\#> OP\_EQ}$$

A redeemer script <red> resides in an input which uses the corresponding output. The following two conditions must hold for the transaction to go through:

(1) $[\![\text{<red>}]\!] = \text{True}$

(2) $[\![\text{<red> OP\_HASH <red\#> OP\_EQ}]\!] = \text{True}$

Therefore, in this case the script residing in the output is simpler, but inputs can also contain arbitrary redeemer scripts (as long as they are of a standard "template").

In this thesis, we will model scripts in a much more general, mathematical sense, so we will eschew from any further investigation of properties particular to Script.

2.3.2 The BitML Calculus

Although Bitcoin is the most widely used blockchain to date, many aspects of it are poorly documented. In general, there is a scarcity of formal models, most of which are either introductory or exploratory.

Some of the most involved and mature previous work on formalizing the operation of Bitcoin is the Bitcoin Modelling Language (BitML) [Bartoletti and Zunino 2018]. First, an idealistic *process calculus* that models Bitcoin contracts is introduced, along with a detailed small-step reduction semantics that models how contracts interact and its non-determinism accounts for the various outcomes.

The semantics consist of transitions between *configurations*, abstracting away all the cryptographic machinery and implementation details of Bitcoin. Consequently, such operational semantics allow one to reason about the concurrent behaviour of the contracts in a *symbolic* setting.

The authors then provide a compiler from BitML contracts to “standard” Bitcoin transactions, proven correct via a correspondence between the symbolic model and the computational model operating on the Bitcoin blockchain. We will return for a more formal treatment of BitML in Section 5.

2.3.3 *Extended UTxO*

In this work, we will consider the version of the UTxO model used by IOHK’s Cardano blockchain[3]. We will refer to this variant as *Extended UTxO* (eUTxO). In contrast to Bitcoin’s *proof-of-work* consensus protocol [Nakamoto 2008], Cardano’s *Ouroboros* protocol [Kiayias et al. 2017] is *proof-of-stake*. This, however, is of no concern at the scope of the abstract accounting model, thus we refrain from formally modelling and comparing different consensus techniques.

The actual extension we care about is the inclusion of *data scripts* in transaction outputs, which essentially provides the validation script in the corresponding input with additional information of an arbitrary type.

This extension of the UTxO model has already been implemented[4], but only informally documented[5]. The reason to extend the UTxO model with data scripts is to bring more expressive power to UTxO-based blockchains, hopefully bringing it on par with Ethereum’s account-based scripting model (see Section 2.4).

However, there is no formal argument to support this claim, and it is the goal of this thesis to provide the first formal investigation of the expressiveness introduced by this extension.

2.4 Account-based: Ethereum

On the other side of the spectrum, lies the second biggest cryptocurrency today, Ethereum [Buterin et al. 2014]. In contrast to UTxO-based systems, Ethereum has a built-in notion of user addresses and operates on a stateful accounting model. It goes even further to distinguish *human accounts* (controlled by a public-private key pair) from *contract accounts* (controlled by some EVM code).

This added expressiveness is also reflected in the quasi-Turing-complete low-level stack-based bytecode language in which contract code is written, namely the *Ethereum Virtual Machine* (EVM). EVM is mostly designed as a target, to which other high-level user-friendly languages will compile.

Solidity. The most widely adopted language that targets the EVM is *Solidity*, whose high-level object-oriented design makes writing common contract use-cases (e.g. crowdfunding campaigns, auctions) rather straightforward.

One of Solidity’s most distinguishing features is the concept of a contract’s *gas*; a limit to the amount of computational steps a contract can perform. When a a transaction is created, its owner specifies a certain amount of gas the contract can consume and pays a transaction fee proportional

[3]www.cardano.org

[4]https://github.com/input-output-hk/plutus/tree/master/wallet-api/src/Ledger

[5]https://github.com/input-output-hk/plutus/blob/master/docs/extended-utxo/README.md

to it. In case of complete depletion (i.e. all gas has been consumed before the contract finishes its execution), all global state changes are reverted as if the contract had never been run. This is a necessary ingredient for smart contract languages that provide arbitrary looping behaviour, since non-termination of the validation phase is certainly undesirable.

SECTION 3

Methodology

3.1 Scope

At this point, we have to stress the fact that we are not aiming for a formalization of a fully-fledged blockchain system with all its bells and whistles, but rather focus on the underlying accounting model. Therefore, we will omit details concerning cryptographic operations and aspects of the actual implementation of such a system. Instead, we will work on an abstract layer that postulates the well-behavedness of these subcomponents, which will hopefully lend itself to more tractable reasoning and give us a clear overview of the essence of the problem.

Restricting the scope of our attempt is also motivated from the fact that individual components such as cryptographic protocols are orthogonal to the functionality we study here. This lack of tight cohesion between the components of the system allows one to safely factor each one out and formalize it independently.

It is important to note that this is not always the case for every domain. A prominent example of this are operating systems, which consist of intricately linked subcomponents (e.g. drivers, memory modules), thus making it impossible to trivially divide the overall proof into small independent ones. In order to overcome this complexity burden, one has to invent novel ways of modular proof mechanization, as exemplified by *CertiKOS* [Chen et al. 2016], a formally verified concurrent OS.

3.2 Proof Mechanization

Fortunately, the sub-components of the system we are examining are not no interdependent, thus lending themselves to separate treatment. Nonetheless, the complexity of the sub-system we care about is still high and 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 a type-logical manner. Proof mechanization will allow us to discover edge cases and increase the confidence of the model under investigation.

3.3 Agda

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. Moreover, programs and proofs share the same structure, e.g. induction in the proof manifests itself as recursion in the program.

While Agda is not ideal for large software development, 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.

Limitation. The main limitation of Agda lies in its lack of a proper proof automation system. While there has been work on providing Agda with such capabilities [Kokke and Swierstra 2015], it requires moving to a meta-programming mindset which would be an additional programming hindrance.

A reasonable alternative would be to use Coq [Barras et al. 1997], which provides a pragmatic scripting language for programming *tactics*, i.e. programs that work on proof contexts and can generate new sub-goals. This approach to proof mechanization has, however, been criticized for widening the gap between informal proofs and programs written in a proof assistant. This clearly goes against the aforementioned principle of *proofs-as-programs*.

3.4 The IOHK approach

At this point, we would like to mention the specific approach taken by IOHK[6]. In contrast to numerous other companies currently creating cryptocurrencies, its main focus is on provably correct protocols with a strong focus on peer-reviewing and robust implementations, rather than fast delivery of results. This is evidenced by the choice of programming languages (Agda, Coq, Haskell, Scala among other) – all functional programming languages with rich type systems – and the use of *property-based testing* [Claessen and Hughes 2011] for the production code.

IOHK's distinct feature is that it advocates a more rigorous development pipeline; ideas are initially worked on paper by pure academics, which create fertile ground for starting formal verification in a proof assistant for more confident results, which result in a prototype/reference implementation in Haskell, which informs the production code-base (also written in Haskell) on the properties that should be tested.

Since this thesis is done in close collaboration with IOHK, it is situated on the second step of aforementioned pipeline; while there has been work on writing papers about the extended UTxO model along with the actual implementation in Haskell, there is still no complete and mechanized account of its properties.

3.5 Functional Programming Principles

One last important manifestation of the functional programming principles behind IOHK is the choice of a UTxO-based cryptocurrency itself.

On the one hand, one can view a UTxO ledger as a dataflow diagram, whose nodes are the submitted transactions and edges represent links between transaction inputs and outputs. On the other hand, account-based ledgers rely on a global state and transaction have a much more complicated specification.

[6]https://iohk.io/

The key point here is that UTxO-based transaction are just pure mathematical functions, which are much more straightforward to model and reason about. Coming back to the principles of functional programming, one could contrast this with the difference between functional and imperative programs. One can use *equational reasoning* for functional programs, due to their *referential transparency*, while this is not possible for imperative programs that contain side-effectful commands. Therefore, we hope that these principles will be reflected in the proof process itself; one would reason about purely functional UTxO-based ledgers in a compositional manner.

The following sections give an overview of the progress made so far on the Agda formalization of the two main subjects of study, namely the Extended UTxO model and the BitML calculus. For the sake of brevity, we refrain from showing the full Agda code along with the complete proofs, but rather provide the most important datatypes and formalized results and explain crucial design choices we made along the way. Furthermore, we will omit notational burden imposed by technicalities particular to Agda, such as *universe polymorphism* and *proof irrelevance*.

SECTION 4

Formal Model I: Extended UTxO

We now set out to model the accounting model of a UTxO-based ledger. We will provide a inherently-typed model of transactions and ledgers; as a result of the development we can show new meta-theoretical results, which we formalize. Moreover, we showcase the reasoning abilities of our model by giving an example of a correct-by-construction ledger. All code is publicly available on Github[7].

We start with the basic types, keeping them abstract since we do not care about details arising from the encoding in an actual implementation. In Agda, *abstract data types* can be realized as *parameters* to the current module:

```
module UTxO.Types (Value : Set) (Hash : Set) where ...
```

For simplicity, we can represent both cryptocurrency values and hashes as natural numbers, but we will later provide a more extensive datatype for values, as we shall see in Section 4.8.

There is also the notion of the *state* of a ledger, which will be provided to transaction scripts and allow them to have stateful behaviour for more complicated schemes (e.g. imposing time constraints). The state components have not been finalized yet, but can easily be extended later when we actually investigate examples with expressive scripts that make use of state information, such as the current length of the ledger (*height*). For the simple examples we will present here, the following model suffices:

```
record State : Set where
  field height : ℕ
          ⋮
```

As mentioned previously, we will not dive into the verification of the cryptographic components of the model, hence we postulate an *irreversible* hashing function which, given any value of any type, gives back an address (i.e. a natural number) and is furthermore *injective* (i.e. it is highly unlikely for two different values to have the same hash).

```
record HashFunction (A : Set) : Set where
  field hashF     : A → Hash
        injective : ∀ {x y} → x# ≡ y# → x ≡ y
```

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

```
postulate
  _# : ∀ {A : Set} → HashFunction A
```

For convenience, we postulate a hash function _# that works for all types and denote functional application of the first field to an element x simply as x#.

4.1 Transactions

In order to model transactions that are part of a distributed ledger, we need to first define transaction *inputs* and *outputs*.

```
record TxOutputRef : Set where
  constructor _ @ _
  field id    : Hash
        index : ℕ

record TxInput : Set where
  field outputRef : TxOutputRef

        R D       : 𝕌
        redeemer  : State → el R
        validator : State → Value → PendingTx → el R → el D → Bool
```

Output references consist of the address that a transaction hashes to, as well as the index in this transaction's list of outputs. *Transaction inputs* refer to some previous output in the ledger, but also contain two types of scripts. The *redeemer* provides evidence of authorization to spend the output. The *validator* then checks whether this is so, having access to the current state of the ledger, the bitcoin output, an overview of the current transaction (*PendingTx*) and data provided by the redeemer and the *data script* (residing in outputs). It is also noteworthy that we immediately model scripts by their *denotational semantics*, omitting unnecessary details relating to concrete syntax, lexing and parsing.

Notice that the result types of redeemers and data scripts are not any Agda type (*Set*), but rather reside in a more restricted universe 𝕌, which can only represent *first-order* data:

```
data 𝕌 : Set where
  UNIT BOOL NAT : 𝕌
  LIST          : 𝕌 → 𝕌
  PRODUCT SUM   : 𝕌 → 𝕌 → 𝕌

el : 𝕌 → Set
el UNIT            = ⊤
el BOOL            = Bool
el NAT             = ℕ
el (PRODUCT x y)   = el x × el y
el (SUM x y)      = el x ⊎ el y
el (LIST x)       = List (el x)
```

This construction is crucial when we later need to check equality between types, since function types would lead to undecidable equality.

Pending transactions are collections of hashes, which are involved in the current transaction. These consist of the hash of the transaction itself, as well as the hashes for all scripts residing in inputs or outputs:

```
record PendingTxInput : Set where
  field
    validatorHash : Hash
    redeemerHash : Hash

record PendingTxOutput : Set where
  field
    dataHash      : Hash

record PendingTx : Set where
  field
    txHash : Hash
    inputs  : List PendingTxInput
    outputs : List PendingTxOutput
```

Transaction outputs send a bitcoin amount to a particular address, which either corresponds to a public key hash of a blockchain participant (P2PKH) or a hash of a next transaction's script (P2SH). Here, we opt to embrace the *inherently-typed* philosophy of Agda and model the type of addresses as an *abstract data type*. That is, we package the following definitions in a module with such a parameter, hence allowing whoever imports the *UTxO* library to use a custom datatype, as long as it is equipped with a hash function and decidable equality:

```
module UTxO (Address : Set)
            (_#ₐ : Hash Address)
            (_≟ₐ_ : Decidable {A = Address} _≡_) where

record TxOutput : Set where
  field value     : Value
        address   : Address

        Data      : 𝕌
        dataScript : State → el Data
```

```
record Tx : Set where
  field inputs  : Set⟨ TxInput ⟩
        outputs : List TxOutput
        forge   : Value
        fee     : Value
Ledger : Set
Ledger = List Tx
```

Transaction outputs consist of a bitcoin amount and the address (out of the available ones) this amount is sent to, as well as the data script, which provides extra information to the aforementioned validator and allows for more expressive schemes. Investigating exactly the extent of this expressiveness is one of the main goals of this thesis.

For a transaction to be submitted, one has to check that each input can actually spend the output it refers to. At this point of interaction, one must combine all scripts, as shown below:

```
runValidation  :  PendingTx
               → (i : TxInput)
               → (o : TxOutput)
               → D i ≡ Data o
               → State
               → Bool
runValidation ptx i o refl st = validator i st (value o) ptx (redeemer i st) (dataScript o st)
```

Note that the intermediate types carried by the respective input and output must align, evidenced by the equality proof that is required as an argument.

Example Transaction. Assume Alice wants to transfer ₿ 10 to Bob and has access to a previous transaction output of ₿ 11 that she can redeem. By paying a transactional fee of ₿ 1, she can submit a transaction that redeems the funds of the unspent output (by providing the hash of its validator, equal to $\mathbb{A}$), and propagates the remaining funds to Bob's address $\mathbb{B}$ (equal to the validator hash of t_{after}): Figure 1 shows the relevant parts of the transactions involved. We do not display the scripts involved here, but we will see their usage in a more extensive example in Section 4.9.

