Program logics for ledgers

Orestis Melkonian
University of Edinburgh, Scotland
Input Output, Global

Wouter Swierstra
Utrecht University, The Netherlands

James Chapman
Input Output, Global

Abstract

Distributed ledgers nowadays manage substantial monetary funds in the form of cryptocurrencies such as Bitcoin, Ethereum, and Cardano. For such ledgers to be safe, operations that add new entries must be cryptographically sound—but it less clear how to reason effectively about such ever-growing linear data structures.

This paper views distributed ledgers as *computer programs*, that, when executed, transfer funds between various parties. As a result, familiar program logics, such as Hoare logic and separation logic, can be defined in this novel setting. Borrowing ideas from concurrent separation logic, this enables modular reasoning principles over arbitrary fragments of any ledger.

All the results presented in this paper have been mechanised in the Agda proof assistant and are publicly available.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Separation logic

Keywords and phrases blockchain, distributed ledgers, UTxO, Hoare logic, separation logic, program semantics, formal verification

Digital Object Identifier 10.4230/LIPIcs...

Funding *Orestis Melkonian*: This work was supported by Input Output (iohk.io) through their funding of the Edinburgh Blockchain Technology Lab.

1 Introduction

Ledger-based cryptocurrencies manage large amounts of money and record monetary transfers in copious detail. The market capitalisation of the top ten cryptocurrencies is currently valued at over 750B USD. The underlying blockchain that records transactions, gigabytes in size, is an ever growing linear data structure. On the Cardano blockchain alone, transactions valuing over 300M USD are recorded every day. How could we ever hope to reason about such colossal and monolithic data structures?

To answer this question, this paper shows how familiar programming logics can be adapted to enable *effective* and *modular* reasoning about ledger-based financial transactions. Just as imperative programs mutate computer memory, financial transactions mutate bank accounts. Hoare logic and separation logic enable us to rigorously prove the correctness of computer programs. Surprisingly—as this paper demonstrates—these logics can be adapted to reason about the financial transactions stored on a ledger with the same degree of confidence. To this end, this paper makes the following novel contributions:

- First and foremost, we show how the financial transactions stored in a ledger form a simple programming language. We present denotational, operational, and axiomatic semantics of account-based ledgers (Section 2), together with a *separation logic* that enables modular reasoning over ledger fragments (Section 3). The separation logic that

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

arises in this context, however, turns out to be subtly different, yet strictly more general than the typical logics used to reason about computer programs.

- We show how these same semantics—denotational, operational, and axiomatic—can be given for blockchain ledgers (Section 4), in particular ones based on the *unspent transaction outputs* (or UTxO) model, such as Bitcoin [20] and Cardano [6]. Separation logic, however, poses more of a challenge as the hash-based nature of the UTxO model adds new side conditions to the frame rule that were not necessary for account-based ledgers.
- To address this problem, we propose a novel variant of UTxO, dubbed *Abstract UTxO*. In contrast to regular UTxO, our Abstract UTxO model supports compositional reasoning using separation logic without further side conditions (Section 5). The resulting logic enables us to reason locally and safely about a limited number of transactions, sprinkled arbitrarily throughout a larger ledger.

All the definitions and theorems presented in this paper have been mechanised in Agda and are publicly available:

`https://omelkonian.github.io/hoare-ledgers`.

We use traditional mathematical notation rather than "literate programming" style. The proofs themselves are typically quite simple— the hard work is in finding the definitions that make them so.

2 Ledgers & Semantics

To start things off, we give a formal definition of the syntax and semantics of a simple ledger. This illustrates one of the key ideas underlying our work, applying programming language theory in a novel domain. For the sake of simplicity, we assume a fixed set of participants $\mathcal{P}$. Each participant may spend or receive *funds*. At any given point in time, we can model the state of all the participants' accounts as a (finite) map, mapping each participant to their current balance:

$$S := \mathcal{P} \mapsto \mathbb{Z}$$

Note that this model allows negative account balances; typically, however, we would only allow non-negative balances or at least put a bound to the amount of debt an account can accrue. This decision was made mainly for pedagogical purposes—we will revisit this choice in the next section.

We will treat a finite map σ as a function from keys to values for simplicity, retrieving a key k with $\sigma(k)$ and constructing a new map with anonymous λ-functions.

A *ledger* records the history of transfers between accounts, which we can represent as a list of transactions of the form:

```
Alice pays Bob 5;
Alice pays Carroll 10;
Dana pays Alice 2;
...
```

We can view such a ledger as a *program*, describing updates to the state of the accounts modelled by S. The abstract syntax of our ledger can be defined as:

$$T := \mathcal{P} \xrightarrow{n} \mathcal{P}$$

$$L := \epsilon \mid T; L$$

Each transaction T describes the transfer of funds n from one person to another; the ledger consists of a *list* of such transactions, with the most recent transaction last. Now that we have the syntax in place, we present the semantics of L in three different styles.

2.1 Denotational semantics

We give the denotational semantics of a ledger by mapping L to a function of type $S \to S$, executing all the transactions in the ledger starting from a given state with given account balances. This semantics is straightforward to define, by composition ($\circ$) of the semantics for a single transaction given by function d:

$$\begin{aligned} [\![\_]\!] &: L \to S \to S \\ [\![\epsilon]\!] &= \mathrm{id} \\ [\![t;l]\!] &= [\![l]\!] \circ d(t) \end{aligned} \qquad d : T \to S \to S \quad d(p_1 \xrightarrow{n} p_2)(\sigma) = \lambda p. \begin{cases} \sigma(p) - n & \text{if } p = p_1 \neq p_2 \\ \sigma(p) + n & \text{if } p = p_2 \neq p_1 \\ \sigma(p) & \text{otherwise} \end{cases}$$

We can formulate and prove a simple compositionality result, stating that the appending of ledgers is mapped to the composition of their denotations.

▶ **Theorem 1.** *For any ledgers l_1 and l_2, we have $[\![l_1 +\!\!+ l_2]\!] = [\![l_2]\!] \circ [\![l_1]\!]$.*

This result, however, gives us only limited modularity—we still need to break a ledger into sequential pieces that we consider individually. To handle large ledgers, however, we would like to reason about *arbitrary* ledger fragments, in particular some subset of the transactions that are related to a specific smart contract in the blockchain setting.

2.2 Operational semantics

Alternatively, we can describe an operational semantics for L. To do so, we define a relation: $\langle l, \sigma \rangle \to \tau$, denoting that running the ledger l in the state σ terminates in the state τ.

