SECTION 1

Introduction

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]). 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 rigorousness 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 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. Moreover, we give an overview of related work, putting an emphasis on existing tools based on static analysis.
- Section **??** describes the technology we will use to formally reason about the problem at hand and some key design decisions we set upfront.

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

- Section 3 describes the formalization of an abstract model for UTxO-based blockchain ledgers.
- Section **??** concerns the formalization of our second object of study, the Bitcoin Modelling Language.
- Section 4 gives an overview of relavant previous work, ranging from static analysis tools to type-driven verification approaches.
- Section **??** discusses possible next steps to continue the line of work stemming from this thesis.
- Section **??** 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 [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 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. 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 $\mathrm{OP\_}$ 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 following script:

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

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.

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 ??.

2.3.3 Extended UTxO

In this work, we will consider the version of the UTxO model used by IOHK's Cardano blockchain[2]. 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, 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 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 (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.

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*.

SECTION 3

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; this gives rise to a notion of *weakening* of available addresses, 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[5].

We start with the basic types, keeping them abstract since we do not care about details arising from the encoding in an actual implementation:

```
postulate
  Address : Set
  Value : Set
  ₿ : ℕ → Value
```

We assume there are types representing addresses and bitcoin values, but also require the ability to construct a value out of a natural number. In the examples that follow, we assume the simplest representation, where both types are the natural numbers.

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 highly unlikely for two different values to have the same hash).

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

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

3.1 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    : Address
        index : ℕ

record TxInput {R D : Set} : Set where
  field outputRef : TxOutputRef
        redeemer  : State → R
        validator : State → Value → R → 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 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 {D : Set} : Set where
  field value      : Value
        address    : Index addresses
        dataScript : State → D

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 ≡ D 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.

3.2 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 ⟩
    spentOutputsTx      = (outputRef ⟨$⟩_) ∘ inputs
    unspentOutputsTx tx = ((tx ♯) ,_) ⟨$⟩ (indices (outputs tx))
```

3.3 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 ∈ 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 : 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 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. Users should only have a type-safe interface to construct ledgers, where each time a transaction is submitted along with the proof that it is valid with respect to the ledger constructed thus far. We provide such an interface with a proof-carrying variant of the standard list construction:

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

3.4 Decision Procedure

Intrinsically-typed ledgers are correct-by-construction, but this does not come for free; we now need to provide substantial proofs alongside each time we submit a new 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* [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 copy 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 *metaprogramming* 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 operators 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 3.9 we give an example construction of a valid ledger and demonstrate that our decision procedure discharges all proof obligations with calls to *toWitness*.

3.5 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, 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
⋮
```

We can now precisely define what it means to weaken an address space; the only necessary ingredient is a *hash-preserving injection* from a smaller address space $\mathbb{A}$ to a larger address space $\mathbb{B}$:

```
module Weakening
  (𝔸 : Set) (_ ♯ᵃ : Hash 𝔸) (_ ≟ᵃ _ : Decidable {A = 𝔸} _≡_)
  (𝔹 : Set) (_ ♯ᵇ : Hash 𝔹) (_ ≟ᵇ _ : Decidable {A = 𝔹} _≡_)
  (A ↪ B : 𝔸 , _ ♯ᵃ ↪ 𝔹 , _ ♯ᵇ)
  where

  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[6]. Since we allow the flexibility for arbitrary injective functions, our weakening result will hopefully prove resilient to such scenarios.

3.6 Combining

Ideally, one would wish for a modular reasoning process, where it is possible to examine subsets of unrelated transactions in a compositional manner. This has to be done in a constrained manner, since we need to preserve the proof of validity when combining two ledgers l and l'.

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)
```

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

$$\rightarrow ValidLedger\ l''$$

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 notice a useful interplay between weakening and combining: 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.

3.7 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 [eut 2019]. 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. $D = State$).

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.