4.2 Unspent Transaction Outputs

With the basic modelling of a ledger and its transaction in place, it is fairly straightforward to inductively define the calculation of a ledger's unspent transaction outputs:

```
unspentOutputs : Ledger → Set⟨ TxOutputRef ⟩
unspentOutputs [] = ∅
unspentOutputs (tx :: txs) = (unspentOutputs txs \ spentOutputsTx tx) ∪ unspentOutputsTx tx
  where
    spentOutputsTx , unspentOutputsTx : Tx → Set⟨ TxOutputRef ⟩
```

$$\begin{aligned}
\mathit{spentOutputsTx} &= (\mathit{outputRef}\ \langle\$\rangle\ \_) \circ \mathit{inputs} \\
\mathit{unspentOutputsTx}\ \mathit{tx} &= (\mathit{tx}\#\ \ @\ \ \_)\ \langle\$\rangle\ \mathit{indices}\ (\mathit{outputs}\ \mathit{tx})
\end{aligned}$$

4.3 Validity of Transactions

In order to submit a transaction, one has to make sure it is valid with respect to the current ledger. We model validity as a record indexed by the transaction to be submitted and the current ledger:

The first four conditions make sure the transaction references and types are well-formed, namely that inputs refer to actual transactions (*validTxRefs*, *validOutputIndices*) which are unspent so far (*validOutputRefs*), but also that intermediate types used in interacting inputs and outputs align (*validDataScriptTypes*).

The last four validation conditions are more interesting, as they ascertain the validity of the submitted transaction, namely that the bitcoin values sum up properly (*preservesValues*), no output is spent twice (*noDoubleSpending*), validation succeeds for each input-output pair (*allInputsValidate*) and outputs hash to the hash of their corresponding validator script (*validateValidHashes*).

The definitions of lookup functions are omitted, as they are uninteresting. The only important design choice is that, instead of modelling lookups as partial functions (i.e. returning *Maybe*), they require a membership proof as an argument moving the responsibility to the caller (as evidenced by their usage in the validity conditions).

Type-safe interface. Since we only wish to construct ledgers that are valid, i.e. submitted transactions are valid with respect to the constructed ledger, we only expose a type-safe interface as a proof-carrying variant of the standard list construction:

```
-- 1) Previous output redeemable by Alice
t_before : Tx
t_before = record {..., outputs = [₿ 11  @  𝔸], ...}
-- 2) Alice sends ₿ 10 to Bob
t : Tx
t = record { inputs   = [t_before#  @  0]
           ; outputs = [₿ 10  @  𝔹]
           ; forge    = ₿ 0
           ; fee      = ₿ 1}
-- 3) Bob spends them in a future transaction
t_after : Tx
t_after = record {inputs = [t#  @  0], ...}
```

Fig. 1. Example transaction: Alice sends B 10 to Bob.

```
record IsValidTx (tx : Tx) (l : Ledger) : Set where
  field
    validTxRefs :
      ∀ i → i ∈ inputs tx →
        Any (λ t → t# ≡ id (outputRef i)) l

    validOutputIndices :
      ∀ i → (i∈ : i ∈ inputs tx) →
        index (outputRef i) <
          length (outputs (lookupTx l (outputRef i) (validTxRefs i i∈)))

    validOutputRefs :
      ∀ i → i ∈ inputs tx →
        outputRef i ∈ unspentOutputs l

    validDataScriptTypes :
      ∀ i → (i∈ : i ∈ inputs tx) →
        D i ≡ D (lookupOutput l (outputRef i) (validTxRefs i i∈) (validOutputIndices i i∈))
      -----------------------------------------------------------------------------------

    preservesValues :
      forge tx + sum (mapWith∈ (inputs tx) λ {i} i∈ →
        lookupValue l i (validTxRefs i i∈) (validOutputIndices i i∈))
          ≡
      fee tx + sum (value ⟨$⟩ outputs tx)

    noDoubleSpending :
      noDuplicates (outputRef ⟨$⟩ inputs tx)

    allInputsValidate :
      ∀ i → (i∈ : i ∈ inputs tx) →
        let out = lookupOutput l (outputRef i) (validTxRefs i i∈) (validOutputIndices i i∈)
            ptx = mkPendingTx l tx validTxRefs validOutputIndices
        in  T (runValidation ptx i out (validDataScriptTypes i i∈) (getState l))

    validateValidHashes :
      ∀ i → (i∈ : i ∈ inputs tx) →
        let out = lookupOutput l (outputRef i) (validTxRefs i i∈) (validOutputIndices i i∈)
        in  (address out)# ≡ validator i#
```

Fig. 2. Validity conditions of a ledger, encoded as a dependent record.

```
data ValidLedger : Ledger → Set where
  ·          : ValidLedger [ ]
  _⊕_⊣_      : ValidLedger l
             → (tx : Tx)
             → IsValidTx tx l
             → ValidLedger (tx :: l)
```

4.4 Decision Procedure

Intrinsically-typed ledgers are correct-by-construction, but this does not come for free; we now need to provide substantial proofs of validity alongside every submitted transaction.

To make the proof process more ergonomic for the user of the framework, we prove that all involved propositions appearing in the *IsValidTx* record are *decidable*, thus defining a decision procedure for closed formulas that do not contain any free variable. This process is commonly referred to as *proof-by-reflection* [Bertot and Castéran 2013, Chapter 16] and has been used for proof automation both in Coq [Gonthier et al. 2013] and Agda [Van Der Walt and Swierstra 2012].

Most operations already come with a decidable counterpart, e.g. _ < _ can be decided by _ <?_ that exists in Agda's standard library. Therefore, what we are essentially doing is copying the initial propositions and replace such operators with their decision procedures. Decidability is captured by the *Dec* datatype, ensuring that we can answer a yes/no question over the enclosed proposition:

```
data Dec (P : Set) : Set where
  yes : (p  : P)  → Dec P
  no  : (¬p : ¬P) → Dec P
```

Having a proof of decidability means we can replace a proof of proposition *P* with a simple call to *toWitness* { *Q* = *P*? } *tt*, where *P*? is the decidable counterpart of *P*.

```
True : Dec P → Set
True (yes _) = ⊤
True (no _)  = ⊥

toWitness : {Q : Dec P} → True Q → P
toWitness {Q = yes p} _ = p
toWitness {Q = no _}  ()
```

For this to compute though, the decided formula needs to be *closed*, meaning it does not contain any variables. One could even go beyond closed formulas by utilizing Agda's recent *meta-programming* facilities (macros), but this is outside of the scope of this thesis.

But what about universal quantification? We certainly know that it is not possible to decide on an arbitrary quantified proposition. Hopefully, all our uses of the ∀ operator later constrain the quantified argument to be an element of a list. Therefore, we can define a specific decidable variant of this format:

```
∀?  : (xs : List A)
    → {P : (x : A) (x∈ : x ∈ xs) → Set}
    → (∀ x → (x∈ : x ∈ xs) → Dec (P x x∈))
    → Dec (∀ x x∈ → P x x∈)
∀? []       P?  = yes λ _ ()
∀? (x :: xs) P? with ∀? xs (λ x′ x∈ → P? x′ (there x∈))
... | no ¬p     = no λ p → ¬p (λ x′ x∈ → p x′ (there x∈))
... | yes p′    with P? x (here refl)
... | no ¬p     = no λ p → ¬p (p x (here refl))
... | yes p     = yes λ { x′ (here refl) → p
                        ; x′ (there x∈) → p′ x′ x∈ }
```

Finally, we are ready to provide a decision procedure for each validity condition using the aforementioned operator for quantification and the decidable counterparts for the standard operators we use. Below we give an example for the *validOutputRefs* condition:

```
validOutputRefs? : ∀ (tx : Tx) (l : Ledger)
  → Dec (∀ i → i ∈ inputs tx → outputRef i ∈ unspentOutputs l)
validOutputRefs? tx l = ∀? (inputs tx) λ i _ → outputRef i ∈? unspentOutputs l
```

In Section 4.9 we give an example construction of a valid ledger and demonstrate that our decision procedure discharges all proof obligations with calls to *toWitness*.

4.5 Weakening Lemma

We have defined everything with respect to a fixed set of available addresses, but it would make sense to be able to include additional addresses without losing the validity of the ledger constructed thus far.

In order to do so, we introduce the notion of *weakening* the address space; the only necessary ingredient is a *hash-preserving injection*[8] from a smaller address space $\mathbb{A}$ to a larger address space $\mathbb{B}$:

```
module Weakening
  (𝔸 : Set) (_#ᵃ : HashFunction 𝔸) (_≟ᵃ_ : Decidable {A = 𝔸} _≡_)
  (𝔹 : Set) (_#ᵇ : HashFunction 𝔹) (_≟ᵇ_ : Decidable {A = 𝔹} _≡_)
  (A↪B : 𝔸 , _#ᵃ ↪ 𝔹 , _#ᵇ)
  where
```

[8] Preserving hashes means an injection *f* satisfies $\forall\ \{a\} \to a\#^{a} \equiv (a\ \langle\$\rangle\ f)\#^{b}$, where we denote transporting via an injection with the binary operator $\_\ \langle\$\rangle\ \_$ and injections with the mixfix operator $\mathbb{A}\ ,\ \_\#^{a} \hookrightarrow \mathbb{B}\ ,\ \_\#^{b}$.

```
import      UTxO.Validity 𝔸 _#ᵃ _≟ᵃ _ as A
open import UTxO.Validity 𝔹 _#ᵇ _≟ᵇ _ as B

weakenTxOutput : A.TxOutput → B.TxOutput
weakenTxOutput out = out { address = A ↪ B ⟨$⟩ address out}

weakenTx : A.Tx → B.Tx
weakenTx tx = tx { outputs = map weakenTxOutput (outputs tx) }

weakenLedger : A.Ledger → B.Ledger
weakenLedger = map weakenTx
```

Notice also that the only place where weakening takes place are transaction outputs, since all other components do not depend on the available address space.

With the weakening properly defined, we can finally prove the *weakening lemma* for the available address space:

```
weakening : ∀ { tx : A.Tx} { l : A.Ledger}
  → A.IsValidTx tx l
    ──────────────────────────────────
  → B.IsValidTx (weakenTx tx) (weakenLedger l)
weakening = . . .
```

The weakening lemma states that the validity of a transaction with respect to a ledger is preserved if we choose to weaken the available address space, which we estimate to be useful when we later prove more intricate properties of the extended UTxO model.

One practical use-case for weakening is moving from a bit representation of addresses to one with more available bits (e.g. 32-bit to 64-bit conversion). This, of course, preserves hashes since the numeric equivalent of the converted addresses will be the same. For instance, as we come closer to the quantum computing age, addresses will have to transition to other encryption schemes involving many more bits[9]. Since we allow the flexibility for arbitrary injective functions, our weakening result will hopefully prove resilient to such scenarios.

4.6 Combining

Ideally, one would wish for a modular reasoning process, akin to how *separation logic* in concurrency is used for reasoning about program memory. In our case, we would be able to examine different ledgers of unrelated (i.e. "separate") transactions in a compositional manner. This has to be done carefully, since we need to preserve the proof of validity when combining two ledgers l and l' .

[9] It is believed that even 2048-bit keys will become vulnerable to rapid decryption from quantum computers.

First of all, the ledgers should not share any transactions with each other: *Disjoint l l′* . Secondly, the resulting ledger *l″* will be some interleaving of these two: *Interleaving l l′ l″*. These conditions are actually sufficient to preserve all validity conditions, except *allInputsValidate*. The issue arises from the dependence of validation results on the current state of the ledger, which is given as argument to each validation script. To remedy this, we further require that the new state, corresponding to a particular interleaving, does not break previous validation results:

```
PreserveValidations : (l : Ledger) (l″ : Ledger) → Interleaving l _ l″ → Set
PreserveValidations l₀ _ inter =
  ∀ tx → (p : tx ∈ l₀) →
    let l  =∈−tail p
        l″ =∈−tail (interleave ⊆ inter p)
    in ∀ {ptx i out vds} → runValidation ptx i out vds (getState l″)
                         ≡ runValidation ptx i out vds (getState l)
```

Putting all conditions together, we are now ready to formulate a *combining* operation for valid ledgers:

```
_ ↔ _ ⊣ _ : ∀ {l l′ l″ : Ledger}
   → ValidLedger l
   → ValidLedger l′
   → Σ[ i ∈ Interleaving l l′ l″ ]
   × Disjoint l l′
   × PreserveValidations l l″ i
   × PreserveValidations l′ l″ (swap i)
     ──────────────────────────────
   → ValidLedger l″
```

The proof inductively proves validity of each transaction in the interleaved ledger, essentially reusing the validity proofs of the ledger constituents.

It is important to note what weakening and combining can be interleaved: if we wish to combine ledgers that use different addresses, we can now just apply weakening first and then combine in a type-safe manner.

4.7 Extension I: Data Scripts

The *dataScript* field in transaction outputs does not appear in the original abstract UTxO model [Zahnentferner 2018a], but is available in the extended version of the UTxO model used in the Cardano blockchain [IOHK 2019a]. This addition raises the expressive level of UTxO-based transaction, since it is now possible to simulate stateful behaviour, passing around state in the data scripts (i.e. *Data* = *State*).

This technique is successfully employed in *Marlowe*, a DSL for financial contracts that compiles down to eUTxO transactions [Seijas and Thompson 2018]. Marlowe is accompanied by a simple small-step semantics, i.e. a state transition system. Using data scripts, compilation is rather straightforward since we can pass around the state of the semantics in the data scripts.

4.8 Extension II: Multi-currency

Many major blockchain systems today support the creation of secondary cryptocurrencies, which are independent of the main currency. In Bitcoin, for instance, *colored coins* allow transactions to assign additional meaning to their outputs (e.g. each coin could correspond to a real-world asset, such as company shares) [Rosenfeld 2012].

This approach, however, has the disadvantage of larger transactions and less efficient processing. One could instead bake the multi-currency feature into the base system, mitigating the need for larger transactions and slow processing. Building on the abstract UTxO model, there are current research efforts on a general framework that provides mechanisms to establish and enforce monetary policies for multiple currencies [Zahnentferner 2019].

Fortunately, the extensions proposed by the multi-currency are orthogonal to the formalization so far. In order to accommodate built-in support for user-defined currencies, we need to generalize the type of *Value* from quantities ($\mathbb{N}$) to maps from *currency identifiers* to quantities.

Thankfully, the value operations used in our validity conditions could be lifted to any *commutative group*[10]. Hence, refactoring the validity conditions consists of merely replacing numeric addition with a point-wise addition on maps $\_ +^{c} \_$.

At the user-level, we define these value maps as a simple list of key-value pairs:

```
Value = List (Hash × ℕ)
```

Note that currency identifiers are not strings, but script hashes. We will justify this decision when we talk about the way *monetary policies* are enforced; each currency comes with a certain scheme of allowing or refusing forging of new funds.