The definition of this relation is entirely straightforward:

$$\frac{}{\langle \epsilon, \sigma \rangle \to \sigma}\ \text{STOP} \qquad \frac{\langle l, d(t)(\sigma) \rangle \to \tau}{\langle t; l, \sigma \rangle \to \tau}\ \text{STEP}$$

It is straightforward to establish that these two semantics coincide:

▶ **Theorem 2.** *For any ledger l and state σ, we have that $[\![l]\!](\sigma) = \tau$ iff $\langle l, \sigma \rangle \to \tau$.*

Naturally, we can use the equivalence of the two semantics to transfer previous results to the operational setting as corollaries, e.g. get the following compositionality principle for the combination of two ledgers as a corollary of Theorem 1 and 2:

▶ **Corollary 3.** *For any ledgers l, l' and states σ, σ', τ we have that $\langle l, \sigma \rangle \to \sigma'$ and $\langle l', \sigma' \rangle \to \tau$ iff $\langle l +\!\!+ l', \sigma \rangle \to \tau$.*

2.3 Axiomatic semantics

We can also define an *axiomatic semantics* for L. To do so, we define inference rules for Hoare triples of the form $\{P\}\ l\ \{Q\}$, where P and Q are predicates on our state space S.

$$\frac{}{\{P\}\ \epsilon\ \{P\}}\ \text{STOP} \qquad \frac{\{P\}\ l\ \{Q\}}{\{P \circ d(t)\}\ t; l\ \{Q\}}\ \text{STEP}$$

We can then add the typical rule for weakening/strengthening pre-/post-conditions:

$$\frac{P' \Rightarrow P \qquad \{P\}\ l\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ l\ \{Q'\}}\ \textsc{consq}$$

Once again, we can relate our axiomatic semantics to its operational and denotational counterparts:

▶ **Theorem 4.** *$\{P\}\ l\ \{Q\}$ holds iff $P(\sigma)$ and $\langle l, \sigma\rangle \to \tau$ implies $Q(\tau)$ for all σ and τ.*

▶ **Theorem 5.** *$\{P\}\ l\ \{Q\}$ holds iff $P(\sigma)$ and $[\![l]\!](\sigma) = \tau$ implies $Q(\tau)$ for all σ and τ.*

Again, we can derive a sequencing rule as a corollary of the equivalent statement about for ledgers in the previous semantics:

$$\frac{\{P\}\ l_1\ \{Q\} \qquad \{Q\}\ l_2\ \{R\}}{\{P\}\ l_1 +\!\!+\ l_2\ \{R\}}\ \textsc{app}$$

▶ Remark 6. For the rest of the paper, whenever we **axiomatize** inference rules (e.g. STOP, STEP, CONSQ) we imply that they are at the same time proven *sound* with respect to the denotational or operational semantics. Moreover, any subsequent **derived** inference rules (e.g. APP above) are implicitly proven using either the axioms or directly appealing to their denotational/operational counterparts.

Example specification

Equipped with a program logic for transactions, we can now formulate properties using Hoare triples and prove them in a sequential fashion akin to *equational reasoning*:[1]

$$\begin{aligned}
&\{\lambda\sigma.\ \sigma(A) = 2\}\\
&\qquad A \xrightarrow{1} B\\
&\{\lambda\sigma.\ \sigma(A) = 1\}\\
&\qquad A \xrightarrow{1} C\\
&\{\lambda\sigma.\ \sigma(A) = 0\}
\end{aligned}$$

The above reads as follows: we start from a state where A holds 2 units of currency; then execute a transfer of one of those from A to B resulting in a state where only a single unit remains in A's account; and we subsequently transfer the other unit to C reaching a final state where A holds no funds.

However, to prove such statements amounts to providing evidence for each Hoare triple at each step, which involves predicates over the whole state although each transaction can only refer to two distinct participants. In the case of a more complicated state space than just a single participant, this approach is *non-compositional*, since you would need to talk about the whole state you care about in one go. This is precisely the reason we now turn our attention to *separation logic* [24].

[1] To see the rules of our various logics in use, we provide some example Hoare-style proofs in Appendix A.

3 Partiality & Separation

In the previous section, we defined the simplest possible semantics for financial ledgers. There are, however, two important drawbacks to the semantics that we have seen so far.

Firstly, we assumed that the value associated with each participant was an *integer*—yet in many financial settings, there is a limit to how many funds may be overdrawn. As a result, attempting to transfer funds may fail. To model this, we revise the state space to disallow negative balances:

$$S := \mathcal{P} \mapsto \mathbb{N}$$

Note that our entire approach can be trivially shifted to any fixed bound other than zero, enabling the modelling of bounded debt. The semantics become more involved, since we need to explicitly handle situations where more than the available funds are being transferred.

The second problem with our semantics is more subtle. Although each of these semantics lets us reason about the ledger $l_1 +\!\!+ l_2$ in terms of the meaning of l_1 and l_2, we cannot easily do the same for an interleaving of the transactions from l_1 and l_2. To address these issues, this section revises our previous semantics, accommodating for partiality, and defines an alternative axiomatic semantics based on *separation logic*.

3.1 Denotational semantics

On the denotational side, errors will be reflected on the *domain* of our semantics which will now move from a total to a partial function space $S \to \mathit{Maybe}\ S$, where just constructs a new state after successful execution and nothing signals an error. As a result, the semantics of a ledger can no longer use function composition to sequence the semantics of its constituent transactions; we need to define the *Kleisli composition* that collapses to nothing if the first partial function fails:

$$(f \ggg g)(s) = \begin{cases} g(s) & \text{if } f(s) = \mathsf{just}\ s \\ \mathsf{nothing} & \text{if } f(s) = \mathsf{nothing} \end{cases}$$

Using this we can now iterate the transactions as before to get the denotation of a ledger:

$$\begin{aligned} [\![\_]\!] &: L \to S \to \mathit{Maybe}\ S \\ [\![\epsilon]\!] &= \mathsf{just} \\ [\![t;l]\!] &= d'(t) \ggg [\![l]\!] \end{aligned} \qquad \begin{aligned} & d' : T \to S \to \mathit{Maybe}\ S \\ & d'(p_1 \xrightarrow{n} p_2)(\sigma) = \begin{cases} \mathsf{just}\ d(p_1 \xrightarrow{n} p_2)(\sigma) & \text{if } \sigma(p_1) \geq n \\ \mathsf{nothing} & \text{otherwise} \end{cases} \end{aligned}$$

The semantics of a single transaction, given by the function d', now checks the validity of each transfer and fails if insufficient funds are available, otherwise reuses the denotation d from the previous section which is now guaranteed to never reach a negative value. We will write "t is valid in σ" as a uniform way to express the validity of a transaction t with respect to a given state σ, which will become more intricate when we consider blockchain ledgers in the next section.

