# Formal investigation of the Extended UTxO model

Laying the foundations for the formal verification of smart contracts

ORESTIS MELKONIAN, Utrecht University, The Netherlands

This report serves as the proposal of my MSc thesis, supervised by Wouter Swierstra from Utrecht University and Manuel Chakravarty from IOHK.

## 1 INTRODUCTION

Although blockchain technology has opened a whole array of interesting new applications (e.g. secure multi-party computation[Andrychowicz et al. 2014], fair protocol design fair[Bentov and Kumaresan 2014], zero-knowledge proof systems[Goldreich et al. 1991]), reasoning about the behaviour of such systems is an exceptionally hard task. This is partly due to their concurrent nature, but also the fiscal nature of the majority of the applications, which require a much higher degree of rigorousness compared to conventional IT applications.

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 down firm grounds, onto which one can further conduct a formal comparison with account-based models used in Ethereum. Put concisely, the research question posed is:

*How much expressiveness do we gain by extending the UTxO model?*
*Is it as expressive as the account-based model used in Ethereum?*

*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. 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 presents the progress made thus far in terms of (mechanized) formal verification, as well as problems we have encountered and also expect along the way. Section 5 discusses next steps for the remainder of the thesis, as well as a rough estimate on when these milestones will be completed.

## 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.

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

Author's address: Orestis Melkonian, Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, melkon.or@gmail.com.

One particular instance of distributed ledgers are blockchain systems, where (unrelated) transaction are bundled together in blocks, which are linearly connected with hashes and distributed to all participants/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/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 assets, 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 [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/blockchain setting, as can be evidenced by the current attempts being made to overcome this [Bhargavan et al. 2016; Sergey et al. 2018; Setzer 2018]. 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 can a language express, most of these restriction are inherited from the accounting model that the underlying system adhere to.

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 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. 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 state 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.

*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 public key <pub#>, a P2PKH output carries the 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 are 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.

One 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 4.2.

*2.3.3 Extended UTxO.* In this work, we will consider the version of the UTxO model used by IOHK's Cardano[2] blockchain. 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, does not concern our study 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 provide the validation script in the corresponding input with additional information of an arbitrary type.

This extension of the UTxO model has already been implemented[3], but only informally documented[4]. The reason to extend the UTxO model with data scripts is to bring more expressive power to UTxO-based blockchains, hoping that it is 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 to.

*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.

[2]www.cardano.org
[3]https://github.com/input-output-hk/plutus/tree/master/wallet-api/src/Ledger
[4]https://github.com/input-output-hk/plutus/blob/master/docs/extended-utxo/README.md

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. At the time of the creation of a transaction, its owner specifies a certain amount of gas the contract can consume and pays a transaction fee proportional to it. In case of complete depletion, all global state changes are reverted. This is a necessary ingredient for smart contract languages that provide arbitrary looping behaviour, since non-termination of the validation phase is certainly undesirable.

If time permits, we will initially provide a formal justification of Solidity and proceed to formally compare the extended UTxO model against it. 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 the same vein, we will try to introduce a lightweight version of Solidity, which we will refer to as *Featherweight Solidity*.

## 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 and thus can be safely factored out and formalized 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 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 Type Theory. 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 formula into a valid Agda type. Moreover, programs closely follow the structure of the corresponding proof, 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 estimate not to hit such a barrier of complexity, 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 script-like 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[5]. 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) – 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 Agda/Coq 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 out papers about the extended UTxO model, there are still no formally verified results.

## 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.

One 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.

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.

[5]https://iohk.io/

## 4 PRELIMINARY RESULTS

This section gives an overview of the progress made so far in the on-going Agda formalization of the two main objects 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*.

### 4.1 Formal Model I: Extended UTxO

We now set out to model the accounting model of a UTxO-based ledger. All code is publicly available on Github[6]. Let us start with some basic types:

```
Address : Set
Address = ℕ

Value : Set
Value = ℕ

₿ : ℕ → Value
₿ v = v
```

Addresses are represented as natural numbers, since we do not care about details arising from the encoding in an actual implementation. Bitcoin values are again represented as natural numbers, but we provide a prefix notation to emphasize that a number corresponds to a bitcoin amount.