We also provide the adding operation, internally using proper maps implemented on AVL trees[11]:

```
open import Data.AVL ℕ-strictTotalOrder

CurrencyMap = Tree (MkValue (λ _ → Hash) (subst (λ _ → ℕ)))

_ +ᶜ _ : Value → Value → Value
c +ᶜ c′ = toList (foldl go (fromList c) c′ )
  where
    go : CurrencyMap → (ℕ × ℕ) → CurrencyMap
    go cur (currency , value) = insertWith currency ((_+ value) ∘ fromMaybe 0) cur
```

[10] Actually, we only ever *add* values, but inverses could be used to *reduce* a currency supply.

[11] https://github.com/agda/agda-stdlib/blob/master/src/Data/AVL.agda

```
sumᶜ : List Value → Value
sumᶜ = foldl _+ᶜ_ []
```

While the multi-currency paper defines a new type of transaction *CurrencyTx* for creating funds, we follow a more lightweight approach, as currently employed in the Cardano blockchain [IOHK 2019b]. This proposal mitigates the need for a new type of transaction and a global registry via a clever use of validator scripts: monetary policies reside in the validator script of the transactional inputs and currency identifiers are just the hashes of those scripts. When one needs to forge a particular currency, two transactions must be submitted: the first only carrying the monetary policy in its output and the second consuming it and forging the desired quantity.

In order to ascertain that forging transactions always follow this scheme, we need to extend our validity record with yet another condition:

```
record IsValidTx (tx : Tx) (l : Ledger) : Set where
  ...
  forging :
    ∀ c → c ∈ keys (forge tx) →
      ∃[ i ] ∃λ (i∈ : i ∈ inputs tx) →
        let out = lookupOutput l (outputRef i) (validTxRefs i i∈) (validOutputIndices i i∈)
        in (address out)# ≡ c
```

The rest of the conditions are the same, modulo the replacement of $\_+\_$ with $\_+^{c}\_$ and *sum* with *sum*ᶜ.

This is actually the first and only validation condition to contain an existential quantification, which poses some issues with our decision procedure for validity. To tackle this, we follow a similar approach to the treatment of universal quantification in Section 4.4:

```
∃?   :  (xs : List A)
     → {P : (x : A) (x∈ : x ∈ xs) → Set ℓ′ }
     → (∀ x → (x∈ : x ∈ xs) → Dec (P x x∈))
     → Dec (∃[x]∃λ (x∈ : x ∈ xs) → P x x∈)
∃? []   P?             = no λ { (x , () , p) }
∃? (x :: xs) P?        with P? x (here refl)
... | yes kp           = yes (x , here refl , p)
... | no¬p             with ∃? xs (λ x′ x∈ → P? x′ (there x∈))
... | yes (x′ , x∈ , p) = yes (x′ , there x∈ , p)
... | no¬pp            = no λ { (x′ , here refl , p) → ¬p p
                              ; (x′ , there x∈ , p) → ¬pp (x′ , x∈ , p) }
```

Now it is straightforward to give a proof of decidability for *forging*:

```
forging? : ∀ (tx : Tx) (l : Ledger)
  → (v₁ : ∀ i → i ∈ inputs tx → Any (λ t → t♯ ≡ id (outputRef i)) l)
  → (v₂ : ∀ i → (i∈ : i ∈ inputs tx) →
        index (outputRef i) < length (outputs (lookupTx l (outputRef i) (v₁ i i∈))))
  → Dec (∀ c → c ∈ keys (forge tx) →
            ∃[ i ] ∃λ (i∈ : i ∈ inputs tx) →
              let out = lookupOutput l (outputRef i) (v₁ i i∈) (v₂ i i∈)
              in (address out)♯ ≡ c)
forging? tx l v₁ v₂ =
  ∀? (keys (forge tx)) λ c _ →
    ∃? (inputs tx) λ i i∈ →
      let out = lookupOutput l (outputRef i) (v₁ i i∈) (v₂ i i∈)
      in (address out)♯ ≟ c
```

4.9 Example: UTxO Ledger

To showcase how we can use our model to construct *correct-by-construction* ledgers, let us revisit the example ledger presented in the Chimeric Ledgers paper [Zahnentferner 2018b].

Any blockchain can be visually represented as a *directed acyclic graph* (DAG), with transactions as nodes and input-output pairs as edges, as shown in Figure 3. The six transactions $t_1 \ \ldots \ t_6$ are self-explanatory, each containing a forge and fee value. The are three participants, represented by addresses $\mathbb{A}$, $\mathbb{B}$ and $\mathbb{C}$, as well as an dedicated address ₿ for the monetary policy of ₿. Notice the special transaction c_0, which enforces the monetary policy of currency ₿ in its outputs (colored in green); the two forging transactions t_1 and t_4 consume these outputs as requested by the validity condition for *forging*. Lastly, there is a single unspent output (coloured in red), namely the single output of t_6: this means at the current state address $\mathbb{C}$ holds ₿ 999.

First, we need to set things up by declaring the list of available addresses and opening our module with this parameter. For brevity, we identify addresses as hashes:

```
Address : Set
Address = ℕ

𝔸 , 𝔹 , ℂ : Address
𝔸 = 1 -- first address
𝔹 = 2 -- second address
ℂ = 3 -- third address

open import UTxO Address (λ x → x) _≟_
```

It is also convenient to define some smart constructors up-front:

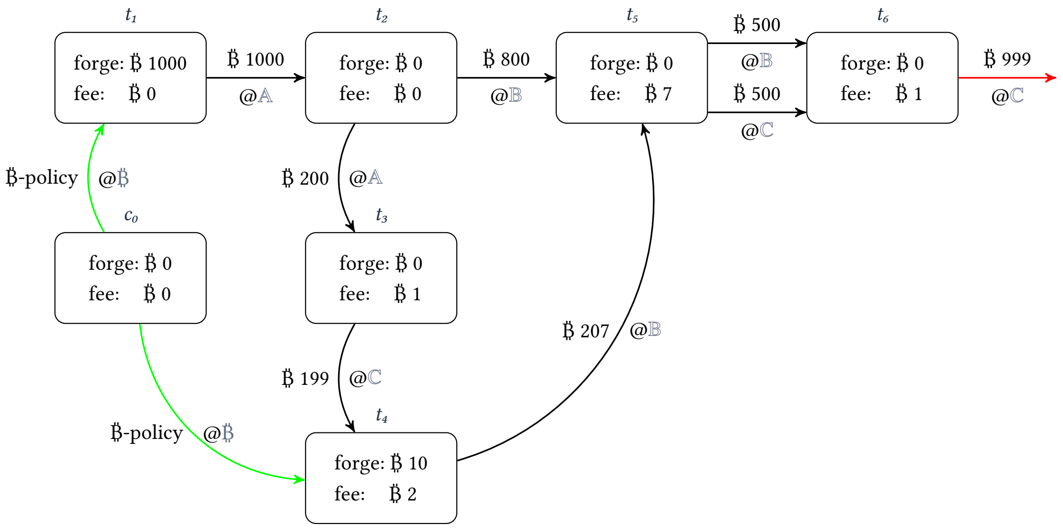

Fig. 3. Example ledger with six transactions

```
₿-validator : State → . . . → Bool
₿-validator (record { height = h }) _ _ _ _ = (h ≡ᵇ 1) ∨ (h ≡ᵇ 4)

mkValidator : TxOutputRef → (State → Value → PendingTx → (ℕ × ℕ) → ℕ → Bool)
mkValidator tin _ _ _ tin′ _ = (id tin ≡ᵇ proj₁ tin′ ) ∧ (index tin ≡ᵇ proj₂ tin′ )

₿ _ : ℕ → Value
₿ v = [(₿-validator# , v)]

withScripts : TxOutputRef → TxInput
withScripts tin = record { outputRef = tin
                         ; redeemer  = λ _ → id tin , index tin
                         ; validator = mkValidator tin}

withPolicy : TxOutputRef → TxInput
withPolicy tin = record { outputRef = tin
                        ; redeemer = λ _ → tt
                        ; validator = ₿-validator}

_ @ _ : Value → Address → TxOutput
v @ addr = record { value = v; address = addr; dataScript = λ _ → tt}
```

₿-validator models a monetary policy that allows forging only at ledger height 1 and 4; *mkValidator* is a script that only validates against the given output reference; ₿ _ creates singleton currency maps for our currency BIT; *withScripts* and *withPolicy* wrap an output reference with the appropriate scripts; _ @ _ creates outputs that do not utilize the data script.

We can then proceed to define the individual transactions defined in Figure 3; the first sub-index of each variable refers to the order the transaction are submitted, while the second sub-index refers to which output of the given transaction we select:

```
c₀ , t₁ , t₂ , t₃ , t₄ , t₅ , t₆ : Tx
c₀ = record { inputs  = [ ]
            ; outputs = ₿ 0 @ (₿-validator#) :: ₿ 0 @ (₿-validator#) :: [ ]
            ; forge   = ₿ 0
            ; fee     = ₿ 0 }
t₁ = record { inputs  = [ withPolicy c₀₀ ]
            ; outputs = [ ₿ 1000 @ 𝔸 ]
            ; forge   = ₿ 1000
            ; fee     = ₿ 0 }
⋮
t₆ = record { inputs  = withScripts t₅₀ :: withScripts t₅₁ :: [ ]
            ; outputs = [ ₿ 999 @ ℂ ]
            ; forge   = ₿ 0
            ; fee     = ₿ 1 }
```

In order for terms involving the *postulated* hash function _# to compute, we use Agda's experimental feature for user-supplied *rewrite rules*:

```
{-# OPTIONS --rewriting #-}
postulate
  eq₁₀ : (mkValidator t₁₀)# ≡ 𝔸
  ⋮
  eq₆₀ : (mkValidator t₆₀)# ≡ ℂ

{-# BUILTIN REWRITE _≡_ #-}
{-# REWRITE eq₀ , eq₁₀ , . . . , eq₆₀ #-}
```

Below we give a correct-by-construction ledger containing all transactions:

```
ex-ledger : ValidLedger (t₆ :: t₅ :: t₄ :: t₃ :: t₂ :: t₁ :: c₀ :: [ ])
ex-ledger =
  · c₀ ⊣ record { . . . }
```

```
⊕ t₁ ⊣ record { validTxRefs          = toWitness {Q = validTxRefs? t₁ l₀} tt
                ; validOutputIndices   = toWitness {Q = validOutputIndices? ...} tt
                ; validOutputRefs      = toWitness {Q = validOutputRef? ...} tt
                ; validDataScriptTypes = toWitness {Q = validDataScriptTypes? ...} tt
                ; preservesValues      = toWitness {Q = preservesValues? ...} tt
                ; noDoubleSpending     = toWitness {Q = noDoubleSpending? ...} tt
                ; allInputsValidate    = toWitness {Q = allInputsValidate? ...} tt
                ; validateValidHashes  = toWitness {Q = validateValidHashes? ...} tt
                ; forging              = toWitness {Q = forging? ...} tt}
⊕ t₂ ⊣ record {...}
⋮
⊕ t₆ ⊣ record {...}
```

First, it is trivial to verify that the only unspent transaction output of our ledger is the output of the last transaction t_6, as demonstrated below:

```
utxo : list (unspentOutputs ex-ledger) ≡ [t₆₀]
utxo = refl
```

Most importantly, notice that no manual proving is necessary, since our decision procedure discharges all validity proofs. In the next release of Agda, it will be possible to even omit the manual calls to the decision procedure (via *toWitness*), by declaring the proof of validity as an implicit *tactic argument*[12].

This machinery allows us to define a compile-time macro for each validity condition that works on the corresponding goal type, and *statically* calls the decision procedure of this condition to extract a proof and fill the required implicit argument. As an example, we give a sketch of the macro for the *validTxRefs* condition below:

```
pattern vtx i i∈ tx t l =
  `λ i : `TxInput ⇒
    `λ i∈ : #0 ` ∈ (`inputs t) ⇒
      `Any (`λ tx ⇒ #0 `# ` ≡ `id `outputRef #2) l

macro
  validTxRefsM : Term → TC ⊤
  validTxRefsM hole = do
    goal ← inferType hole
    case goal of λ
      { (vtx _ _ _ t l) →
        t′ ← unquoteTC t
```

[12] https://agda.readthedocs.io/en/latest/language/implicit-arguments.html#tactic-arguments

```
      l′ ← unquoteTC l
      case validTxRefs? t′ l′ of λ
        { (yes p) → quoteTC p >>= unify hole
        ; (no _)  → typeError [ strErr "validity condition does not hold" ]
        }
  ; t → typeError [ strErr "wrong type of goal" ]
  }
```

We first define a pattern to capture the validity condition in AST form; Agda provides a *reflection* mechanism[13], that defines Agda's language constructs as regular Agda datatypes. Note the use of *quoted* expressions in the definition of the *vtx* pattern, which also uses De Bruijn indices for variables bound in λ-abstractions.

Then, we define the macro as a *metaprogram* running in the type-checking monad *TC*. After pattern matching on the goal type and making sure it has the expected form, we run the decision procedure, in this case *validTxRefs?*. If the computation reached a positive answer, we automatically fill the required term with the proof of validity carried by the *yes* constructor. In case the transaction is not valid, we report a compile-time error.

We can now replace the operator for appending (valid) transactions to a ledger, with one that uses *implicit* tactic arguments instead:

```
_⊕_ : ValidLedger l
    → (tx : Tx)
    → {@(tactic validTxRefsM) : ∀ i → i ∈ inputs tx → Any (λ t → t♯ ≡ id (outputRef i)) l
    → ...
    → ValidLedger (tx :: l)
(l ⊕ tx) {vtx} ... = l ⊕ tx ⊣ record {validTxRefs = vtx , ...}
```

[13] https://github.com/agda/agda/blob/master/src/data/lib/prim/Agda/Builtin/Reflection.agda

SECTION 5

Formal Model II: BitML Calculus

Now let us shift our focus to our second subject of study, the BitML calculus for modelling smart contracts. In this subsection we sketch the formalized part of BitML we have covered so far, namely the syntax and small-step semantics of BitML contracts, its game-theoretic symbolic model, as well as an example execution of a contract under these semantics. All code is publicly available on Github[14].

First, we begin with some basic definitions that will be used throughout this section. Instead of giving a fixed datatype of participants, we parametrise our module with a given *abstract data type* of participants that we can check for equality, as well as non-empty list of honest participants:

```
module BitML (Participant : Set) (_ ≟ₚ _ : Decidable {A = Participant} _≡_)
             (Honest : List⁺ Participant)
             where

Time : Set
Time = ℕ

Value : Set
Value = ℕ

Secret : Set
Secret = String

record Deposit : Set where
  constructor _has_
  field participant : Participant
        value : Value
```

[14]https://github.com/omelkonian/formal-bitml

Representation of time and monetary values is again simplistic, both modelled as natural numbers. while we model participant secrets as simple strings[15]. A deposit consists of the participant that owns it and the number of bitcoins it carries.