3.2 Operational semantics

The operational semantics remain mostly unchanged, aside from an additional check in the STEP rule:

$$\frac{}{\langle \epsilon, \sigma \rangle \to \sigma}\ \text{STOP} \qquad \frac{t \text{ is valid in } \sigma \qquad \langle l, d(t)(\sigma) \rangle \to \tau}{\langle t; l, \sigma \rangle \to \tau}\ \text{STEP}$$

It is no coincidence that there is such a minimal overhead on the operational semantics. Rather, this stems from its *relational* presentation, where partiality is inherently possible and rules only specify successful behaviour.

3.3 Axiomatic semantics

The base case is not affected in any way:

$$\frac{}{\{P\}\ \epsilon\ \{P\}}\ \text{STOP}$$

We have more choice when it comes to adapting our axiomatic semantics: should we ensure all transactions succeed? Or do we want to observe failing transactions?

- **Total correctness:** By enforcing that the precondition P implies a transaction's validity, we ensure that adding a new transaction in a ledger always succeed:

$$\frac{P \Rightarrow \text{t is valid} \qquad \{P\}\ l\ \{Q\}}{\{P \circ d(t)\}\ t; l\ \{Q\}}\ \text{STEP}$$

  This choice lets us focus on the successful cases only.
- **Partial correctness:** Alternatively, we can reason about those cases where a transaction fails, using the error-handling semantics d':

$$\frac{\{P\}\ l\ \{Q\}}{\{\uparrow P \circ d'(t)\}\ t; l\ \{Q\}}\ \text{STEP}$$

  Here the predicate transformer, $\uparrow$, lifts a predicate over S to a predicate over *Maybe* S. There are two canonical ways to achieve this lifting: the **weak** lifting that collapses to true when a transaction fails; the **strong** lifting that collapses to false upon failure.

Throughout this paper, we will use the *strong and partial* version of correctness, which we again prove **sound** with respect to the denotational semantics:

▶ **Theorem 7.** *$\{P\}\ l\ \{Q\}$ holds iff $P(\sigma)$ and $[\![l]\!](\sigma) = \mathsf{just}\ \tau$ implies $Q(\tau)$ for all σ and τ.*

3.4 Separation logic

In the previous sections, we gave three semantics for ledgers. Yet each relies on having the *complete* ledger at our disposal—we cannot yet use these semantics to reason about arbitrary subsets of transactions, independent of the others. To this end, we define a *separating conjunction* combining two predicates, P and Q, on our state space S. Before we do so, however, we need to consider how to combine states S. In most program language semantics, this is done by splitting the *heap* into two (disjoint) parts. The separating conjunction, $P * Q$, is then defined as follows:

$$(P * Q)(\sigma) := \exists \sigma_1.\ \exists \sigma_2.\ P(\sigma_1)\ \wedge\ Q(\sigma_2)\ \wedge\ \sigma = \sigma_1 \uplus \sigma_2$$

When considering financial ledgers, however, we can do better. As each transaction preserves the overall funds, we do *not* require the maps to be disjoint; instead, we can divide the *funds* from both maps into two distinct parts! To do so, we begin by defining the following operation on operation combining states by pointwise addition of their funds:

$$(\sigma_1 \oplus \sigma_2)(p) := \sigma_1(p) + \sigma_2(p)$$

Using this operation, we can now define the separating conjunction of predicates as follows:

$$(P * Q)(\sigma) := \exists \sigma_1.\ \exists \sigma_2.\ P(\sigma_1)\ \wedge\ Q(\sigma_2)\ \wedge\ \sigma = \sigma_1 \oplus \sigma_2$$

The frame rule, used to introduce the separating conjunction, now becomes:

$$\frac{\{P\}\ l\ \{Q\}}{\{P * R\}\ l\ \{Q * R\}}\ \text{FRAME}$$

Crucially, this version of the frame rule does not have the usual side conditions required to reason about imperative languages, namely, that the set of variables modified by l must be disjoint from the free variables mentioned by R. Intuitively, this rule is valid since transactions preserve the total amount of funds in circulation: we can split off some of these funds (leaving funds that satisfy R left over), move these funds in accordance with l, and then recombine the result with the funds satisfying R.

To complete this semantics, however, we need to add a few basic rules that are currently missing. The rule for handling a single transaction is very simple indeed:

$$\frac{}{\{p_1 \mapsto n\}\ p_1 \xrightarrow{n} p_2\ \{p_2 \mapsto n\}}\ \text{SEND}$$

The precondition, $p_1 \mapsto n$, states that participant p_1 has a total of n funds (and all other participants have none). After executing this transaction, p_2 has received these n funds (and all other participants, including p_1, have none). By itself, this rule does not seem useful—but in combination with the frame rule above, it can be used to execute a single transaction in any larger state—leaving all other funds untouched.

The final two rules describe the behaviour of an entire ledger:

$$\frac{}{\{emp\}\ \epsilon\ \{emp\}}\ \text{EMPTY} \qquad \frac{\{P\}\ l_1\ \{Q\} \qquad \{Q\}\ l_2\ \{R\}}{\{P\}\ l_1 +\!\!+\ l_2\ \{R\}}\ \text{APP}$$

The first rule states that the empty ledger leaves the empty state unchanged; the second describes how transactions from two non-empty ledgers are run sequentially.

Concurrent separation logic

Furthermore, we can define a (non-deterministic) interleaving operation on ledgers, $l_1 \,||\, l_2$. One of the more promising observations we can make is that the familiar rule for concurrent separation logic also holds for the interleaving of two ledgers:

$$\frac{\{\ P_1\ \}\quad l_1\quad \{\ Q_1\ \} \qquad \{\ P_2\ \}\quad l_2\quad \{\ Q_2\ \}}{\{\ P_1 * P_2\ \}\quad l_1 \,||\, l_2\quad \{\ Q_1 * Q_2\ \}}\ \text{PAR}$$

This provides a modular reasoning principle for ledgers: it allows us to focus on an arbitrary subset of the ledger's transactions and reason about this subset in isolation. Whenever we interleave its transactions with the remainder of the ledger, any properties we have established still hold of the composite ledger.