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).

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

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*).

As mentioned previously, we will not dive into the verification of the cryptological 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 statistically impossible for two different values to have the same hash).

```
postulate
  _♯ : ∀{A : Set} → A → Address
  ♯ − injective : ∀{A : Set} {x y : A} → x♯ ≡ y♯ → x ≡ y
```

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

*4.1.1 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
  field
    id    : Address
    index : ℕ

record TxInput : Set where
  field
    outputRef : TxOutputRef
    R         : Set
    redeemer  : State → R
    D         : Set
    validator : State → Value → R → D → Bool
```

*Output references* consist of the address that this output's 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 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.

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 available addresses as module parameters. That is, we package the following definitions in a module with such a parameter, as shown below:

```
module UTxO (addresses : List Address) where

record TxOutput : Set where
  field
    value      : Value
    address    : Index addresses

    Data       : Set
    dataScript : State → 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 : (i : TxInput) → (o : TxOutput) → D i ≡ Data o → State → Bool
runValidation i o refl st = validator i st (value o) (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.

*4.1.2 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:

```
unspentOutputsTx : Tx → Set⟨ TxOutputRef ⟩
unspentOutputsTx tx = fromList (((tx♯) indexed-@_) <$> (indices (outputs tx)))

spentOutputsTx : Tx → Set⟨ TxOutputRef ⟩
spentOutputsTx tx = fromList (outputRef <$> inputs tx)

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

*4.1.3 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:

```
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 SET.∈′ unspentOutputs l
    validDataScriptTypes :
      ∀i → (i∈ : i ∈ inputs tx) →
        D i ≡ Data (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 <$> outputs tx)
    noDoubleSpending :
      SETₒ.noDuplicates (outputRef <$> inputs tx)
    allInputsValidate :
      ∀i → (i∈ : i ∈ inputs tx) →
        let
          out : TxOutput
          out = lookupOutput l (outputRef i) (validTxRefs i i∈) (validOutputIndices i i∈)
        in
          ∀(st : State) →
            T (runValidation i out (validDataScriptTypes i i∈) st)
    validateValidHashes :
      ∀i → (i∈ : i ∈ inputs tx) →
        let
          out : TxOutput
          out = lookupOutput l (outputRef i) (validTxRefs i i∈) (validOutputIndices i i∈)
        in
          toℕ (address out) ≡ (validator i)♯
```

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 three validation condition are more interesting, as they ascertain the submitted transaction is valid, 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 definition of lookup functions is 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 evidences by their usage in the validity conditions).

*4.1.4 Weakening Lemma.* but it would make sense to be able to add without losing the validity of the ledger constructed thus far.

In order to do, we need to first expose the basic datatypes from inside the module, introducing their *primed* version which takes the corresponding module parameter as an index:

```
Ledger′ : List Address → Set
Ledger′ as = Ledger
  where open import UTxO as

Tx′ : List Address → Set
Tx′ as = Tx
  where open import UTxO as

IsValidTx′ : (as : List Address) → Tx′ as → Ledger′ as → Set
IsValidTx′ as t l = IsValidTx t l
  where open import UTxO as

TxOutput′ : List Address → Set
TxOutput′ as = TxOutput
  where open import UTxO as
```

We can now precisely define what it means to weaken an address space; one just adds more available addresses without removing any of the pre-existing addresses:

```
weakenTxOutput : ∀{ as bs } → Prefix as bs → TxOutput′ as → TxOutput′ bs
weakenTxOutput { as } { bs } pr
    record { value = v; dataScript = ds; address = addr }
  = record { value = v; dataScript = ds; address = inject≤ addr (prefix-length pr) }
  where open import UTxO bs

weakenTx : ∀{ as bs } → Prefix as bs → Tx′ as → Tx′ bs
weakenTx { as } { bs } pr record { inputs  = inputs
                                 ; outputs = outputs
                                 ; forge   = forge
                                 ; fee     = fee
                                 }
                        = record { inputs  = inputs
                                 ; outputs = weakenTxOutput pr <$> outputs
                                 ; forge   = forge
                                 ; fee     = fee
                                 }

weakenLedger : ∀{ as bs } → Prefix as bs → Ledger′ as → Ledger′ bs
weakenLedger pr = map (weakenTx pr)
```

For simplicity's sake, we allow extension at the end of the address space instead of anywhere in between[7]. 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 : ∀{ as bs : List Address } { tx : Tx′ as } { l : Ledger′ as }
  → (pr : Prefix as bs)
  → IsValidTx′ as tx l
    ────────────────────────────────────────────
  → IsValidTx′ bs (weakenTx pr tx) (weakenLedger pr 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.

*4.1.5 Example.* 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 and HK 2018].

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 1.

First, we need to set things up by declaring the list of available addresses and opening our module with this parameter.

[7]Technically, we require *Prefix as bs* instead of the more flexible *as* ⊆ *bs*.

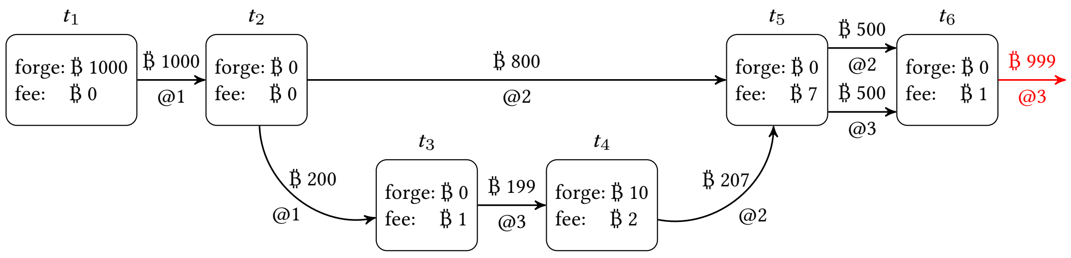

Fig. 1. Example ledger with six transactions (unspent outputs are coloured in red)

```
addresses : List Address
addresses = 1 :: 2 :: 3 :: []

open import UTxO addresses

dummyValidator : State → Value → ℕ → ℕ → Bool
dummyValidator = λ _ _ _ _ → true

withScripts : TxOutputRef → TxInput
withScripts tin = record { outputRef = tin
                         ; redeemer  = λ _ → 0
                         ; validator = dummyValidator
                         }

₿ _@_ : Value → Index addresses → TxOutput
₿ v@ addr = record { value      = v
                   ; address    = addr
                   ; dataScript = λ _ → 0
                   }

postulate
  validator♯ : ∀{ i : Index addresses} → toℕ i ≡ dummyValidator♯
```

Note that, since we will not utilize the expressive power of scripts in this example, we also provide convenient short cuts for defining inputs and outputs with dummy default scripts. Furthermore, we postulate that the addresses are actually the hashes of validators scripts, corresponding to the P2SH scheme in Bitcoin.

We can then proceed to define the individual transactions depicted in Figure 1[8]:

[8] 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.