We, furthermore, introduce a simplistic language of logical predicates and arithmetic expressions with the usual constructs (e.g. numerical addition, logical conjunction) and give the usual semantics (predicates on booleans and arithmetic on naturals). A more unusual feature of these expressions is the ability to calculate length of secrets (within arithmetic expressions) and, in order to ensure more type safety later on, all expressions are indexed by the secrets they internally use.

```
data Arith : List Secret → Set where
  `_ : ℕ → Arith [ ]
  `len : (s : Secret) → Arith [ s ]
  _`+_ : Arith sₗ → Arith sᵣ → Arith (sₗ ++ sᵣ)
  _`-_ : Arith sₗ → Arith sᵣ → Arith (sₗ ++ sᵣ)

ℕ⟦_⟧ : ∀ {s} → Arith s → ℕ
ℕ⟦_⟧ = ...

data Predicate : List Secret → Set where
  `True : Predicate [ ]
  _`∧_ : Predicate sₗ → Predicate sᵣ → Predicate (sₗ ++ sᵣ)
  `¬_ : ∀ {s} → Predicate s → Predicate s
  _`≡_ : Arith sₗ → Arith sᵣ → Predicate (sₗ ++ sᵣ)
  _`<_ : Arith sₗ → Arith sᵣ → Predicate (sₗ ++ sᵣ)

𝔹⟦_⟧ : ∀ {s} → Predicate s → Bool
𝔹⟦_⟧ = ...
```

5.1 Contracts in BitML

A *contract advertisement* consists of a set of *preconditions*, which require some resources from the involved participants prior to the contract's execution, and a *contract*, which specifies the rules according to which bitcoins are transferred between participants.

Preconditions either require participants to have a deposit of a certain value on their name (volatile or not) or commit to a certain secret. A *persistent* deposit has to be provided before the contract is stipulated, while a *volatile* deposit may be needed dynamically during the execution of

[15] Of course, one could provide more realistic types (e.g. support for multiple currencies or words of specific length) to be closer to the implementation, as shown for the UTxO model in Section 4.

the contract. Both volatile and persistent deposits required by a precondition are captured in its two type-level indices, respectively:

```
data Precondition : List Value → List Value → Set where
  -- volatile deposit
  _ ? _ : Participant → (v : Value) → Precondition [v] []
  -- persistent deposit
  _ ! _ : Participant → (v : Value) → Precondition [] [v]
  -- committed secret
  _ # _ : Participant → Secret → Precondition [] []
  -- conjunction
  _ ∧ _  :  Precondition vsᵛ vsᵖ → Precondition vsᵛ′ vsᵖ′
        → Precondition (vsᵛ ++ vsᵛ′ ) (vsᵖ ++ vsᵖ′ )
```

Moving on to actual contracts, we define them by means of a collection of five types of commands; *put* injects participant deposits and revealed secrets in the remaining contract, *withdraw* transfers the current funds to a participant, *split* distributes the current funds across different individual contracts, _ : _ requires the authorization from a participant to proceed and *after* _ : _ allows further execution of the contract only after some time has passed.

```
data Contract  :  Value       -- the monetary value it carries
               → List Value -- the volatile deposits it presumes
               → Set where
  -- collect deposits and secrets
  put _ reveal _ if _ ⇒ _ ⊣ _ : ∀ {s′  : List Secret} {
    → (vs : List Value) → (s : List Secret) → Predicate s′  → Contract (v + sum vs) vs′
    → s′  ⊆ s
    → Contract v (vs′  ++ vs)
  -- transfer the remaining balance to a participant
  withdraw : ∀ {v vs} → Participant → Contract v vs
  -- split the balance across different branches
  split : ∀ {vs}
    → (cs : List (∃[v] Contract v vs))
    → Contract (sum (proj₁ ⟨$⟩ cs)) vs
  -- wait for participant's authorization
  _ : _ : Participant → Contract v vs → Contract v vs
  -- wait until some time passes
  after _ : _ : Time → Contract v vs → Contract v vs
```

There is a lot of type-level manipulation across all constructors, since we need to make sure that indices are calculated properly. For instance, the total value in a contract constructed by the *split* command is the sum of the values carried by each branch. The *put* command[16] additionally requires an explicit proof that the predicate of the if part only uses secrets revealed by the same command.

We also introduce an intuitive syntax for declaring the different branches of a *split* command, emphasizing the *linear* nature of the contract's total monetary value:

```
_ ⊸ _ : ∀ {vs} → (v : Value) → Contract v vs → ∃[v] Contract v vs
v ⊸ c = v , c
```

Having defined both preconditions and contracts, we arrive at the definition of a contract advertisement:

```
record Advertisement (v : Value) (vsᶜ vsᵛ vsᵖ : List Value) : Set where
  constructor _⟨_⟩∶-_
  field G     : Precondition vsᵛ vsᵖ
        C     : Contracts v vsᶜ
        valid : length vsᶜ ⩽ length vsᵛ
              × participantsᵍ G ++ participantsᶜ C ⊆ (participant ⟨$⟩ persistentDeposits G)
              × v ≡ sum vsᵖ
```

Notice that in order to construct an advertisement, one has to also provide proof of the contract's validity with respect to the given preconditions, namely that all deposit references in the contract are declared in the precondition and each involved participant is required to have a persistent deposit.

To clarify things so far, let us see a simple example of a contract advertisement. We first open the *BitML* module with a trivial datatype for participants, consisting of *A* and *B*:

```
data Participant : Set where
  A B : Participant

_≟_ : Decidable {A = Participant} _≡_
⋮

Honest : Σ[ps ∈ List Participant] (length ps > 0)
Honest = [A] , ≤-refl

open BitML Participant _≟_ [A]⁺
```

16 *put* comprises of several components and we will omit those that do not contain any helpful information, e.g. write *put* _ ⇒ _ when there are no revealed secrets and the predicate trivially holds.

We then define an advertisement, whose type already says a lot about what is going on; it carries ₿ 5, presumes the existence of at least one deposit of ₿ 200, and requires both participants to pay a persistent deposit beforehand.

```
ex-ad : Advertisement 5 [200] [200] [3 , 2]
ex-ad = ⟨ B ? 200 ∧ B ! 3 ∧ A ! 2 ⟩
          split ( 2 ⊸ withdraw B
                ⊕ 2 ⊸ after 42 : withdraw A
                ⊕ 1 ⊸ put [200] ⇒ B : withdraw {201} A ⊣ . . .
                )
          ⊣ . . .
```

Looking at the precondition itself, we see that the required deposit will be provided by *A*. The contract first splits the bitcoins across three branches: the first one gives ₿ 2 to *B*, the second one gives ₿ 2 to *A* after some time period, while the third one retrieves *B*'s deposit of ₿ 200 and allows *B* to authorize the withdrawal of the remaining funds (currently ₿ 201) from *A*.

We have omitted the proofs that ascertain the well-formedness of the *put* command and the advertisement, as they are straightforward and do not provide any more intuition[17].

5.2 Small-step Semantics

BitML is a *process calculus*, which is geared specifically towards smart contracts. Contrary to most process calculi that provide primitive operators for inter-process communication via message-passing [Hoare 1978], the BitML calculus does not provide such built-in features.

It, instead, provides domain-specific synchronization mechanisms through its *small-step* reduction semantics. These essentially define a *labelled transition system* between *configurations*, where *action* labels are emitted on every transition and represent the required actions of the participants. This symbolic model consists of two layers; the bottom one transitioning between *untimed* configurations and the top one that works on *timed* configurations.

We start with the datatype of actions, which showcases the principal actions required to satisfy an advertisement's preconditions and an action to pick one branch of a collection of contracts (introduced by the choice operator ⊕). We have omitted uninteresting actions concerning the manipulation of deposits, such as dividing, joining, donating and destroying them. Since we will often need versions of the types of advertisements/contracts with their indices existentially quantified, we first provide aliases for them. For convenience in notation, we will sometimes write ∃*A* to mean this existential packing of the indices of *A*:

```
AdvertisedContracts : Set
AdvertisedContracts = List (∃[v] ∃[vsᶜ] ∃[vsᵛ] ∃[vsᵖ] Advertisement v vsᶜ vsᵛ vsᵖ)

ActiveContracts : Set
ActiveContracts = List (∃[v] ∃[vs] List (Contract v vs))
```

[17] In fact, we have defined decidable procedures for all such proofs using the *proof-by-reflection* pattern [Van Der Walt and Swierstra 2012]. These automatically discharge all proof obligations, when there are no variables involved.

```
data Action (p : Participant) -- the participant that authorizes this action
  :   AdvertisedContracts     -- the contract advertisements it requires
  → ActiveContracts           -- the active contracts it requires
  → List Value                -- the deposits it requires from this participant
  → List Deposit              -- the deposits it produces
  → Set where

  -- join two deposits deposits
  _ ↔ _ : ∀ {vs}
        → (i : Index vs)
        → (j : Index vs)
        → Action p [] [] vs ((p has_) ⟨$⟩ updateAt ((i , vs !! i + vs !! j) :: (j , 0) :: []) vs)
  -- commit secrets to stipulate an advertisement
  ♯▷ _ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
       → Action p [v , vsᶜ , vsᵛ , vsᵖ , ad] [] [] []
  -- spend x to stipulate an advertisement
  _ ▷ˢ _ : (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
         → (i : Index vsᵖ)
         → Action p [v , vsᶜ , vsᵛ , vsᵖ , ad] [] [vsᵖ !! i] []
  -- pick a branch
  _ ▷ᵇ _ : (c : List (Contract v vs))
         → (i : Index c)
         → Action p [] [v , vs , c] [] []
  ⋮
```

The action datatype is parametrised[18] over the participant who performs it and includes several indices representing the prerequisites the current configuration has to satisfy, in order for the action to be considered valid (e.g. one cannot spend a deposit to stipulate an advertisement that does not exist).

The first index refers to advertisements that appear in the current configuration, the second to contracts that have already been stipulated, the third to deposits owned by the participant currently performing the action and the fourth declares new deposits that will be created by the action. For instance, the join operation _ ↔ _ requires a non-empty list of deposits and produces a modification, where the two values at indices i and j are merged in position i.

Although our indexing scheme might seem a bit heavyweight now, it makes many little details and assumptions explicit, which would bite us later on when we will need to reason about them.

[18] In Agda, datatype parameters are similar to indices, but are not allowed to vary across constructors.

Continuing from our previous example advertisement, let's see an example action where A spends the required ₿ 100 to stipulate the example contract[19]:

```
ex-spend : Action A [5 , [200] , [200] , [100] , ex-ad] [] [100] []
ex-spend = ex-ad ▷ˢ 0 SF
```

The 0 *SF* is not a mere natural number, but inhibits *Fin* (*length* vs^p), which ensures we can only construct actions that spend valid persistent deposits.

BitML's small-step semantics is a state transition system, whose states we call *configurations*. These are built from advertisements, active contracts, deposits, action authorizations and committed/revealed secrets:

```
data Configuration′ :  --      current       ×       required
                           AdvertisedContracts × AdvertisedContracts
                         → ActiveContracts     × ActiveContracts
                         → List Deposit        × List Deposit
                         → Set where

   -- empty configuration
  ∅ : Configuration′ ([] , []) ([] , []) ([] , [])

   -- contract advertisement
  `_  :  (ad : Advertisement v vsᶜ vsᵛ vsᵖ)
     → Configuration′ ([v , vsᶜ , vsᵛ , vsᵖ , ad] , []) ([] , []) ([] , [])

   -- active contract
  ⟨ _ , _ ⟩ᶜ   :  (c : List (Contract v vs)) → Value
               → Configuration′ ([] , []) ([v , vs , c] , []) ([] , [])

   -- deposit redeemable by a participant
  ⟨ _ , _ ⟩ᵈ   :  (p : Participant) → (v : Value)
                 →    Configuration′ ([] , []) ([] , []) ([p has v] , [])

   -- authorization to perform an action
  _ [_]  :  (p : Participant) → Action p ads cs vs ds
        → Configuration′ ([] , ads) ([] , cs) (ds , ((p has _) ⟨$⟩ vs))

   -- committed secret
  ⟨ _ : _#_ ⟩  :  Participant → Secret → Maybe ℕ
               → Configuration′ ([] , []) ([] , []) ([] , [])

   -- revealed secret
  _ : _#_  :  Participant → Secret → ℕ
           → Configuration′ ([] , []) ([] , []) ([] , [])

   -- parallel composition
  _ | _  :  Configuration′ (ads¹ , rads¹) (cs¹ , rcs¹) (ds¹ , rds¹)
```

[19] Notice that we have to make all indices of the advertisement explicit in the second index in the action's type signature.

$$\begin{aligned}
&\to \mathit{Configuration}'\ (\mathit{ads}^{\mathrm{r}}\ ,\ \mathit{rads}^{\mathrm{r}})\ (\mathit{cs}^{\mathrm{r}}\ ,\ \mathit{rcs}^{\mathrm{r}})\ (\mathit{ds}^{\mathrm{r}}\ ,\ \mathit{rds}^{\mathrm{r}})\\
&\to \mathit{Configuration}'\ (\mathit{ads}^{\mathrm{l}} \mathbin{+\!\!+} \mathit{ads}^{\mathrm{r}}\ ,\ \mathit{rads}^{\mathrm{l}} \mathbin{+\!\!+} (\mathit{rads}^{\mathrm{r}} \setminus \mathit{ads}^{\mathrm{l}}))\\
&\qquad (\mathit{cs}^{\mathrm{l}} \mathbin{+\!\!+} \mathit{cs}^{\mathrm{r}}\ ,\ \mathit{rcs}^{\mathrm{l}} \mathbin{+\!\!+} (\mathit{rcs}^{\mathrm{r}} \setminus \mathit{cs}^{\mathrm{l}}))\\
&\qquad ((\mathit{ds}^{\mathrm{l}} \setminus \mathit{rds}^{\mathrm{r}}) \mathbin{+\!\!+} \mathit{ds}^{\mathrm{r}}\ ,\ \mathit{rds}^{\mathrm{l}} \mathbin{+\!\!+} (\mathit{rds}^{\mathrm{r}} \setminus \mathit{ds}^{\mathrm{l}}))
\end{aligned}$$

The indices are quite involved, since we need to record both the current advertisements, stipulated contracts and deposits and the required ones for the configuration to become valid. The most interesting case is the parallel composition operator, where the resources provided by the left operand might satisfy some requirements of the right operand. Moreover, consumed deposits have to be eliminated as there can be no double spending, while the number of advertisements and contracts always grows.