▶ Remark 8. At this point, we have discovered that the monoidal (de)composition of values gives us the modularity we desired on arbitrary ledger interleavings $l_1 \,||\, l_2$. There is nothing preventing us from bringing that lesson back to the denotational semantics—Theorem 7 assures us that the same principle holds there—although arguably it was much harder to uncover in the denotational or operational setting by themselves.

However, the need for a separation logic arises from the modularity we are after at the specification level, that is when we consider the (monoidal) combination of two predicates, where the notion of Hoare triples and particularly the separating conjunction of two predicates $P * Q$ provides a *convenient* abstraction for reasoning about ledger fragments.

4 UTxO

In the coming sections, we will explore how to define similar semantics for UTxO-based blockchains. To do so, requires abandoning our previous assumption that there is a fixed set of participants, each with their own account. Instead, funds are locked by a *validator script*. Funds can be spent by anyone, provided they can provide the *redeemer data*, that is, data mapped to true by the associated validator script:

$$\mathit{Output} := \{\mathsf{validator} : \mathit{DATA} \to \mathbb{B}, \mathsf{value} : \mathbb{N}\}$$

Typically, such a validator script might require a public key to unlock the funds which are locked by the corresponding private key.

Transactions will now need to consume previous (unspent) outputs, to which we can refer by using the transaction's hash and the index into its outputs (we write $t^{\#}_{k}$ to refer to the k-th output of t), as well as providing redeemer data

$$\begin{aligned} \mathit{Ref} &:= \{\mathsf{tx} : \mathit{HASH}, \mathsf{index} : \mathbb{N}\} \\ \mathit{Input} &:= \{\mathsf{ref} : \mathit{Ref}, \mathsf{redeemer} : \mathit{DATA}\} \end{aligned}$$

Transactions consume such references and produce new outputs locked by validators:

$$\begin{aligned} T &:= \{\mathsf{inputs} : [\mathit{Input}],\ \mathsf{outputs} : [\mathit{Output}]\} \\ L &:= \epsilon \ \mid T; L \end{aligned}$$

For the sake of clarity, we have elided some additional transaction fields and context provided to validators that do not play a significant role in our investigation; a single transaction field $forge : \mathbb{N}$ immediately gets us to Bitcoin's UTxO model [2], an extra transaction field $\mathit{datum} : \mathit{DATA}$ and a context argument to validators summarising the current spending transaction further give us the fully Extended UTxO model employed by Cardano [6] that supports fully expressive smart contracts, and generalising output values from $\mathbb{N}$ to mappings of currencies to $\mathbb{N}$ further enables native tokens and multi-currency support [8, 7].

The overall state of the ledger is a set of unspent transaction outputs (UTxOs), modelled as a finite map from output references to funds locked by validator scripts:

$$S := \mathit{Ref} \mapsto \mathit{Output}$$

We again treat finite maps as functions from keys to values; we write $k \in \sigma$ when map σ contains a mapping for reference k, $\sigma \setminus ks$ to remove a set of keys ks in a given map σ, and $\sigma \uplus \sigma'$ for the *disjoint union*.

4.1 Denotational semantics

In the previous section, a transaction could fail if participants try to transfer more funds than they have in their account. In the UTxO setting, transactions are only valid under certain conditions. Given transaction t and state σ, t is valid in σ iff *all* the following criteria are met:

- **referenced outputs are unspent in σ**:

$$\forall (i \in t.\mathsf{inputs}).\ i.\mathsf{ref} \in \sigma$$

- **there is no double spending**:

$$\forall (i,\ j \in t.\mathsf{inputs}).\ i.\mathsf{ref} \neq j.\mathsf{ref}$$

- **value is preserved**:

$$\sum_{i \in t.\mathsf{inputs}} \sigma(i.\mathsf{ref}) = \sum_{o \in t.\mathsf{outputs}} o.\mathsf{value}$$

- **all inputs validate**:

$$\forall (i \in t.\mathsf{inputs}).\ \sigma(i.\mathsf{ref}).\mathsf{validator}(i.\mathsf{redeemer}) = \mathsf{true}$$

Apart from the different validity checks, the only other difference with the previous semantics lies in the denotation of a single transaction in d. Instead of updating account balances, it instead removes all previous UTxOs consumed by the transaction's inputs and then inserts new UTxOs for each of its outputs:

$$d : T \to S \to S$$
$$d(t)(\sigma) = \sigma \setminus \{i.\mathsf{ref} \mid i \in t.\mathsf{inputs}\} \uplus \{t^{\#}_{k} \mapsto o \mid t.\mathsf{outputs}[k] = o\}$$

Now we can give the denotational semantics as before, namely a (partial) state transition between valid states:

$$\begin{aligned} [\![\_]\!] &: L \to S \to \mathit{Maybe}\ S \\ [\![\epsilon]\!] &= \mathsf{just} \\ [\![t;l]\!] &= d'(t) \ggg [\![l]\!] \end{aligned} \qquad \begin{aligned} d' &: T \to S \to \mathit{Maybe}\ S \\ d'(t)(\sigma) &= \begin{cases} \mathsf{just}\ d(t)(\sigma) & \text{if } t \text{ valid in } \sigma \\ \mathsf{nothing} & \text{otherwise} \end{cases} \end{aligned}$$

We can reuse the previous operational and axiomatic semantics that we saw for account-based ledgers, using the new state transition function, d, as well as the more involved check that validates a potential transaction, as outlined above.

4.2 Separation Logic

So far it has been straightforward to extend our results from the previous sections to UTxO-based blockchains: once we have the denotation of a single transaction, the semantics of a ledger is simply the composition of its constituent transactions. When we attempt to define a separation logic for the UTxO model, however, we encounter a new problem.