```
t_1 : Tx
t_1 = record { inputs  = []
             ; outputs = [₿ 1000 @ 0]
             ; forge   = ₿ 1000
             ; fee     = ₿ 0
             }
t_2 : Tx
t_2 = record { inputs  = [withScripts t_10]
             ; outputs = ₿ 800 @ 1 :: ₿ 200 @ 0 :: []
             ; forge   = ₿ 0
             ; fee     = ₿ 0
             }
t_3 : Tx
t_3 = record { inputs  = [withScripts t_21]
             ; outputs = [₿ 199 @ 2]
             ; forge   = ₿ 0
             ; fee     = ₿ 1
             }
t_4 : Tx
t_4 = record { inputs  = [withScripts t_30]
             ; outputs = [₿ 207 @ 1]
             ; forge   = ₿ 10
             ; fee     = ₿ 2
             }
t_5 : Tx
t_5 = record { inputs  = withScripts t_20 :: withScripts t_40 :: []
             ; outputs = ₿ 500 @ 1 :: ₿ 500 @ 2 :: []
             ; forge   = ₿ 0
             ; fee     = ₿ 7
             }
t_6 : Tx
t_6 = record { inputs  = withScripts t_50 :: withScripts t_51 :: []
             ; outputs = [₿ 999 @ 2]
             ; forge   = ₿ 0
             ; fee     = ₿ 1
             }
```

Finally, we can construct a correct-by-construction ledger, by iteratively submitting each transaction along with the proof that it is valid with respect to the ledger constructed thus far[9].

[9] Here, we use a specialized notation of the form $\bullet t_1 \vdash p_1 \oplus t_2 \vdash p_2 \oplus \cdots \oplus t_n \vdash p_n$, where each insertion of transaction $t_x$ requires a proof of validity $p_x$ as well. Technically, the $\oplus$ operator has type $(l : Ledger) \rightarrow (t : Tx) \rightarrow IsValidTx\ t\ l \rightarrow Ledger$.

```
ex-ledger : Ledger
ex-ledger = • t₁ ⊢ record { validTxRefs            = λ i ()
                          ; validOutputIndices     = λ i ()
                          ; validOutputRefs        = λ i ()
                          ; validDataScriptTypes   = λ i ()
                          ; preservesValues        = refl
                          ; noDoubleSpending       = tt
                          ; allInputsValidate      = λ i ()
                          ; validateValidHashes    = λ i ()
                          }
            ⊕ t₂ ⊢ record {...}
            ⊕ t₃ ⊢ record {...}
            ⊕ t₄ ⊢ record {...}
            ⊕ t₅ ⊢ record {...}
            ⊕ t₆ ⊢ record {...}
```

The proof validating the submission of the first transaction $t_1$ is trivially discharged. While the rest of the proofs are quite involved, it is worthy to note that their size/complexity stays constant independently of the ledger length. This is mainly due to the re-usability of proof components, arising from the main functions being inductively defined.

It is now 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-l₆ : list (unspentOutputs l₆) ≡ [t₆₀]
utxo-l₆ = refl
```

## 4.2 Formal Model II: BitML Calculus

Now we move on to our second object of study, the BitML calculus for modelling smart contracts.

All code is publicly available on Github[10]. First, we begin with some basic definitions that will be used throughout this section:

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

```
module Types (Participant : Set) (Honest : List⁺ Participant) where

Time : Set
Time = ℕ

Value : Set
Value = ℕ

record Deposit : Set where
  constructor _ has _
  field
    participant : Participant
    value : Value

Secret : Set
Secret = String

data Arith : Secrets → Set where ...
ℕ⟦_⟧ : ∀{s} → Arith s → ℕ
ℕ⟦_⟧ = ...

data Predicate : Secrets → Set where ...
𝔹⟦_⟧ : ∀{s} → Predicate s → Bool
𝔹⟦_⟧ = ...
```

Instead of giving a fixed datatype of participants, we parametrise our module with a given *universe* of participants and a non-empty list of honest participants. Representation of time and monetary values is again done using natural numbers, while we model participant secrets as simple strings. A deposits 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.

*4.2.1 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. Notice the index of the datatype below, which captures all deposits required:

```
data Precondition : List Value → Set where
  -- volatile deposit
  _?_ : Participant → (v : Value) → Precondition [ v ]
  -- persistent deposit
  _!_ : Participant → (v : Value) → Precondition [ v ]
  -- committed secret
  _♯_ : Participant → Secret → Precondition [ ]
  -- conjunction
  _∧_ : ∀{ vs₁ vsᵣ } → Precondition vs₁ → Precondition vsᵣ → Precondition (vs₁ ++ vsᵣ)
```

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
                → Values -- the deposits it presumes
                → Set where
  -- collect deposits and secrets
  put _ reveal _ if _ ⇒ _ ⊢ _  :  (vs : List Value)
                                → (s : Secrets)
                                → Predicate s′
                                → Contract (v + sum vs) vs′
                                → s′ ⊆ s
                                → Contract v (vs′ ++ vs)
  -- transfer the remaining balance to a participant
  withdraw : ∀{ v } → Participant → Contract v [ ]
  -- split the balance across different branches
  split : (cs : List (∃[ v ] ∃[ vs ] Contract v vs))
     → Contract (sum (proj₁ <$> cs)) (concat (proj₂ <$> cs))
  -- 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[11] 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:

[11] *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.