By composing configurations, we will eventually end up in a *closed* configuration, where all required indices are empty (i.e. the configuration is self-contained):

$$\begin{aligned}
&\mathit{Configuration} : \mathit{AdvertisedContracts} \to \mathit{ActiveContracts} \to \mathit{List\ Deposit} \to \mathit{Set}\\
&\mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds} = \mathit{Configuration}'\ (\mathit{ads}\ ,[\,])\ (\mathit{cs}\ ,[\,])\ (\mathit{ds}\ ,[\,])
\end{aligned}$$

We are now ready to declare the inference rules of the bottom layer of our small-step semantics, by defining an inductive datatype modelling the binary step relation between untimed configurations:

$$\begin{aligned}
&\mathbf{data}\ \_\longrightarrow\_ : \mathit{Configuration}\ \mathit{ads}\ \mathit{cs}\ \mathit{ds} \to \mathit{Configuration}\ \mathit{ads}'\ \mathit{cs}'\ \mathit{ds}' \to \mathit{Set}\ \mathbf{where}\\
&\quad \mathit{DEP\text{-}AuthJoin} :\\
&\qquad \langle\, A\ ,\ v\,\rangle^{\mathrm{d}} \mid \langle\, A\ ,\ v'\,\rangle^{\mathrm{d}} \mid \Gamma \longrightarrow \langle\, A\ ,\ v\,\rangle^{\mathrm{d}} \mid \langle\, A\ ,\ v'\,\rangle^{\mathrm{d}} \mid A\ [0 \leftrightarrow 1] \mid \Gamma\\
\\
&\quad \mathit{DEP\text{-}Join} :\\
&\qquad \langle\, A\ ,\ v\,\rangle^{\mathrm{d}} \mid \langle\, A\ ,\ v'\,\rangle^{\mathrm{d}} \mid A\ [0 \leftrightarrow 1] \mid \Gamma \longrightarrow \langle\, A\ ,\ v + v'\,\rangle^{\mathrm{d}} \mid \Gamma\\
\\
&\quad \mathit{C\text{-}Advertise} : \forall\ \{\Gamma\ \mathit{ad}\}\\
&\qquad \to \exists[\,p \in \mathit{participants}^{\mathrm{g}}\ (G\ \mathit{ad})\,]\ p \in \mathit{Hon}\\
&\qquad \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\\
&\qquad \to \Gamma \longrightarrow \text{`}\mathit{ad} \mid \Gamma\\
\\
&\quad \mathit{C\text{-}AuthCommit} : \forall\ \{A\ \mathit{ad}\ \Gamma\}\\
&\qquad \to \mathit{secrets}\ (G\ \mathit{ad}) \equiv a_0\ \ldots\ a_n\\
&\qquad \to (A \in \mathit{Hon} \to \forall\ [\,i \in 0\ \ldots\ n\,]\ a_i \not\equiv \bot)\\
&\qquad \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\\
&\qquad \to \text{`}\mathit{ad} \mid \Gamma \longrightarrow \text{`}\mathit{ad} \mid \Gamma \mid \ldots \langle\, A : a_i \# N_i\,\rangle \ldots \mid A\ [\sharp\!\triangleright\ \mathit{ad}]\\
\\
&\quad \mathit{C\text{-}Control} : \forall\ \{\Gamma\ C\ i\ D\}\\
&\qquad \to C\ !!\ i \equiv A_1 : A_2 : \ldots : A_n : D
\end{aligned}$$

```
    ──────────────────────────────────────────────
  → ⟨ C , v ⟩ᶜ | . . .  Aᵢ [C ▷ᵇ i]  . . . | Γ ⟶ ⟨ D , v ⟩ᶜ | Γ
⋮
```

There is a total of 18 rules we need to define, but we choose to depict only a representative subset of them. For a detailed overview of all the rules, we refer the reader to the original BitML paper [Bartoletti and Zunino 2018], as well as to the source code of the BitML compiler [20].

The first pair of rules initially appends the authorization to merge two deposits to the current configuration (rule *DEP-AuthJoin*) and then performs the actual join (rule *DEP-Join*). This is a common pattern across all rules, where we first collect authorizations for an action by all involved participants, and then we fire a subsequent rule to perform this action. *C-Advertise* advertises a new contract, mandating that at least one of the participants involved in the pre-condition is honest and requiring that all deposits needed for stipulation are available in the surrounding context. *C-AuthCommit* allows participants to commit to the secrets required by the contract's pre-condition, but only dishonest ones can commit to the invalid length ⊥. Lastly, *C-Control* allows participants to give their authorization required by a particular branch out of the current choices present in the contract, discarding any time constraints along the way.

It is noteworthy to mention that during the transcriptions of the complete set of rules from the paper [Bartoletti and Zunino 2018] to our dependently-typed setting, we discovered some discrepancies or over-complications, which we document extensively in Section 5.6.

The inference rules above have elided any treatment of time constraints; this is handled by the top layer, whose states are now timed configurations. The only interesting inference rule is the one that handles time decorations of the form *after* _ : _, since all other cases are dispatched to the bottom layer (which just ignores timing aspects).

```
record Configurationᵗ (ads : AdvertisedContracts)
                      (cs : ActiveContracts)
                      (ds : Deposits) : Set where
  constructor _ @ _
  field cfg  : Configuration ads cs ds
        time : Time

data _ ⟶ₜ _ : Configurationᵗ ads cs ds → Configurationᵗ ads′ cs′ ds′  → Set where
  Action : ∀ {Γ Γ′ t}
    → Γ ⟶ Γ′
    ────────────────
    → Γ @ t ⟶ₜ Γ′ @ t
  Delay : ∀ {Γ t δ}
    ────────────────────
```

[20] https://github.com/bitml-lang/bitml-compiler

```
    → Γ  @  t ⟶ₜ Γ  @  (t + δ)

  Timeout : ∀ {Γ Γ′ t i contract}
    → All (_ ≤ t) (timeDecorations (contract !! i)) -- all time constraints are satisfied
    → ⟨ [contract !! i] , v ⟩ᶜ | Γ ⟶ Γ′              -- resulting state if we pick this branch
      ───────────────────────────────────────────────
    → (⟨ contract , v ⟩ᶜ | Γ)  @  t ⟶ₜ Γ′  @  t
```

5.3 Reasoning Modulo Permutation

In the definitions above, we have assumed that $(\_ | \_ , \varnothing)$ forms a *commutative monoid*, which allowed us to always present the required sub-configuration individually on the far left of a composite configuration. While such definitions enjoy a striking similarity to the ones appearing in the original paper [Bartoletti and Zunino 2018] (and should always be preferred in an informal textual setting), this approach does not suffice for a mechanized account of the model. After all, this explicit treatment of all intuitive assumptions/details is what makes our approach robust and will lead to a deeper understanding of how these systems behave. To overcome this intricacy, we introduce an *equivalence relation* on configurations, which holds when they are just permutations of one another:

```
  _≈_ : Configuration ads cs ds → Configuration ads cs ds → Set
  c ≈ c′  = cfgToList c ↭ cfgToList c′
    where
      open import Data.List.Permutation using (_↭_)

      cfgToList : Configuration′ p₁ p₂ p₃ → List (∃[p₁] ∃[p₂] ∃[p₃] Configuration′ p₁ p₂ p₃)
      cfgToList ∅                    = []
      cfgToList (l | r)              = cfgToList l ++ cfgToList r
      cfgToList {p₁} {p₂} {p₃} c = [p₁ , p₂ , p₃ , c]
```

Given this reordering mechanism, we now need to generalize all our inference rules to implicitly reorder the current and next configuration of the step relation. We achieve this by introducing a new variable for each of the operands of the resulting step relations, replacing the operands with these variables and requiring that they are re-orderings of the previous configurations, as shown in the following generalization of the *DEP-AuthJoin* rule[21]:

```
  DEP-AuthJoin :
        Γ′  ≈ ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | Γ
            ∈ Configuration ads cs (A has v :: A has v′ :: ds)
      → Γ″  ≈ ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | A [0 ↔ 1] | Γ
            ∈ Configuration ads cs (A has (v + v′) :: ds)
```

[21] In fact, it is not necessary to reorder both ends for the step relation; at least one would be adequate.

$$\frac{}{\to \Gamma' \longrightarrow \Gamma''}$$

Unfortunately, we now have more proof obligations of the re-ordering relation lying around, which makes reasoning about our semantics rather tedious. We are currently investigating different techniques to model such reasoning up to equivalence:

- *Quotient types* [Altenkirch et al. 2011] allow equipping a type with an equivalence relation. If we assume the axiom that two elements of the underlying type are *propositionally* equal when they are equivalent, we could discharge our current proof burden trivially by reflexivity. Unfortunately, while one can easily define *setoids* in Agda, there is not enough support from the underlying type system to make reasoning about such an equivalence as easy as with built-in equality.
- Going a step further into more advanced notions of equality, we arrive at *homotopy type theory* [Univalent Foundations Program 2013], which tries to bridge the gap between reasoning about isomorphic objects in informal pen-paper proofs and the way we achieve this in mechanized formal methods. Again, realizing practical systems with such an enriched theory is a topic of current research [Cohen et al. 2016] and no mature implementation exists yet, so we cannot integrate it with our current development in any pragmatic way.
- The crucial problems we have encountered so far are attributed to the non-deterministic nature of BitML, which is actually inherent in any process calculus. Building upon this idea, we plan to take a step back and investigate different reasoning techniques for a minimal process calculus. Once we have an approach that is more suitable, we will incorporate it in our full-blown BitML calculus. Current efforts are available on Github[22].

For the time being, the complexity that arises from having the permutation proofs in the premises of each and every one of the 18 rules, poses a significant burden to our development. As a quick workaround, we can factor out the permutation relation in the *reflexive transitive closure* of the step relation, which will eventually constitute our custom syntax for proving derivations, inspired by equational reasoning:

```
data _ ⟶* _ : Configuration ads cs ds → Configuration ads′ cs′ ds′ → Set where

  _ □ :

        (M : Configuration ads cs ds)
        ─────────────────
     → M ⟶* M

  _ ⟶⟨ _ ⟩_ : ∀ {L′ M M′ N}
     → (L : Configuration ads cs ds)
     → L′ ⟶ M′
```

[22] https://github.com/omelkonian/formal-process-calculus

$$\begin{array}{l}
\to M \longrightarrow^{*} N \\
\to \{\_ : L \approx L' \times M \approx M' \} \\
\hline
\to L \longrightarrow^{*} N
\end{array}$$

$$\begin{array}{l}
\mathit{begin}\ \_ : \forall\ \{M\ N\} \\
\quad \to M \longrightarrow^{*} N \\
\hline
\quad \to M \longrightarrow^{*} N \\
\mathit{begin}\ x = x
\end{array}$$

The permutation relation is actually decidable, so we can always discharge the implicitly required proof, similarly to the techniques described in Section 4.9.

5.4 Example: Timed-commitment Protocol

We are finally ready to see a more intuitive example of the *timed-commitment protocol*, where a participant commits to revealing a valid secret a (e.g. ”qwerty”) to another participant, but loses her deposit of ₿ 1 if she does not meet a certain deadline t:

$$\begin{array}{l}
tc : \mathit{Advertisement}\ 1\ [\,]\ [\,]\ (1 :: 0 :: [\,]) \\
tc = \langle\ A\ !\ 1 \land A\ \#\ a \land B\ !\ 0\ \rangle\ \mathit{reveal}\ [\,a\,] \Rightarrow \mathit{withdraw}\ A \dashv \ldots\ \oplus\ \mathit{after}\ t : \mathit{withdraw}\ B
\end{array}$$

Below is one possible reduction in the bottom layer of our small-step semantics, demonstrating the case where the participant actually meets the deadline:

$$\begin{array}{l}
\mathit{tc\text{-}semantics} : \langle\ A\ ,\ 1\ \rangle^{\mathrm{d}} \longrightarrow^{*} \langle\ A\ ,\ 1\ \rangle^{\mathrm{d}} \mid A : a\ \#\ 6 \\
\mathit{tc\text{-}semantics} = \\
\quad \mathit{begin} \\
\qquad \langle\ A\ ,\ 1\ \rangle^{\mathrm{d}} \\
\quad \longrightarrow\langle\ \mathit{C\text{-}Advertise}\ \ldots\ \rangle \\
\qquad `tc \mid \langle\ A\ ,\ 1\ \rangle^{\mathrm{d}} \\
\quad \longrightarrow\langle\ \mathit{C\text{-}AuthCommit}\ \ldots\ \rangle \\
\qquad `tc \mid \langle\ A\ ,\ 1\ \rangle^{\mathrm{d}} \mid \langle\ A : a\ \#\ 6\ \rangle \mid A\ [\,\sharp \rhd\ tc\,] \\
\quad \longrightarrow\langle\ \mathit{C\text{-}AuthInit}\ \ldots\ \rangle \\
\qquad `tc \mid \langle\ A\ ,\ 1\ \rangle^{\mathrm{d}} \mid \langle\ A : a\ \#\ 6\ \rangle \mid A\ [\,\sharp \rhd\ tc\,] \mid A\ [\,tc \rhd^{s}\ 0\,] \\
\quad \longrightarrow\langle\ \mathit{C\text{-}Init}\ \ldots\ \rangle \\
\qquad \langle\ tc\ ,\ 1\ \rangle^{\mathrm{c}} \mid \langle\ A : a\ \#\ \mathit{inj}_1\ 6\ \rangle \\
\quad \longrightarrow\langle\ \mathit{C\text{-}AuthRev}\ \ldots\ \rangle \\
\qquad \langle\ tc\ ,\ 1\ \rangle^{\mathrm{c}} \mid A : a\ \#\ 6 \\
\quad \longrightarrow\langle\ \mathit{C\text{-}Control}\ \ldots\ \rangle \\
\qquad \langle\ [\,\mathit{reveal}\ [\,a\,] \Rightarrow \mathit{withdraw}\ A \dashv \ldots\,]\ ,\ 1\ \rangle^{\mathrm{c}} \mid A : a\ \#\ 6
\end{array}$$

```
  ⟶⟨ C-PutRev ... ⟩
    ⟨ [withdraw A] , 1 ⟩^c | A : a # 6
  ⟶⟨ C-Withdraw ... ⟩
    ⟨ A , 1 ⟩^d | A : a # 6
  □
```

At first, *A* holds a deposit of ₿ 1, as required by the contract's precondition. Then, the contract is advertised and the participants slowly provide the corresponding prerequisites (i.e. *A* commits to a secret via *C-AuthCommit* and spends the required deposit via *C-AuthInit*, while *B* does not do anything). After all pre-conditions have been satisfied, the contract is stipulated (rule *C-Init*) and the secret is successfully revealed (rule *C-AuthRev*). Finally, the first branch is picked (rule *C-Control*) and *A* retrieves her deposit back (rules *C-PutRev* and *C-Withdraw*).