The UTxO model refers to existing outputs *by name*, that is, using the hash of the enclosing transaction. In the account-based ledgers from the previous sections, funds are transferred directly *by value*. This allowed us to split and combine the finite maps, $\sigma_1 \oplus \sigma_2$, that associate each participant with their available funds. In the UTxO situation, however, funds are locked by a validator script and must be consumed as a whole: we cannot readily split and combine funds in the same way as we saw previously. Therefore, predicates such as $t_3^\# \mapsto v * t_3^\# \mapsto v'$ no longer make sense, since the third output of transaction t can only be spent once. Consequently, our separating conjunction has to be restricted only to *disjoint* fragments of the global state, as is typical in separation logics reasoning about mutable memory:

$$(P * Q)(\sigma) := \exists \sigma_1.\ \exists \sigma_2.\ P(\sigma_1)\ \wedge\ Q(\sigma_2)\ \wedge\ \sigma = \sigma_1 \uplus \sigma_2$$

As a result, we have to extend the frame rule with a side-condition, as is typical for semantics of imperative programming languages, to ensure the predicate R is separate from the fragments modified by the ledger l:

$$\frac{\{P\}\ l\ \{Q\} \qquad l \# R}{\{P * R\}\ l\ \{Q * R\}}\ \textsc{Frame}$$

The condition $l \# R$ ensures all references in l are disjoint from the *support* of R, i.e. precisely when the validity of the predicate does not depend on parts of the state that the ledger's transactions mutate:

$$l \# R := \forall s. R(s) \leftrightarrow R(s \setminus \{i.\mathsf{ref} \mid i \in l.\mathsf{inputs}\})$$

Similarly, the parallel rule also needs to be restricted to only *disjoint interleavings*:

$$\frac{\{\ P_1\ \}\quad l_1\quad \{\ Q_1\ \} \qquad \{\ P_2\ \}\quad l_2\quad \{\ Q_2\ \} \qquad l_1 \# P_2 \qquad l_2 \# P_1}{\{\ P_1 * P_2\ \}\quad l_1 \,||\, l_2\quad \{\ Q_1 * Q_2\ \}}\ \textsc{Par}$$

This is the point where our development has been rendered *non-compositional*, since we have to constantly reason about the dependency of the small part we are focusing on with respect to the entirety of the existing ledger.

▶ Remark 9. One might wonder whether similar issues apply in the case of non-UTxO, account-based blockchains like Ethereum. There, the same issue with hash-based referencing applies, which will naturally also appear in the form of disjointness conditions, hence losing the compositional properties we are after. Moreover, we believe the underlying execution model, based on global mutate state, will be even less compositional and inhibit modular reasoning for orthogonal reasons. This further motivates our interest in the UTxO model and its variants, culminating in the proposed solution we show next.

5 Abstract UTxO

Another way to approach the problems with a separation logic for UTxO ledgers identified in the previous section would be to tweak the UTxO model itself to make it easy to accommodate compositional reasoning techniques.

Rather than give up on UTxO entirely, we instead define a variation of UTxO, abstracting away the hash-based references we saw previously. Rather than refer to unspent outputs by their *name*, we refer to them by *value*:

$$\mathit{Ref} := \mathit{Output}$$

The rest of the basic definitions remain intact, except that the state of the ledger can no longer be represented by a map from references to outputs, but rather as a *bag* of outputs, since we need to keep track of duplicates which are now perfectly fine.

$$S := \mathit{Bag}\langle \mathit{Output} \rangle$$

These bags, also known as *multi-sets*, can again be viewed as functions mapping outputs to quantities ($\mathbb{N}$), so we will reuse the notation from the previous sections; now $\sigma(k)$ returns how many times an element k occurs in bag σ. If we furthermore exploit the monoidal nature of the number of occurrences, we get access to an *overlapping union* operator that performs pointwise addition, as well as a notion of *bag inclusion*:

$$(\sigma_1 \oplus \sigma_2)(p) := \sigma_1(p) + \sigma_2(p)$$
$$\sigma \subseteq \tau := \forall x . \sigma(x) \leq \tau(x)$$

We call the resulting ledger model *Abstract UTxO* (AUTxO), given that it abstracts away the ordering on transaction outputs imposed by the UTxO model.

5.1 Denotational semantics

To define a denotational semantics for AUTxO, we need to revise the validity conditions that check a transaction t given a current ledger state σ, and redefine the state transition function, d. Validity of abstract transactions closely follows the criteria we set previously in Section 4.1, except that inputs now only contain a monetary value locked by a validator (i.e. they are no longer represented as unspent outputs attached to previous transactions), so we need only check that the current bag of unspent values contains at least the consumed amount, and there is no longer a requirement to check for duplicate references, since it is now perfectly sensible to have two inputs that carry the same value. Formally, t is valid in σ iff *all* the following conditions hold:

- **there are sufficient funds in σ**:

$$t.\mathsf{inputs} \subseteq \sigma$$

- **value is preserved**:

$$\sum_{i \in t.\mathsf{inputs}} i.\mathsf{ref} = \sum_{o \in t.\mathsf{outputs}} o.\mathsf{value}$$

- **all inputs validate**:

$$\forall (i \in t.\mathsf{inputs}).\ i.\mathsf{ref}.\mathsf{validator}(i.\mathsf{redeemer}) = \mathsf{true}$$

Notice that value preservation has become significantly simpler to formulate in this more abstract model, since we no longer need to query the value of a referenced output from the current state σ; the reference *is* the value!

The denotational semantics of a single transaction removes previously unspent transaction outputs, replacing them with the outputs of the new transaction:

$$d : T \to S \to S$$
$$d(t)(\sigma) = \sigma \setminus \{i.\mathsf{ref} \mid i \in t.\mathsf{inputs}\} \oplus \{o.\mathsf{value} \mid o \in t.\mathsf{outputs}\}$$

We derive the rest of the scaffolding to sequentially derive the denotation of a whole ledger exactly as before:

$$[\![\_]\!] : L \to S \to \mathit{Maybe}\ S \qquad d' : T \to S \to \mathit{Maybe}\ S$$
$$[\![\epsilon]\!] = \mathsf{just} \qquad d'(t)(\sigma) = \begin{cases} \mathsf{just}\ d(t)(\sigma) & \text{if } t \text{ valid in } \sigma \\ \mathsf{nothing} & \text{otherwise} \end{cases}$$
$$[\![t; l]\!] = d'(t) \ggg [\![l]\!]$$

The operational and axiomatic semantics do not change in any way, except that they work on predicates over bags of outputs instead of maps from references to outputs.

5.2 Separation Logic

We can finally regain modularity for our separation logic, thanks to transaction inputs in AUTxO referring to existing outputs *by value*. In particular, we can define the separating conjunction as follows:

$$(P * Q)(\sigma) := \exists \sigma_1.\ \exists \sigma_2.\ P(\sigma_1)\ \wedge\ Q(\sigma_2)\ \wedge\ \sigma = \sigma_1 \oplus \sigma_2$$

where we utilise the monoidal composition of two bags that may overlap, regardless of whether they are disjoint or not.