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

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

```
record Advertisement (v : Value) (vsᶜ vsᵍ : List Value) : Set where
  constructor _⟨ _ ⟩⊢ _
  field
    G     :  Precondition vs
    C     :  Contracts v vs
    valid :  length vsᶜ ⩽ length vsᵍ
          ×  participantsᵍ G ++ participantsᶜ C ⊆ (participant <$> persistentDepositsᵖ G)
```

Notice that in order to construct an advertisement, one has to 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:

```
open BitML (A | B) [A]⁺

ex-ad : Advertisement 5 [200] (200 :: 100 :: [])
ex-ad = ⟨ B ! 200 ∧ A ! 100 ⟩
          split  ( 2 ⊸ withdraw B
                 ⊕ 2 ⊸ after 100 : withdraw A
                 ⊕ 1 ⊸ put [200] ⇒ B : withdraw {201} A
                 )
          ⊢ . . .
```

We first need to open our module with a fixed set of participants (in this case $A$ and $B$). 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 the deposit of ₿ 200 and ₿ 100 respectively.

Looking at the precondition itself, we see that the required deposits will be provided by $B$ and $A$, respectively. 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 authorise the withdrawal of the remaining funds (currently ₿ 201) from $A$.

We have omitted the proofs that ascertain the well-formedness of the *put* and *split* commands, as they are straightforward and do not provide any more intuition[12].

*4.2.2 Small-step Semantics.* BitML is a *process calculus*, specialized for the case of 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

[12] 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.

*action* labels are emitted on every transition representing 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.

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

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

data Action (p : Participant) -- the participant that authorises this action
  :   AdvertisedContracts     -- the contract advertisments it requires
  → ActiveContracts           -- the active contracts it requires
  → Values                    -- the deposits it requires from this participant
  → List Deposit              -- the deposits it produces
  → Set where
  -- commit secrets to stipulate an advertisement
  ♯▷ _ : (ad : Advertisement v vsᶜ vsᵍ)
        → Action p [v, vsᶜ, vsᵍ, ad] [] [] []
  -- spend x to stipulate an advertisement
  _ ▷ˢ _ : (ad : Advertisement v vsᶜ vsᵍ)
         → (i : Index vsᵍ)
         → Action p [v, vsᶜ, vsᵍ, ad] [] [vsᵍ ‼ i] []
  -- pick a branch
  _ ▷ᵇ _ : (c : List (Contract v vs))
         → (i : Index c)
         → Action p [] [v, vs, c] [] []
  ⋮
```

The action datatype is parametrised[13] 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

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

(e.g. dividing a deposit would require a single deposit as the third index and produce two other deposits in its fourth index).

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

Continuing from our previous example advertisement, let's see an example action where $A$ spends the required ₿ 100 to stipulate the example contract[14]:

$$\begin{aligned}&\textit{ex-spend} : \textit{Action}\ A\ [5, [100], 200 :: 100 :: [\,], \textit{ex-ad}]\ [\,]\ [100]\ [\,]\\&\textit{ex-spend} = \textit{ex-ad} \triangleright^{s} 1\end{aligned}$$

Configurations are now built from advertisements, active contracts, deposits, action authorizations and committed/revealed secrets:

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