We chose to omit the proofs required at the application of each inference rules (replaced with ... above), since these are tedious and mostly uninteresting. Moreover, we plan to develop decision procedures for these proofs[23] to automate this part of the proof development process.

5.5 Symbolic Model

The approach taken by BitML defines two models that describe participant interaction; the *symbolic model* works on the abstract level of BitML contracts, while the *computational model* is defined at the level of concrete Bitcoin transactions.

In order to formalize the BitML's symbolic model, we first notice that a constructed derivation witnesses one of many possible contract executions. In other words, derivations of our small-step semantics model *traces* of the contract execution. Our symbolic model will provide a game-theoretic view over those traces, where each participant has a certain *strategy* that selects moves depending on the current trace of previous moves. Moves here should be understood just as emissions of a label, i.e. application of a certain inference rule.

5.5.1 Labelled Step Relation

To that end, we associate a label to each inference rule and extend the original step relation to additionally emit labels, hence defining a *labelled transition system*.

We first define the set of labels, which basically distinguish which rule was used, along with all (non-proof) arguments that are required by the rule:

```
data Label : Set where

  auth-join [_ , _ ↔ _] : Participant → DepositIndex → DepositIndex → Label
  join [_ ↔ _] :                        DepositIndex → DepositIndex → Label

  advertise [_] : ∃Advertisement → Label

  auth-commit [_ , _ , _] : Participant → ∃Advertisement → List CommittedSecret → Label
```

[23] Most proofs of decidability are in the Agda standard library already, but there is still a lot of "plumbing" to be done.

```
auth-init [_,_,_] : Participant → ∃Advertisement → DepositIndex → Label
init [_] : ∃Advertisement → Label

auth-control [_,_▷ᵇ_] : Participant → (c : ∃Contracts) → Index (proj₂ (proj₂ c)) → Label
control : Label
⋮
delay [_] : Time → Label
```

Notice how we existentially pack indexed types, so that *Label* remains simply-typed. This is essential, as it would be tedious to manipulate indices when there is no need for them. Moreover, some indices are now just ℕ instead of *Fin*, losing the guarantee to remain well-scoped.

The step relation will now emit the corresponding label for each rule. Below, we give the updated kind signature and an example for the *DEP-AuthJoin* rule:

```
data _ —→⟦_⟧_ : Configuration ads cs ds
              → Label
              → Configuration ads′ cs′ ds′
              → Set where
  ⋮
  DEP-AuthJoin :
     ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | Γ
   —→⟦ auth-join [ A , 0 ↔ 1 ] ⟧
     ⟨ A , v ⟩ᵈ | ⟨ A , v′ ⟩ᵈ | A [ 0 ↔ 1 ] | Γ
  ⋮
```

Naturally, the reflexive transitive closure of the augmented step relation will now hold a sequence of labels as well:

```
data _ —→*⟦_⟧_ : Configuration ads cs ds
               → List Label
               → Configuration ads′ cs′ ds′
               → Set where

  _ ∎  :
        (M : Configuration ads cs ds)
        ─────────────────
      → M —→*⟦ [] ⟧ M

  _ —→⟨_⟩_ : ∀ {L′ M M′ N}
```

```
    → (L : Configuration ads cs ds)
    → L′  —→⟦ a ⟧ M′
    → M —→*⟦ as ⟧ N
    → { _ : L ≈ L′  × M ≈ M′ }
      ─────────────────────
    → L —→*⟦ a :: as ⟧ N

begin _ : ∀ {M N}
  → M —→*⟦ as ⟧ N
    ────────────
  → M —→*⟦ as ⟧ N
begin x = x
```

The timed variants of the step relation follow exactly the same procedure, so we do not repeat the definitions here.

5.5.2 Traces

Values of type _ —→*⟦ _ ⟧ _ model execution traces. Since the complex type indices of the step-relation datatype is not as useful here, we define a simpler datatype of execution traces that is a list of labelled transitions between (existentially-packed) timed configurations:

```
data Trace : Set where
  _∙          : ∃TimedConfiguration → Trace

  _ ::⟦ _ ⟧ _ : ∃TimedConfiguration → Label → Trace → Trace
```

Stripping. Strategies will make moves based on these traces, so we need a *stripping* operation that traverses a configuration with its emitted labels and removes any sensitive information (i.e. committed secrets):

```
stripCfg : Configuration′ p₁ p₂ p₃ → Configuration′ p₁ p₂ p₃
stripCfg⟨ p : a # _ ⟩ = ⟨ p : a # nothing ⟩
stripCfg (l | r ⊣ p) =   stripCfg l | stripCfg r ⊣ p
stripCfg c           =   c

stripLabel : Label → Label
stripLabel auth-commit [p , ad , _] = auth-commit [p , ad , []]
stripLabel a = a

_ ∗ : Trace → Trace
(. . . , Γ  @  t) ∗          = (. . . , stripCfg Γ  @  t)
(. . . , Γ  @  t) ::⟦ α ⟧ ts = (. . . , stripCfg Γ  @  t) ::⟦ stripLabel α ⟧ (ts ∗)
```

5.5.3 Strategies

Participant strategies are functions which, given the (stripped) trace so far, pick a set of possible next moves for its participant. These moves cannot be arbitrary; they have to satisfy several validity conditions which we require as proof in the datatype definition itself.

Strategies are expected to be PPTIME algorithms, so as to have a certain computational bound that guarantees secrets are sufficiently hard to guess by adversaries, etc. While recent research suggests that it is possible to track complexity bounds in the type system [Danielsson 2008], working on a resource-aware logic would make this much more difficult in search of tooling and infrastructure, thus we ignore this requirement and simply model strategies as regular functions.

Before we define the types of strategies, we give a convenient notation to extend a trace with another (timed) transition, which essentially projects the last timed configuration out of a trace and relates it to the second operand:

```
_ ↣⟦ _ ⟧ _ : Trace → Label → ∃TimedConfiguration → Set
R ↣⟦ α ⟧ (_ , _ , _ , tc′ )
  = proj₂ (proj₂ (proj₂ (lastCfg R))) ⟶⟦ α ⟧ tc′
```

Honest strategies. Each honest participant is modelled by a symbolic strategy that outputs a set of possible next moves with respect to the current trace. These moves have to be *valid*, thus we define *honest strategies* as a dependent record:

```
record HonestStrategy (A : Participant) : Set where
  field
    strategy : Trace → List Label
    valid    : A ∈ Hon                                              (1)
             × (∀ R α → α ∈ strategy (R ∗) →                        (2)
                 ∃[R′ ] (R ↣⟦ α ⟧ R′ ))
             × (∀ R α → α ∈ strategy (R ∗) →                        (3)
                 Allₘ (_ ≡ A) (authDecoration α))
             × (∀ R Δ Δ′ ad →                                       (4)
                 auth-commit [A , ad , Δ] ∈ strategy (R ∗) →
                 auth-commit [A , ad , Δ′ ] ∈ strategy (R ∗) →
                   Δ ≡ Δ′ )
             × (∀ R T α → α ∈ strategy (R ∗) →                      (5)
                 ∃[α′ ] (R ↣⟦ α′ ⟧ T ) →
                 ∃[R″] (T ∷⟦ α ⟧ R ↣⟦ α ⟧ R″) →
                   α ∈ strategy ((T ∷⟦ α ⟧ R) ∗))
```

Condition (1) restricts our participants to the honest subset[24] and condition (2) requires that chosen moves are in accordance to the small-step semantics of BitML. Condition (3) states that one cannot authorize moves for other participants, condition (4) requires that the lengths of committed secrets are *coherent* (i.e. no different lengths for the same secrets across moves) and condition (5) dictates that decisions are *consistent*, such that moves that are not chosen will still be selected by the strategy in a future run (if they remain valid).

All honest participants should be accompanied by such a strategy, so we pack all honest strategies in one single datatype:

```
HonestStrategies : Set
HonestStrategies = ∀ {A} → A ∈ Hon → HonestStrategy A
```

Adversary strategies. All dishonest participant will be modelled by a single adversary *Adv*, whose strategy now additionally takes the moves chosen by the honest participants and makes the final decision.

Naturally, the chosen move is subject to certain conditions and is again a dependent record:

```
record AdversarialStrategy (Adv : Participant) : Set where
  field
    strategy : Trace → List (Participant × List Label) → Label

    valid    : Adv ∉ Hon                                                    (1)
             × (∀ {B ad Δ} → B ∉ Hon → α ≡ auth-commit [B , ad , Δ] →       (2)
                 α ≡ strategy (R ∗) [])
             × ∀ {R : Trace} {moves : List (Participant × List Label)} →    (3)
                 let α = strategy (R ∗) moves in
                 (   ∃[A]
                       ( A ∈ Hon
                       × authDecoration α ≡ just A
                       × α ∈ concatMap proj₂ moves)
                 ⊎ (   authDecoration α ≡ nothing
                   × (∀ δ → α ≢ delay [δ])
                   × ∃[R′ ] (R ⟿⟦ α ⟧ R′ ))
                 ⊎ (∃[B]
                      (   (authDecoration α ≡ just B)
                       × (B ∉ Hon)
                       × (∀ s → α ≢ auth-rev [B , s])
                       × ∃[R′ ] (R ⟿⟦ α ⟧ R′ )))
                 ⊎ ∃[δ]
                      ( (α ≡ delay [δ])
```

[24] Recall that *Hon* is non-empty, i.e. there is always at least one honest participant.

```
        × All (λ {(_ , Λ) → (Λ ≡ [])
                              ⊎ Any (λ {delay [ δ′ ] → δ′ ≥δ; _ → ⊥}) Λ}) moves)
    ⊎ ∃[ B ]∃[ s ]
        ( α ≡ auth-rev [ B , s ]
        × B ∉ Hon
        × ⟨ B : s#nothing ⟩ ∈ (R ∗)
        × ∃[ R∗′ ]∃[ Δ ]∃[ ad ]
            ( R∗′ ∈ prefixTraces (R ∗)
            × strategy R∗′ [] ≡ auth-commit [ B , ad , Δ ]
            × (s , nothing) ∈ Δ)))
```

The first two conditions state that the adversary is not one of the honest participants and that committing cannot depend on the honest moves, respectively. Condition (3) constraints the move that is chosen by the adversary, such that one of the following conditions hold:

(1) The move was chosen out of the available honest moves.
(2) It is not a *delay*, nor does it require any authorization.
(3) It is authorized by a dishonest participant, but is not a secret-revealing move.
(4) It is a *delay*, but one that does not influence the time constraints of the honest participants.
(5) It reveals a secret from a dishonest participant, in which case there is valid commit (i.e. with non-$\bot$ length) somewhere in the previous trace.

A complete set of strategies includes a strategy for each honest participant and a single adversarial strategy:

```
Strategies : Set
Strategies = AdversarialStrategy × HonestStrategies
```

We can now describe how to proceed execution on the current trace, namely by retrieving possible moves from all honest participants and giving control to the adversary to make the final choice for a label:

```
runAdversary : Strategies → Trace → Label
runAdversary (S† , S) R = strategy S† (R ∗) (runHonestAll (R ∗) S)
  where
    runHonestAll : Trace → List (Participant × List Label) → HonestMoves
    runHonestAll R S = mapWith ∈ Hon (λ {A} A ∈→ A , strategy (S A ∈) (R ∗))
```

Symbolic Conformance. Given a trace, we can formulate a notion of *conformance* of a trace with respect to a set of strategies, namely when we transitioned from an initial configuration to the current trace using only moves obtained by those strategies:

```
data _-conforms-to-_ : Trace → Strategies → Set where

  base : ∀ {Γ : Configuration ads cs ds} {SS : Strategies}
    → Initial Γ
      ────────────────────────────────────────
    → (ads , cs , ds , Γ @ 0)∙ -conforms-to- SS

  step : ∀ {R : Trace} {T′ : ∃TimedConfiguration} {SS : Strategies}
    → R-conforms-to-SS
    → R —→⟦ runAdversary SS R ⟧ T′
      ───────────────────────────────────────────────
    → (T′ ∷⟦ runAdversary SS R ⟧ R) -conforms-to- SS
```

5.5.4 *Meta-theoretical results*

To increase confidence in our symbolic model, we proceed with the mechanization of two meta-theoretical lemmas.

Stripping preserves semantics. The first one concerns the operation of stripping sensitive values out of a trace. If we exclude moves that reveal or commit secrets (i.e. rules *AuthRefv* and *AuthCommit*), we can formally prove that stripping preserves the small-step semantics:

```
∗ – preserves-semantics :
  (∀ A s   → α ≢ auth-rev [ A , s ]) →
  (∀ A ad Δ → α ≢ auth-commit [ A , ad , Δ ])
  → (∀ T′   → R —→⟦ α ⟧ T′
               ─────────────────
            → R ∗ —→⟦ α ⟧ T′ ∗)
  × (∀ T′   → R ∗ —→⟦ α ⟧ T
               ──────────────────────────────
            → ∃[ T″ ] (R —→⟦ α ⟧ T″) × (T′ ∗ ≡ T″ ∗)
```

The second part of the conclusion states that if we have a transition from a stripped state, then there is an equivalent target state (modulo additional sensitive information) to which the unstripped state can transition.

Adversarial moves are always semantic. Lastly, it holds that all moves that can be chosen by the adversary are admitted by the small-step semantics:

```
adversarial-move-is-semantic :
  ∃[ T′ ] (R —→⟦ runAdversary (S† , S) R ⟧ T′ )
```

The proofs have not been completely formalized yet, since there are a lot of cases to cover and our "over-indexing" approach has proven difficult to work with. More specifically, as our type indices get increasingly complicated, we get a lot of proof obligations at the usage sites of the indexed datatypes, where the Agda compiler will encounter complicated equalities during normalization (e.g. $ys \setminus ([\,] \setminus ys) \equiv ys$), which cannot be automatically solved. In these cases, we need to always rewrite the goal manually until it reaches a point where statements become trivial.