Note that the elements in our case are pairs of a validator function and available funds. While previously we were using the monoidal action on the monetary funds, we now just compose at the level of bag occurrences leaving the value intact. That means that if the same validator locks two values v and v', we cannot deduce that it locks $v + v'$—a property that the simple account-based ledgers did support. We sketch a further abstraction that accounts for this deeper composition in Section 6.2 by inserting silent transactions that redistribute funds, but leave a formal investigation for future work.

The resulting inference rules are identical to the ones presented previously for account-based ledgers in Section 3, where we now use the monoidal actions on bags of values instead of the pointwise sum on finite maps.

$$\frac{\{P\}\ l\ \{Q\}}{\{P * R\}\ l\ \{Q * R\}}\ \textsc{frame} \qquad \frac{\{\ P_1\ \}\quad l_1\quad \{\ Q_1\ \} \qquad \{\ P_2\ \}\quad l_2\quad \{\ Q_2\ \}}{\{\ P_1 * P_2\ \}\quad l_1 \,||\, l_2\quad \{\ Q_1 * Q_2\ \}}\ \textsc{par}$$

In particular, the PAR rule enables us to reason about separate parts of the ledger independently. We can now prove properties of at the AUTxO level in a modular fashion, and have confidence that they also hold in an equivalent UTxO ledger where outputs are ordered and hash references are explicitly by name.

Example use case

In order to see how our emphasis on tracing the flow of values leads to a modular approach that is flexible enough to cover realistic problems, let us go through the scenario of trying to formally verify a smart contract running on top of a UTxO ledger.

First, the contract under investigation might have two completely distinct flows of value that you would like to reason about in isolation. Alternatively, you might want to track the total value carried by the contract and, say, prove that is remains constant or within some range. Zooming out even further, you might want to track funds running across multiple contracts and make sure certain conditions are met that depend on how these contracts interact.

Our approach readily adapts to all these levels of granularity, since they all share the same monoidal core that allows us to split funds, which in turn enable modular reasoning. Therefore, we believe our approach provides robust foundations for smart contract verification in general, starting from the primitive level of the ledger while being flexible enough to scale to more realistic settings involving smart contracts.

5.3 Sound abstraction

The relation between AUTxO and UTxO is not yet satisfying, as we need some kind of *full abstraction* [17] result that lets us conduct compositional proofs at the *abstract* ($\mathbb{A}$) level which then translate to properties about an actual *concrete* ($\mathbb{C}$) ledger. One can informally see that all properties that do not observe the implementation details of the concrete model (i.e. the order of transaction outputs and their specific hashes), should be derivable from their abstract counterparts.

To formalise the intuition above, we first define the abstraction of a concrete state as viewing its *range* as a bag:

$$abs^S : \mathbb{A}.S \to \mathbb{C}.S$$
$$abs^S(\sigma)G = \{\sigma(k) | k \in \sigma\}$$

We can then build up abstraction functions for *valid* transactions (abs^T) and ledgers (abs^L), where we resolve the actual outputs that references consume. Most importantly, UTxO validity is transformed into AUTxO validity, making it possible to then relate their respective denotational semantics.

▶ **Lemma 10.** *Given a UTxO transaction t valid in σ, applying the UTxO semantics and then abstracting the resulting state is the same as first abstracting the state and then running the AUTxO semantics on the abstracted transaction:*

$$\frac{t \text{ valid in } \sigma \qquad \mathbb{C}[\![\ t\ ]\!](\sigma) = \mathsf{just}\ \tau}{\mathbb{A}[\![\ abs^T(t)\ ]\!](abs^S(\sigma)) = \mathsf{just}\ abs^S(\tau)}$$

This naturally generalises to ledgers, where a ledger l is considered valid in σ when each transaction in sequence remains valid starting from σ:

$$\frac{l \text{ valid in } \sigma \qquad \mathbb{C}[\![\ l\ ]\!](\sigma) = \mathsf{just}\ \tau}{\mathbb{A}[\![\ abs^L(l)\ ]\!](abs^S(\sigma)) = \mathsf{just}\ abs^S(\tau)}$$

Finally, we can prove soundness of our abstract model with respect to the UTxO model, at least for properties that do not observe implementation details.

▶ **Theorem 11.** *Given a UTxO ledger l valid in some initial concrete state σ, we can discharge a concrete Hoare triple with abstract pre-/post-conditions by proving its abstract counterpart:*

$$\frac{\mathbb{A}\{P\}\ abs^L(l)\ \{Q\} \qquad l \text{ valid in } \sigma}{\mathbb{C}\{P \circ abs^S\}\ l\ \{Q \circ abs^S\}}\ \textsc{soundness}$$

where both Hoare triples have been implicitly instantiated to the state σ that is universally quantified at the outermost level.

This means it is *sound* to conduct modular roofs on the abstract level; the equivalent statement on concrete ledgers will also hold. Note that our abstract model is not *complete*, since we can only cover abstract state predicates of the form $P \circ abs^S$, thus we cannot hope to prove a *full abstraction* result.

▶ Remark 12. While making this formal connection to UTxO is important to make sure our results readily transfer to existing blockchains, there is still something to be said about AUTxO in isolation, as an alternative underlying model for new blockchains. From the pragmatic lens of blockchain validation, AUTxO seems to allow far more liberal transaction sequences than UTxO, where you would need to re-submit transactions to resolve conflicts. This contention bottleneck heavily influences how many transactions can be validated in parallel, hence a blockchain built on AUTxO might allow higher transaction throughput. Although an experimental validation of this claim still remains, we note that there have been some initial experiments that explore similar relaxations of the UTxO model [18], as employed in the IOTA distributed ledger [19].

6 Discussion

6.1 Related Work

Blockchain Theory

The entire line of research on UTxO-based ledgers starts from Bitcoin [20, 2, 3], later extended in the Cardano blockchain to *Extended UTxO* (EUTxO) [29] so as to enable the full expressivity of smart contracts. Thankfully, there are mechanised formalisations for the meta-theory of both Bitcoin [27] and EUTxO [6, 7], all of which however suffer from a monolithic approach, where the only reasoning provided is based on induction over the whole history of the ledger. We believe that the approach present here does not contradict in any way with the basic assumptions in these formulations; we expect it can be readily deployed in each respective setting. One experiment for ledger modularity in the EUTxO setting [16] led to the inevitable non-compositional notion of separation we addressed here.