```
data Configuration′ : -- current               × required
                        AdvertisedContracts × AdvertisedContracts
                      → ActiveContracts     × ActiveContracts
                      → List Deposit        × List Deposit
                      → Set where
  -- empty
  ∅ : Configuration′ ([], []) ([], []) ([], [])
  -- contract advertisement
  `_ : (ad : Advertisement v vsᶜ vsᵍ)
    → Configuration′ ([v, vsᶜ, vsᵍ, ad], []) ([], []) ([], [])
  -- active contract
  ⟨_,_⟩ᶜ : (c : List (Contract v vs))
          → (v′ : 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
           → (s : Secret)
           → (n : ℕ ⊎ ⊥)
           → {pr : n ≡ inj₁ n′ → n′ ≡ length s}
           → Configuration′ ([], []) ([], []) ([], [])
  -- revealed secret
  _:_♯_ : Participant
        → (s : Secret)
        → (n : ℕ)
        → {pr : length s ≡ n}
        → Configuration′ ([], []) ([], []) ([], [])
  -- parallel composition
  _|_ : Configuration′ (adsˡ, radsˡ) (csˡ, rcsˡ) (dsˡ, rdsˡ)
      → Configuration′ (adsʳ, radsʳ) (csʳ, rcsʳ) (dsʳ, rdsʳ)
      → Configuration′ (adsˡ          ++ adsʳ, radsˡ ++ (radsʳ \ adsˡ))
                       (csˡ           ++ csʳ  , rcsˡ  ++ (rcsʳ  \ csˡ))
                       ((dsˡ \ rdsʳ) ++ dsʳ  , rdsˡ  ++ (rdsʳ  \ dsˡ))
```

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

```
Configuration : AdvertisedContracts → ActiveContracts → List Deposit → Set
Configuration ads cs ds = Configuration′ (ads, []) (cs, []) (ds, [])
```

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:

```
data _ ⟶ _ : Configuration ads cs ds → Configuration ads′ cs′ ds′  → Set where
  DEP-AuthJoin :
    ⟨ A, v ⟩ᵈ | ⟨ A, v′  ⟩ᵈ | Γ ⟶ ⟨ A, v ⟩ᵈ | ⟨ A, v′  ⟩ᵈ | A [0↔1] | Γ

  DEP-Join :
    ⟨ A, v ⟩ᵈ | ⟨ A, v′  ⟩ᵈ | A [0↔1] | Γ ⟶ ⟨ A, v + v′  ⟩ᵈ | Γ

  C-Advertise : ∀{Γ ad}
    → ∃[p ∈ participantsᵍ (G ad)] p ∈ Hon)
    ────────────────────────────────────
    → Γ ⟶ ‘ad | Γ

  C-AuthCommit : ∀{A ad Γ }
    → secrets (G ad) ≡ a₀ . . . aₙ
    → (A ∈ Hon → ∀[i ∈ 0 . . . n] aᵢ≢⊥)
    ────────────────────────────────────
    → ‘ad | Γ ⟶ ‘ad | Γ | . . . ⟨ A : aᵢ♯Nᵢ ⟩ . . . |A auth [♯▷ad]

  C-Control : ∀{Γ C i D}
    → C ‼ i ≡ A₁ : A₂ : . . . : A : D
    ────────────────────────────────────
    → ⟨ 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. The first pair of rules initially appends the authorisation 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 authorisations 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 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 precondition, 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 a discrepancy in the [*C-AuthRev*] rule, namely that there was no context Γ. Moreover, in order to later facilitate equational reasoning, we re-factored the [*C-Control*] to not contain the inner step as a hypothesis, but instead immediately inject it in the result operand of the step relation.

A scrupulous reader might have noticed that the inference rules above have elided any treatment of timely 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 timely 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}
      ──────────────────────────
    → 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
```

Having defined the step relation in this way allows for equational reasoning, a powerful tool for writing complex proofs:

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

  _□ : (M : Configuration ads cs ds) → M ↠ M

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

begin _ : ∀{M N} → M ↠ N → M ↠ N
```

*4.2.3 Example.* 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 $t$:

```
tc : Advertisement 1 [] (1 :: 0 :: [])
tc = ⟨ A :!1 ∧ A : ♯a ∧ B :!0 ⟩
         reveal [a] ⇒ withdraw A ⊢ . . .
       ⊕ after t : withdraw B
```