A possible way of tackling this issue is factoring complex index dependencies out of datatype constructors and requiring them as additional *explicit* proof arguments. For example, instead of accumulating secrets in pre-condition expressions, we could do the following:

```
-- before
_‘ + _ : Arith sₗ → Arith sᵣ → Arith (sₗ ++ sᵣ)

-- after
_‘ + _ ⊣ _ : Arith sₗ → Arith sᵣ → s ≡ sₗ ++ sᵣ → Arith s
```

That way, we can get a hold of these proof requirements explicitly, instead of implicitly guiding the Agda compiler through rewriting.

In retrospect, it might be worthwhile to take a step back and simplify indices across the whole development. One such simplification would be to remove secrets as indices of expressions in contract pre-conditions, but this would mean type-safety has to be sacrificed in the typing of *put* commands. Another approach would be to follow the original BitML formulation and identify resources with string identifiers, instead of the DeBruijn encoding we followed throughout our work (via the use of *Fin* numbers). However, we do not recommend totally abandoning type-safety, but rather move to a string-based representation where you extrinsically ensure that deposits in configurations are well-scoped.

5.6 BitML Paper Fixes

It is expected in any mechanization of a substantial amount of theoretical work to encounter inconsistencies in the pen-and-paper version, ranging from simple typographical mistakes and omissions to fundamental design problems. This is certainly one of the primary selling points for formal verification; corner cases that are difficult to find by testing or similar methods, can instead be discovered with rigorous formal methods.

Our formal development was no exception, since we encountered several issues with the original presentation, which led to the modifications presented below.

Inference Rules. Rule *DEP-Join* requires two symmetric invocations of the *DEP-AuthJoin* rule, but it is unclear if this gives us anything meaningful. Instead, we choose to simplify the rule by requiring just one authorization.

When rule *C-AuthRev* is presented in the original BitML paper, it seems to act on an atomic configuration $\langle\, A : \alpha\#\mathbb{N}\,\rangle$. This renders the rule useless in any practical scenario, so we extend the rule to include a surrounding context:

```
⟨ A : s # just n ⟩ | Γ ⟶⟦ auth-rev [ A , s ] ⟧ A : s # n | Γ
```

Small-step Derivations as Equational Reasoning. In Section 5.4, we saw an example derivation of our small-step semantics, given in an equational-reasoning style. This is possible, because the involved rules follow a certain format.

Alas, rule *C-Control* includes another transition in its premises which results in the same state Γ' as the transition in the conclusion, resulting in a tree-like proof structure. which is arguably inconvenient for textual presentation. This is problematic when we try to reason in an equational-reasoning style using our multi-step relation $\_ \longrightarrow^* \_$, since this branching will break our sequential way of presenting the proof step by step.

To avoid this issue, notice how we can "linearize" the proof structure by removing the premise and replacing the target configuration of the conclusion with the source configuration of the removed premise. Our version of *C-Control* in Section 5.2 reflects this important refactoring.

Conditions for Adversarial Strategies. Moves chosen by an adversarial strategy come in two forms: labels and pairs (A , j) of an honest participant A with an index into his/her current moves. However, this is unnecessary, since we can both cases uniformly using our *Label* type.

Semantics-preserving Stripping. The meta-theoretical lemma concerning stripping in the original paper (*Lemma 3*) requires that the transition considered is not an application of the *Auth-Rev* rule. It turns out this is not a strong enough guarantee, since the *AuthCommit* rule also contains sensitive information, thus would not be preserved after stripping. We, therefore, fix the statement in Lemma 3 to additionally require that $\alpha \not\equiv A : \langle\, G \,\rangle C \,,\, \Delta$.

SECTION 6

Related Work

6.1 Static Analysis Tools

Bugs in smart contracts have led to significant financial losses (c.f. DAO attack), thus it is crucial we can automatically detect them. Moreover, we must detect them statically, since contracts become immutable once deployed. This is exceptionally hard though, due to the concurrent execution inherent in smart contracts, which is why most efforts so far have been on static analysis techniques for particular classes of bugs.

MadMax. In Ethereum smart contracts, programs written or compiled to EVM bytecode, hold a valuable resource called *gas* (c.f. Section 2.4). This amount puts a threshold on the number of computational steps a contract can execute until it completes. Out-of-gas errors can lead to undefined behaviour, that can be exploited by a malicious attacker.

MadMax is a scalable program analysis tool, that aims to statically detect such gas-related vulnerabilities with very high precision [Grech et al. 2018]. The techniques employed include *control-flow analysis* and declarative logic programs that form queries about the program structure.

Effectively Callback Free (ECF) Analysis. A lot of security issues in Ethereum arise from the use of callback functions in smart contracts. This abstraction poses a great deal of complexity on understanding contract behaviour, since they break modular reasoning.

In Grossman et al. [2017], a class of *effectively callback-free* (ECF) programs is defined, where such issues are not possible. Then, a program analysis tool is provided to verify such a property, which can additionally be realized either statically or dynamically.

Verifying Liquidity in BitML Contracts. The BitML compiler that accompanies the original paper [Bartoletti and Zunino 2018], written in Racket[25], also provides a *model checker* to verify *liquidity* of contracts written in its DSL; liquid contracts never freeze funds, i.e. making them irredeemable by any participant[26].

The crucial observation that makes verification possible, is that liquidity is a decidable property. Model-checking is possible in a finite state space, derived from a finite variant of BitML's infinite semantics.

[25] https://github.com/bitml-lang/bitml-compiler

[26] An example vulnerability occurred in the Ethereum Parity Wallet, which froze ~160M USD.

6.2 Type-driven Approaches

Recently there has been increased demand for more rigid formal methods in the blockchain domain [Miller et al. 2018] and we believe the field would greatly benefit from a language-based, type-driven approach [Sheard et al. 2010] alongside a mechanized meta-theory.

Scilla. One such example is Scilla, an intermediate language for smart contracts, with a formal semantics based on communicating automata [Sergey et al. 2018]. Scilla, however, follows an *extrinsic* approach to software verification: contracts are written in a simply-typed DSL embedded in Coq [Barras et al. 1997] and dependent types are used to verify their safety and temporal properties.

On the other hand, our work explores a new point in the design space, exploiting the dependent type system of Agda [Norell 2008] to encode *intrinsically*-typed contracts, whose behaviour is more predictable and easier to reason about. Nonetheless, this comes with the price of tedious type-level manipulation, as witnessed throughout our formal development. Intricate datatype indices, in particular, are notoriously difficult to get right and refactor in an iterative fashion.

Setzer's Bitcoin Model. A formal model, which is very similar to our own formal model of UTxO-based ledgers, is Setzer's effort to model Bitcoin in Agda [Setzer 2018].

There, Setzer utilizes an extended form of Agda's unique feature of *induction-recursion*; the types of transactions and ledgers are mutually, inductively defined and, at the same time, the set of unspent transaction outputs is recursively computed. This mitigates the need to carry proofs that ascertain all lookups succeed and references have valid targets.

Alas, these advanced techniques create a significant gap between the pen-and-paper mathematical formulation and the corresponding mechanized model. This is the primary reason we chose to have a simpler treatment of the basic types, treating the well-scopedness of lookups extrinsically (i.e. within the *IsValidTx* dependent record). Another reason for being skeptical to such a statically-defined model is the difficulty to later extend it with dynamic operations, such as continuous change of the participant set.

SECTION 7

Future Work

In this section, we describe possible next steps for both our formalizations, namely the (extended) UTxO model and the BitML calculus.

The majority of the suggestions are straightforward or completely orthogonal to our current system, therefore we believe they can be incorporated in a relatively short-term period.

Most importantly, we give the ambitious vision of integrating our two objects of study, giving rise to a *certified compiler* from BitML contracts to UTxO transactions; this will constitute a major part of the author's upcoming PhD studies.

7.1 Extended UTxO

7.1.1 *Non-fungible Tokens*

Although we have implemented and formalized support for user-issued cryptocurrencies, the multi-currency infrastacture in the current development of Cardano also supports *non-fungible tokens* (NFTs) [IOHK 2019a]. These tokens represent unique assets that are not interchangable (i.e. fungible) and have already be used in crypto-gaming, where in-game assets are controlled by the player instead of the game developer.

In order to accommodate NFTs, a very similar extension to the one employed for the initial support for multiple currencies is needed. Specifically, again we have to generalize the *Value* type from *single-level* maps from currency identifiers to quantities, to *two-level* maps that introduce an intermediate level of *tokens*. In other words, a currency can hold items of a distinct identity (token), which can in turn have a certain amount of supply (quantity).

As expected, the necessary refactoring is simple:

- Generalize from *Map Hash* $\mathbb{N}$ to *Map Hash* (*Map Token* $\mathbb{N}$).
- Lift algebraic operations to the new representation point-wise, just like we did to initially support multiple currencies.

An interesting side-effect of this way of implementing NFTs is the ability to investigate a whole spectrum between fungible and non-fungible token currencies, e.g. when having more than one distinct tokens.

7.1.2 *Plutus Integration*

In our current formalization of the extended UTxO model, scripts are immediately modelled by their denotations (i.e. pure mathematical functions). This is not accurate, however, since scripts are actually pieces of program text. However, there is current development by James Chapman of IOHK to formalize the meta-theory of Plutus, Cardano's scripting language[27].

Since we mostly care about Plutus as a scripting language, it would be possible to replace the denotations with actual Plutus Core source code and utilize the formalized meta-theory to acquire the denotational semantics when needed. Arguably, this has certain benefits, such as providing decidable equality for our scripts[28] and, consequently, decidable equality for whole transactions and ledgers.

7.1.3 Multi-signature Scheme

Another extension we deem worthy of being included in our eUTxO formalization, is the recent proposal to support *multi-signature* transactions [Corduan and Güdemann 2019]. This extension introduces a new validation scheme for transactions, where an unspent output can only be consumed if a pre-defined set of digital signatures from different participants is provided.

The main changes involve adding a new witness type to the transactions, namely the set of required signatures. Then, the validation mechanism enforces a check between the pre-defined set of signatures and the required ones. A slight increase in the complexity of our formal framework is necessary, but we, nevertheless, expect this extension to be more-or-less orthogonal to the existing features.

7.2 BitML

7.2.1 Decision Procedures

The current proof development process of our BitML formal model is far from user-friendly; the user has to supply inline proofs in copious amounts while using our dependently-typed definitions. Thankfully, most of these can be proven decidable once and for all, and then a simple call to the decision procedure would do the work.

As shown in Section 4.9, we could use Agda's latest feature for *tactic arguments* to mitigate the need for the user to provide any proofs, e.g. when writing contracts or small-step derivations.

7.2.2 Towards Completeness

Continuing our work on the formalization of the BitML paper [Bartoletti and Zunino 2018], there is still a lot of theoretical ground to be covered:

- While we currently have the symbolic model and its meta-theory in place, there are still various holes in the proofs; nothing major, but it is always a good idea to cover all corner cases. Most of these holes correspond to insignificant proof obligations stemming from our heavy indexing scheme, such as the list equalities arising from the uses of the composition operator _ | _ for BitML configurations. However, a few remaining holes are not as trivial and should be investigated for further confidence in the model, such as covering all possible cases in the meta-theoretical proofs of BitML's symbolic model in Section 5.5.4.

[27] https://github.com/input-output-hk/plutus-metatheory

[28] Of course, two arbitrary Agda functions cannot be checked for equality.

- Another import task is to define the computational model; a counterpart of the symbolic model augmented with pragmatic computational properties to more closely resemble the low-level details of Bitcoin.
- When both symbolic and computational strategies have been formalized, we will be able to finally prove the correctness of the BitML compiler, which translates high-level BitML contracts to low-level standard Bitcoin transactions. The symbolic model concerns the input of the compiler, while the computational one concerns the output. This endeavour will involve implementing the actual translation and proving *coherence* between the symbolic and the computational model. Proving coherence essentially requires providing a (weak) *simulation* between the two models; each step in the symbolic part is matched by (multiple) steps in the computational one.

7.3 UTxO-BitML Integration

So far we have investigated the two models under study separately, but it would be interesting to see whether these can be intertwined in some way.

First, note that it is entirely possible to simulate the compilation scheme given in [Bartoletti and Zunino 2018] with our eUTxO model, but now compiling to a more abstract notion of UTxO transactions, rather than *standard* Bitcoin transactions. Nonetheless, we believe this would be overly complicated for our purposes, since the extensions our eUTxO model supports can make things much simpler. For instance, *data scripts* make it possible to simulate *stateful*, *on-chain* computation. This is ideal for implementing a small-step interpreter, since our reduction semantics is defined as a (labelled) transition system itself. In fact, this has already been successfully employed by *Marlowe*, whose implementation of the small-step semantics of its financial contracts follows exactly this stateful scheme via data scripts[29].

One could argue that the original BitML-to-Bitcoin compiler is less useful than a compiler to our eUTxO formal model, due to the latter being more abstract without consideration for ad-hoc features of Bitcoin, thus more amendable to easier reasoning and generally more flexible. Therefore, it might be worthwhile to skip the formalization of BitML's computational model all together, and instead focus on a BitML-to-eUTxO compiler instead.

A significant benefit of compiling down to our intrinsically-typed ledgers, is the guarantee that we only ever get **valid** transactions. Alas, we need to have a similar operational semantics for our eUTxO model to state a *compilation correctness* theorem. Fortunately, IOHK's internal formal methods team already has an up-to-date mathematical specification of small-step semantics for Cardano ledgers [IOHK 2019c], upon which we can rely for a *mechanical* reduction semantics and eventually a *certified compiler*.

Lastly, and it would be beneficial to review the different modelling techniques used across both models, identifying their key strengths and witnesses. With this in mind, we could refactor crucial parts of each model for the sake of elegance, clarity and ease of reasoning.

[29]https://github.com/input-output-hk/marlowe/blob/master/docs/tutorial-v2.0/marlowe-plutus.md

7.4 BitML-Marlowe Comparison

Another possible research endeavour is a formal comparison between the BitML calculus and the Marlowe DSL. In fact, this is already under investigation by the Marlowe team, as recent commits on Github suggest[30].

They both provide a high-level description of smart contracts and they both lend themselves to an operational reduction semantics. Looking at the mere size of BitML's inference rules, Marlowe's small-step semantics seems a lot simpler. Therefore, we believe it would be interesting to investigate whether BitML's formulation can be simplified, possibly taking inspiration by the language constructs of Marlowe.

To this end, a formalization of Marlowe in Agda should be prototyped, followed by a mechanization of its meta-theory. Then, a compilation correctness results would guarantee that any step taken by Marlowe can be simulated by one or more steps in BitML's semantics, essentially leading to a *full abstraction* result; Marlowe exhibits the same behavioural properties as BitML, and we can safely reason in its more abstract framework. If that is indeed the case, effort should be probably better spent on the eUTxO-Marlowe integration, rather than eUTxO-BitML.