On the Bitcoin side, there is a mechanised program logic for reasoning about Bitcoin's script language [1] based on *predicate transformer* semantics [11]; the striking similarity with our work lies in the use of weakest preconditions to model access control, which is essentially what we use to define the STEP rule for our Hoare logic, i.e. in the calculated pre-condition $\uparrow P \circ d'(t)$.

Alternative approaches to solving the modularity problem include the algebraic model of *Idealised UTxO* [13] where ledgers are generalised to *ledger chunks* with open-ended inputs rather than an inductive structure and naming is handled using *nominal techniques* [12], as well as the categorical treatment of Nester's material history [21, 22] where one reasons about resources and ownership in the intuitive graphical language of *symmetric monoidal categories* [25, 9].

In the non-UTxO setting, where the underlying ledger follows the account-based variant of models led by Ethereum, an approach based on ownership influenced by the program logic literature is used for implementing *sharding*—a technique for scaling up transaction validation across multiple nodes—for the Zilliqa blockchain [23].

Concurrency Theory

Analogies between the study of blockchains and classic concurrent or distributed computing have already been noted by experts in the latter that subsequently became involved in blockchain research [14, 26].

One particular separation logic in existing work bears close resemblance to the one developed in this paper, namely that of *fractional permissions* [4, 10] for handling partial ownership of resources. Similarly to our work, separating conjunction does not enforce disjointness but admits some level of overlap, in this case used to model scenarios in parallel programming with many readers and a single writer, for instance.

Last but not least, we note our initial inspiration from previous work that applied the idea of separation logic on something other than computer programs mutating memory, namely in the domain of version control systems [28].

6.2 Future Work

Decompositionality

One aspect that fails to translate to the UTxO setting is the treatment of separated conjunctions as arithmetic formulas, where equivalences such as $A \mapsto 2 \approx A \mapsto 1 * A \mapsto 1$ hold by definition. We can refer to this property as *decompositionality*, since it lets us automatically decompose a large resource into its constituent parts.

This is simply not true in the UTxO model, as noted in Section 4.2, since we still need to consume previous outputs as a whole, whose funds are predetermined by the enclosing transaction. However, we could get around this by *silently* inserting transactions that perform the necessary split/merge operations, thus allowing us to reason at an even more abstract level *modulo* transactions that merely redistribute funds. Accounting for such silent steps in the (A)UTxO model is a topic for further work.

Connection with existing separation logics

Although our approach draws heavily from the rich literature of separation logic in programming languages, we have not yet made a formal connection with our definitions and various notions of separation. One way to accomplish that is to instantiate an existing framework that supports various kinds of separation logics. A suitable candidate for that would be *Abstract Separation Logic* [5], where we could prove that the various ledger states across our development actually obey the interface and corresponding laws of *separation algebras*.

A more practically oriented course of action would be to directly implement our proposal in the Iris framework [15] which supports a wide variety of separation logics in the Coq proof assistant. Given how extensible Iris is and the relative simplicity of our program logics, the transliteration of our Agda formalisation to Coq/Iris should be straightforward and quickly give us a practical verification tool.

6.3 Conclusion

We have presented a compositional approach to reasoning about UTxO ledgers, made possible by exploiting the analogy between programs mutating memory and transactions transferring funds between accounts. The key methodological insight is that the ledger can be viewed as a (restricted) programming language, thus opening up the possibility of developing program logics to reason about (sequences of) transactions. We have demonstrated how ideas from separation logic in particular provide the modularity principle to reason about ledger fragments independently of one another.

In the future, this work may lay the foundations for scaling up verification of complex UTxO-based smart contracts, offering multiple levels of abstraction or even multiple program logics depending on the desired level of modularity and detail. Reasoning about monolithic ledgers cannot scale without modular reasoning principles—this paper presents a first step in that direction.

References

1 Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer. Verification of Bitcoin script in Agda using weakest preconditions for access control. In Henning Basold, Jesper Cockx, and Silvia Ghilezan, editors, *27th International Conference on Types for Proofs and Programs, TYPES 2021, June 14-18, 2021, Leiden, The Netherlands (Virtual Conference)*, volume 239 of *LIPIcs*, pages 1:1–1:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.TYPES.2021.1`.

2 Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of Bitcoin transactions. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 541–560. Springer, 2018. `doi:10.1007/978-3-662-58387-6\_29`.

3 Massimo Bartoletti and Roberto Zunino. Formal models of Bitcoin contracts: A survey. *Frontiers Blockchain*, 2:8, 2019. doi:10.3389/fbloc.2019.00008.

4 John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. doi:10.1007/3-540-44898-5_4.

5 Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and Abstract Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 366–378. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.30.

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

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

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

9 Bob Coecke, Tobias Fritz, and Robert W. Spekkens. A mathematical theory of resources. *Inf. Comput.*, 250:59–86, 2016. doi:10.1016/j.ic.2016.02.008.

10 Thibault Dardinier, Peter Müller, and Alexander J. Summers. Fractional resources in unbounded separation logic. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1066–1092, 2022. doi:10.1145/3563326.

11 Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

12 Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002. doi:10.1007/s001650200016.

13 Murdoch James Gabbay. Algebras of UTxO blockchains. *Math. Struct. Comput. Sci.*, 31(9):1034–1089, 2021. doi:10.1017/S0960129521000438.

14 Maurice Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2):78–85, 2019. doi:10.1145/3209623.

15 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order Concurrent Separation Logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

16 Orestis Melkonian, Wouter Swierstra, and Manuel M. T. Chakravarty. Formal investigation of the Extended UTxO model. In *4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe)*, 2019. URL: https://omelkonian.github.io/data/publications/formal-utxo.pdf.

17 Robin Milner. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977. doi:10.1016/0304-3975(77)90053-6.

18 Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and Hans Moog. Reality-based UTXO ledger. *CoRR*, abs/2205.01345, 2022. arXiv:2205.01345, doi:10.48550/arXiv.2205.01345.

19 Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and Hans Moog. Tangle 2.0 leaderless nakamoto consensus on the heaviest DAG. *IEEE Access*, 10:105807–105842, 2022. doi:10.1109/ACCESS.2022.3211422.

20 S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/en/bitcoin-paper, October 2008.

21 Chad Nester. A foundation for ledger structures. In Emmanuelle Anceaume, Christophe Bisière, Matthieu Bouvard, Quentin Bramas, and Catherine Casamatta, editors, *2nd International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2020, October 26-27, 2020, Toulouse, France*, volume 82 of *OASIcs*, pages 7:1–7:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/OASIcs.Tokenomics.2020.7.