3.8 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*[7]. 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:

$$Value = List\ (Hash \times \mathbb{N})$$

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

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 [8]:

```
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

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, currently employed in the Cardano blockchain [mul 2019]. 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}.

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

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 3.4:

```
∃?  :  (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
```

3.9 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 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 1. The six transactions $t_1 \ \ldots \ t_6$ are self-explanatory, each containing a forge and fee value. Notice the special transaction c_0, which

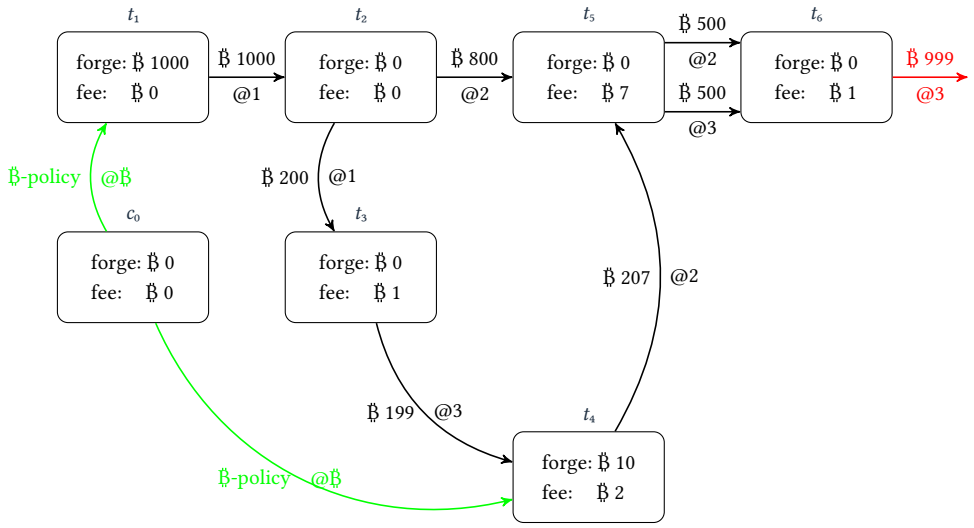

Fig. 1. Example ledger with six transactions (unspent outputs are coloured in red)

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.

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

```
Address : Set
Address = ℕ

1ᵃ , 2ᵃ , 3ᵃ , ₿ᵃ : Address
1ᵃ = 111  -- first address
2ᵃ = 222  -- second address
3ᵃ = 333  -- third address
₿ᵃ = 1234 -- BIT identifier

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

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

```
₿-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 = [(₿ᵃ , 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 → Index addresses → 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 1; 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 @ 0]
            ; forge   = ₿ 1000
            ; fee     = ₿ 0
            }
t₂ = record {inputs  = [withScripts t₁₀]
```

```
                ; outputs = ₿ 800 @ 1 :: ₿ 200 @ 0 :: []
                ; forge   = ₿ 0
                ; fee     = ₿ 0
                }
t₃ = record {inputs  = [ withScripts t₂₁ ]
                ; outputs = [ ₿ 199 @ 2 ]
                ; forge   = ₿ 0
                ; fee     = ₿ 1
                }
t₄ = record {inputs  = withScripts t₃₀ :: withPolicy c₀₁ :: []
                ; outputs = [ ₿ 207 @ 1 ]
                ; forge   = ₿ 10
                ; fee     = ₿ 2
                }
t₅ = record {inputs  = withScripts t₂₀ :: withScripts t₄₀ :: []
                ; outputs = ₿ 500 @ 1 :: ₿ 500 @ 2 :: []
                ; forge   = ₿ 0
                ; fee     = ₿ 7
                }
t₆ = record {inputs  = withScripts t₅₀ :: withScripts t₅₁ :: []
                ; outputs = [ ₿ 999 @ 2 ]
                ; 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
  eq₀  : ₿-validator        ♯ ≡ ₿ ᵃ
  eq₁₀ : (mkValidator t₁₀) ♯ ≡ 1 ᵃ
  ⋮
  eq₆₀ : (mkValidator t₆₀) ♯ ≡ 3 ᵃ

{-# 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*[9].

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
```

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

```
case goal of λ
  { (vtx _ _ _ t l) →
    t′ ←unquoteTC t
    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[10], 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 , ... }
```

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

SECTION 4

Related Work

4.1 Static Analysis Tools

TODO: BitML Liquidity ... TODO: Madmax ... TODO: Mooly ...

4.2 Scilla

TODO: extrinsic ...

4.3 Setzer's Agda model

TODO: similar, but ...

References

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

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

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.

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).

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.

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.

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.

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

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).

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.