7.5 Featherweight Solidity

One of the posed research questions concerns the expressiveness of the extended UTxO model with respect to Ethereum-like account-based ledgers.

Since Solidity is a fully-fledged programming language with lots of features (e.g. static typing, inheritance, libraries, user-defined types), it makes sense to restrict our formal study to a compact subset of Solidity that is easy to reason about. This is the approach also taken in Featherweight Java [Igarashi et al. 2001]; a subset of Java that omits complex features such as reflection, in favour of easier behavioural reasoning and a more formal investigation of its semantics. In a similar vein, we will attempt to formalize a lightweight version of Solidity, which we will refer to as *Featherweight Solidity*. Fortunately, there have already been recent efforts in F* to analyze and verify Ethereum smart contracts, which already describe a simplified model of Solidity [Bhargavan et al. 2016].

As an initial step, one should try out different example contracts in Solidity and check whether they can be transcribed to contracts appropriate for an extended UTxO ledger.

7.6 Proof Automation

Last but not least, our current dependently-typed approach to formalizing our models has led to a significant proof burden, as evidenced by the complicated type signatures presented throughout our formal development. This certainly makes the reasoning process quite tedious and time consuming, so a reasonable task would be to implement automatic proof-search procedures using Agda meta-programming [Kokke and Swierstra 2015]. We have already done so for the validity condition of UTxO ledgers (Section 4.4), but wish to also provide decision procedures for the ledgers weakening and combining, as well as for significant proof obligations in the BitML model.

[30]https://github.com/input-output-hk/marlowe/blob/master/semantics-3.0/BItSem.hs

SECTION 8

Conclusion

Our main contributions are the formalization of two existing mathematical models, namely the abstract model for UTxO-based ledgers and the BitML calculus.

We start with an intrinsically-typed model of UTxO ledgers, as presented in the mathematical formulation in [Zahnentferner 2018a]. Moreover, we further account for two extensions currently used in the Cardano blockchain: *data scripts* to make more expressive transactional schemes possible and *multi-currency* support to allow multiple cryptocurrencies to coexist in the same ledger. Having a formal framework at hand allows us to proceed with a more algebraic treatment of UTxO ledgers, which we demonstrate with the introduction of two operations: *weakening* injects the abstract type of the available address space to a larger type, while *combining* allows merging two disjoint ledgers, automatically extracting a proof of validity for the resulting ledger from the existing proofs of the sub-ledgers. Finally, an example construction of a correct-by-definition ledger is given, where no manual proof is necessary, since all proofs can be discharged with the decision procedure for the validity conditions.

Our BitML formalization closely follows the original BitML paper [Bartoletti and Zunino 2018], but we also impose a lot of type-level invariants, in order to have a more clear view of what constitutes a *well-formed* contract. We carry this type-driven mentality over to the *small-step semantics* of BitML contracts, where the type system keeps our inference rules in check. To showcase that the proposed modelling of the semantics can accommodate ordinary proof methods, we give an example derivation for the standard contract implementing the *timed-commitment protocol*. We do not go as far as mechanizing the proposed translation from BitML contracts to Bitcoin transactions, but rather restrain our formalization efforts to BitML's *symbolic model*: a game-theoretic reasoning framework for participant strategies, accompanied by mechanized meta-theoretical results that gives us more confidence for the model.

Apart from the individual formalization of our two objects of study, we also deem it worthy to investigate on how to combine these two, envisioning a *certified compiler* from BitML contracts to UTxO-based transactions with (extended) scripts. There, any attack scenario that we would discover in BitML's symbolic model, could be safely determined to manifest in the UTxO semantics as well.

While the original BitML paper gives a compiler from BitML contracts to standard Bitcoin transactions, along with a compilation correctness proof, we believe that the abstract UTxO model would be a more suitable target. Not only will the translation be more useful, since we abstract away technicalities specific to Bitcoin and can accommodate other UTxO-based blockchains, but also easier to implement and reason about, since the added expressivity arising from the extensions will make the translation more straightforward.

Moving to a dependently-typed settings gives us a much more in-depth view on how everything works, but there are also a lot of design decisions involved in the choice of datatypes and modelling approaches. Throughout the thesis, we have described and justified our own decisions, possibly driven by the particular advantages and disadvantages of Agda. Nonetheless, there has also been discussion on alternative ways to approach the modelling process, as well as thoughts for future directions.

In addition to the mechanization of existing formulations, we sketch a research plan to ultimately get our hands on a *certified* compiler from BitML contracts to UTxO transactions, where we can reason in the BitML level and safely transfer the results to the actual behaviour in a UTxO-based ledger.

Through our current mechanized results, we hope to have further motivated the use of language-oriented, type-driven solutions to blockchain semantics in general, and the semantics of smart contract behaviour in particular.

References

Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. 2011. Definable quotients in type theory. *Draft paper* (2011), 48–49.

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

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

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

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, A Gollamudi, G Gonthier, N Kobeissi, A Rastogi, T Sibut-Pinote, N Swamy, and S Zanella-Béguelin. 2016. Short paper: Formal verification of smart contracts. In *Proceedings of the 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in conjunction with ACM CCS*. 91–96.

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

Hao Chen, Xiongnan Newman Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward compositional verification of interruptible OS kernels and device drivers. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 431–447.

Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a constructive interpretation of the univalence axiom. *CoRR* abs/1611.02108 (2016). arXiv:1611.02108 http://arxiv.org/abs/1611.02108

Jared Corduan and Matthias Güdemann. 2019. A Formal Specification of a Multi-Signature Scheme using Scripts. Retrieved 7/2019 from https://hydra.iohk.io/build/835279/download/2/multi-sig.pdf

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 133–144.

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.

Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2013. *A small scale reflection extension for the Coq system.* Ph.D. Dissertation. Inria Saclay Ile de France.

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

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

Charles Antony Richard Hoare. 1978. Communicating sequential processes. In *The origin of concurrent programming*. Springer, 413–443.

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.

Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 3 (2001), 396–450.

IOHK. 2019a. The Extended UTxO Model. Retrieved 2/2019 from https://github.com/input-output-hk/plutus/blob/master/docs/extended-utxo/README.md

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

IOHK. 2019c. Small Step Semantics for Cardano. Retrieved 7/2019 from https://hydra.iohk.io/build/902242/download/1/small-step-semantics.pdf

Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388.

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

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

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.

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

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

Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing contracts: an adventure in financial engineering. *ACM SIG-PLAN Notices* 35, 9 (2000), 280–292.

Meni Rosenfeld. 2012. Overview of colored coins. *White paper, bitcoil. co. il* 41 (2012).

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

Pablo Lamela Seijas, Simon J Thompson, and Darryl McAdams. 2016. Scripting smart contracts for distributed ledger technology. *IACR Cryptology ePrint Archive* 2016 (2016), 1156.

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

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

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

Univalent Foundations Program. 2013. *Homotopy type theory: Univalent foundations of mathematics*. Univalent Foundations.

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.

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

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

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

For the sake of brevity and clarity, we have omitted various technical details throughout the presentation of our formal development. We present those which we deem relevant in this Appendix.

SECTION A

Generalized Variables

We use Agda's recent capabilities for *generalized variables*, which allow one to declare variable names of a certain type at the top-level and then omit their binding at the usage sites in type definitions for clarity.

Below we give a complete set of all variables used throughout this thesis:

```
variable
  -- General
  ℓ : Level
  A B C D : Set ℓ

  -- UTxO
  l l′ l″ : Ledger
  tx : Tx
  i : TxInput
  i∈ : i ∈ inputs tx

  -- BitML
  p A B : Participant
  A₁ . . . A_N : Participant
  a₁ . . . a_N : Secret
  N₁ . . . N_N : ℕ ⊎ ⊥
  v v′ : Value
  vs vsᶜ vsᵛ vsᵖ : List Value
  ad : Advertisement v vsᶜ vsᵛ vsᵖ
  contract C D : List (Contract v vs)
  i : Index C

  ads ads′ ads″ rads adsʳ radsʳ adsˡ radsˡ : AdvertisedContracts
```

$cs\ \ cs'\ \ cs''\, rcs\ cs^{\mathrm{r}}\ \ rcs^{\mathrm{r}}\ \ cs^{\mathrm{l}}\ \ rcs^{\mathrm{l}} : \mathit{ActiveContracts}$

$ds\ ds'\ \ ds''\, rds\ ds^{\mathrm{r}}\ \ rds^{\mathrm{r}}\ \ ds^{\mathrm{l}}\ \ rds^{\mathrm{l}} : \mathit{Deposits}$

$\Gamma\ \mathit{\Gamma_0}\ L\ M\ N : \mathit{Configuration}\ ads\ cs\ ds$

$\Gamma'\ \ : \mathit{Configuration}\ ads'\ \ cs'\ \ ds'$

$\Gamma''\ : \mathit{Configuration}\ ads''\, cs''\, ds''$

$t\ \delta : \mathit{Time}$

$\alpha : \mathit{Label}$

$\alpha s : \mathit{List\ Label}$

$R\ R'\ \ R''\ : \mathit{Trace}$

$T\ T'\ \ T''\ : \exists \mathit{TimedConfigurations}$

$S : \mathit{HonestStrategies}$

$S\dagger\ : \mathit{AdversarialStrategy}$

$SS : \mathit{Strategies}$

SECTION B

List Utilities

B.1 Indexed Operations

When we lookup elements in a list, we use indices that are finite numbers less than the length of the list:

```
open import Fin using (Fin , zero , suc)

Index : List A → Set
Index = Fin ∘ length

indices : List A → List ℕ
indices = upTo ∘ length

_!!_ : (vs : List A) → Index vs → A
(x :: _)  !! zero    =  x
(_ :: xs) !! (suc i) =  xs !! i

delete : (vs : List A) → Index vs → List A
delete []             ()
delete (_ :: xs) zero     = xs
delete (x :: vs) (suc f) = x :: delete vs f

_!!_:=_ : (vs : List A) → Index vs → A → List A
[]          !! ()     := _
(_ :: xs) !! zero  := y = y :: xs
(x :: xs) !! suc i := y = x :: (xs !! i⟨ y ⟩)
```

Also note the type-safe operations of lookup (_ !! _), deletion (*delete*) and update (_ !! _ := _).

B.2 Set-like Interface

When calculating the set of unspent transaction outputs of a ledger in the UTxO model (Section 4.2), we used set-theoretic operations, namely empty set ∅; set difference _ \ _; set union _ ∪ _; set membership _ ∈ _; set cardinality | _ |. First, note that these require that we can decide (i.e. compute) equality between elements of a set. We model this by encapsulating all set-related definitions in a module parametrised by an abstract data type, which is however equipped with *decidable equality*:

```
module Data.Set' {A : Set} (_≟_ : Decidable (_≡_ {A = A})) where

  open import Data.List.Membership.DecPropositional _≟_
    using (_∈?_)
    renaming (_∈_ to _∈′_)

  _≟₁_ : Decidable {A = List A} _≡_
  []       ≟₁ []       = yes refl
  []       ≟₁ _ :: _   = no λ ()
  _ :: _   ≟₁ []       = no λ ()
  x :: xs ≟₁ y :: ys with x ≟ y
  ... | no¬p      = no λ {refl → ¬p refl}
  ... | yes refl with xs ≟₁ ys
  ... | no¬pp   = no λ {refl → ¬pp refl}
  ... | yes refl  = yes refl
```

We now define a set as a list coupled with a proof that it does not contain duplicate elements:

```
open import Data.List.Relation ∘ Unary.Unique.Propositional {A} using (Unique)

record Set' : Set where
  constructor⟨ _ ⟩ ⊣ _
  field list      : List A
        ∘ uniq : Unique list
```

The implementation of the set-theoretic operations now has to preserve the proofs of uniqueness:

```
∅ : Set'
∅ = ⟨ [] ⟩ ⊣ . . .

_∈_ : A → Set' → Set
o ∈ ⟨ os ⟩ ⊣ _ = o∈′os

∣_∣ : Set' → ℕ
∣_∣ = length ∘ list
```

```
_ \ _ : Set’ → Set’ → Set’
⟨ xs ⟩ ⊣ . . . \ ⟨ ys ⟩ ⊣ . . . = ⟨ filter (λ x → ¬?(x ∈? ys)) xs ⟩ ⊣ . . .

_ ∪ _ : Set’ → Set’ → Set’
x@(⟨ xs ⟩ ⊣ . . .) ∪ y@(⟨ ys ⟩ ⊣ . . .) = ⟨ xs ++ list (y \ x) ⟩ ⊣ . . .
```

SECTION C

Decidable Equality

Our decision procedures always rely on the fact that we have decidable equality for the types involved in the propositions under question (see Section 4.4). Here, we demonstrate how to decide equality of the type of actions in the semantics of BitML, but a very similar procedure applies for all other cases:

```
_≟_ : Decidable {A = Action p ads cs vs ds} _≡_

(♯▷  ad) ≟ (♯▷ ∘ad) = yes refl
(♯▷  ad) ≟ (∘ad ▷ˢ  i) = no λ ()

(ad ▷ˢ  i) ≟ (♯▷ ∘ad) = no λ ()
(ad ▷ˢ  i) ≟ (∘ad TRIˢ i′ ) with i SET-fin.≟  i′
... | no¬p    = no λ {refl → ¬p refl}
... | yes refl = yes refl

(c ▷ᵇ  i) ≟ (∘c ▷ᵇ  i′ ) with i SET-fin.≟  i′
... | no¬p    = no λ {refl → ¬p refl}
... | yes refl = yes refl
⋮

_∃≟_ : Decidable {A = ∃Action} _≡_
(p , ads , cs , vs , ds , a) ∃≟ (p′  , ads′  , cs′  , vs′  , ds′  , a′ )
  with p SET-participant.≟  p′
... | no  ¬p  = no λ {refl → ¬p refl}
... | yes refl
  with ads SET-advert.≟  ads′
... | no  ¬p  = no λ {refl → ¬p refl}
... | yes refl
  with cs SET-contract.≟  cs′
... | no  ¬p  = no λ {refl → ¬p refl}
... | yes refl
  with vs SET-ℕ.≟ vs′
... | no ¬p  = no λ { refl → ¬p refl}
... | yes refl
  with ds SET-deposit.≟  ds′
... | no ¬p  = no λ { refl → ¬p refl}
... | yes refl
  with a ≟ a′
... | no ¬p  = no λ { refl → ¬p refl}
... | yes refl = yes refl
```

We can then rightfully use the set-like operations, as described in Appendix B.2:

```
import Data.Set' as SET
module SET-action = SET _∃≟_

Set⟨ Action ⟩ : Set
Set⟨ Action ⟩ = Set'
  where open SET-action
```