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

```
tc-semantics : ⟨ A , 1 ⟩ᵈ —↠ ⟨ A , 1 ⟩ᵈ | A : a ♯ 6
tc-semantics =
  begin
    ⟨ A , 1 ⟩ᵈ
  —→⟨ C-Advertise ⟩
    ` tc | ⟨ A , 1 ⟩ᵈ
  —→⟨ C-AuthCommit ⟩
    ` tc | ⟨ A , 1 ⟩ᵈ | ⟨ A : a ♯ 6 ⟩ | A [ ♯▷  tc ]
  —→⟨ C-AuthCommit ⟩
    ` tc | ⟨ A , 1 ⟩ᵈ | ⟨ A : a ♯ 6 ⟩ | A [ ♯▷  tc ] | B [ ♯▷  tc ]
  —→⟨ C-AuthInit ⟩
    ` tc | ⟨ A , 1 ⟩ᵈ | ⟨ A : a ♯ 6 ⟩ | A [ ♯▷  tc ] | B [ ♯▷  tc ] | A [ tc ▷ˢ 0 ]
  —→⟨ C-Init ⟩
    ⟨ tc , 1 ⟩ᶜ | ⟨ A : a ♯ inj₁ 6 ⟩
  —→⟨ C-AuthRev ⟩
    ⟨ tc , 1 ⟩ᶜ | A : a ♯ 6
  —→⟨ C-Control ⟩
    ⟨ [ reveal [ a ] ⇒ withdraw A ⊢ … ] , 1 ⟩ᶜ | A : a ♯ 6
  —→⟨ 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*]).

## 4.3 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 such 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.Relation ∘ Permutation.Inductive 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 generalise 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 generalisation of the [*DEP-AuthJoin*] rule[15]:

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

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* [hom 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 elegant and pragmatic, we will incorporate it in our full-blown BitML calculus.

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

| | 2018 | 2019 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Dec | Jan | Feb | Mar | Apr | May | Jun | Jul |
| **UTxO** | | | | | | | | |
| Basic abstract model | ■ | ■ | | | | | | |
| Weakening lemma | | ■ | | | | | | |
| Extend with data scripts | | | ■ | | | | | |
| Multi-currency | | | | ■ | | | | |
| **BitML** | | | | | | | | |
| Small-step Semantics | ■ | ■ | ■ | | | | | |
| Symbolic Model | | | | ■ | | | | |
| Computational Model | | | | ■ | | | | |
| Coherence | | | | | ■ | ■ | | |
| Computational Soundness | | | | | ■ | ■ | | |
| **UTxO-BitML Integration** | | | | | | | | |
| Compiling BitML contracts to UTxO | | | | ■ | ■ | | | |
| **Solidity** | | | | | | | | |
| Featherweight Model | | | | ■ | | | | |
| Comparison framework | | | | | ■ | ■ | | |
| **Evaluation** | | | | | | | | |
| Example contracts | | | | | ■ | ■ | ■ | |
| **Writeup** | | | | | | | | |
| Write final thesis | | | | | | ■ | ■ | ■ |

Fig. 2. My workplan.

## 5 PLANNING

### 5.1 Extended UTxO

... multi-currency ...

### 5.2 BitML Calculus

... symbolic runs ... computational runs ... coherence

### 5.3 UTxO-BitML Integration

### 5.4 Plutus Integration

### 5.5 Featherweight Solidity

### 5.6 Formal Comparison

... lots of examples ...

### 5.7 Proof Automation

### 5.8 Timetable

... mention things that are really out-of-scope ...

## REFERENCES

2010. Script - Bitcoin Wiki. Retrieved 2/2019 from https://en.bitcoin.it/wiki/Script

2013. Homotopy Type Theory: Univalent Foundations of Mathematics. *CoRR* abs/1308.0729 (2013). arXiv:1308.0729 http://arxiv.org/abs/1308.0729

2018. Formal verification of a Cardano wallet. Retrieved 2/2019 from https://cardanodocs.com/files/formal-specification-of-the-cardano-wallet.pdf

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

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.

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

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.

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.

S 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.

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.

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.

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).

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

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