22 Chad Nester. The structure of concurrent process histories. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2021. doi:10.1007/978-3-030-78142-2_13.

23 George Pîrlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with ownership and commutativity analysis. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1327–1341. ACM, 2021. doi:10.1145/3453483.3454112.

24 John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

25 Peter Selinger. A survey of graphical languages for monoidal categories. *New structures for physics*, pages 289–355, 2011.

26 Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, volume 10323 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2017. doi:10.1007/978-3-319-70278-0_30.

27 Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. URL: http://arxiv.org/abs/1804.06398, arXiv:1804.06398.

28 Wouter Swierstra and Andres Löh. The semantics of version control. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 43–54. ACM, 2014. doi:10.1145/2661136.2661137.

29 Joachim Zahnentferner. An abstract model of UTxO-based cryptocurrencies with scripts. *IACR Cryptol. ePrint Arch.*, page 469, 2018. URL: https://eprint.iacr.org/2018/469.

A Examples

Here we present some example derivations in the various logics developed throughout the paper, in order to demonstrate the relative strengths and weaknesses of each approach.

Apart from the rules presented in the main body of the paper, we will also make use of the following auxiliary lemmas:

$$\frac{}{\{\ \mathrm{A} \mapsto \mathrm{n}\ \}\ \mathrm{A} \xrightarrow{n} \mathrm{B};\ \mathrm{B} \xrightarrow{n} \mathrm{A}\ \{\ \mathrm{A} \mapsto \mathrm{n}\ \}}\ \textsc{cancel} \qquad \frac{}{\{P * Q\} \approx \{Q * P\}}\ \textsc{swap}$$

Simple example using FRAME

The FRAME rule lets us focus on a small part of a larger separating conjunction and apply the rule locally:

$$\begin{array}{ll}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \\
\quad A \xrightarrow{1} B & \dashv \textsc{frame}(C \mapsto 0 * D \mapsto 1, \textsc{send}) \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 0 * D \mapsto 1\} & \\
\quad \approx & \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 0 * B \mapsto 1\} & \\
\quad D \xrightarrow{1} C & \dashv \textsc{frame}(A \mapsto 0 * B \mapsto 1, \textsc{send} \circ \textsc{swap}) \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 0 * B \mapsto 1\} & \\
\quad \approx & \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 1 * D \mapsto 0\} & \\
\quad B \xrightarrow{1} A & \dashv \textsc{frame}(C \mapsto 1 * D \mapsto 0, \textsc{send} \circ \textsc{swap}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 1 * D \mapsto 0\} & \\
\quad \approx & \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 1 * B \mapsto 0\} & \\
\quad C \xrightarrow{1} D & \dashv \textsc{frame}(A \mapsto 1 * B \mapsto 0, \textsc{send}) \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 1 * B \mapsto 0\} & \\
\quad \approx & \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \blacktriangleleft
\end{array}$$

Simple example using PAR

Notice how in the previous example the first and third transaction only involve A and B, while the other two only involve C and D. That's why we can do better using the PAR rule, where we assemble a compositional proof from smaller proofs:

$$\begin{array}{ll}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \\
\quad (A \xrightarrow{1} B; B \xrightarrow{1} A) | (D \xrightarrow{1} C; C \xrightarrow{1} D) & \\
\quad \ni (A \xrightarrow{1} B; D \xrightarrow{1} C; B \xrightarrow{1} A; C \xrightarrow{1} D) & \dashv \textsc{par}(H^{AB}, H^{CD}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \blacktriangleleft
\end{array}$$

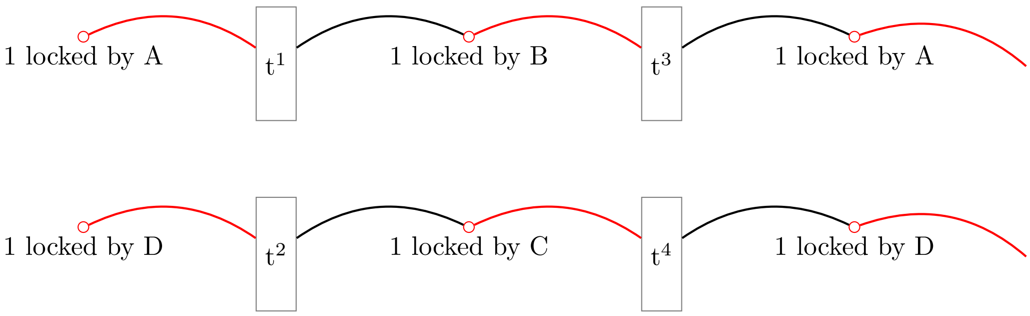
where

$$H^{AB} := \quad \begin{array}{ll} \{A \mapsto 1 * B \mapsto 0\} & \\ \quad A \xrightarrow{1} B ; B \xrightarrow{1} A & \dashv \textsc{cancel} \\ \{A \mapsto 1 * B \mapsto 0\} & \blacktriangleleft \end{array}$$

$$H^{CD} := \quad \begin{array}{ll} \{C \mapsto 0 * D \mapsto 1\} & \\ \quad D \xrightarrow{1} C ; C \xrightarrow{1} D & \dashv \textsc{cancel} \\ \{C \mapsto 0 * D \mapsto 1\} & \blacktriangleleft \end{array}$$

UTxO example using FRAME

We can conduct similar proofs for UTxO-based ledgers, although our predicates now have to also include references to previous transactions. We denote singleton predicates by $t_i \mapsto v\ at\ p$, where we require a single UTxO to be unspent in the i-th output of transaction t, holding a value v locked by validator function p.



$$\begin{array}{ll}
\{t_0^0 \mapsto 1\ at\ A * t_1^0 \mapsto 1\ at\ D\} & \\
\quad t^1 & \dashv \textsc{frame}(t_1^0 \mapsto 1\ at\ D, \dots, \textsc{send}) \\
\{t_0^1 \mapsto 1\ \ at\ \ B * t_1^0 \mapsto 1\ at\ D\} & \\
\quad \approx & \\
\{t_1^0 \mapsto 1\ at\ D * t_0^1 \mapsto 1\ at\ B\} & \\
\quad t^2 & \dashv \textsc{frame}(t_0^1 \mapsto 1\ at\ B, \dots, \textsc{send}) \\
\{t_0^2 \mapsto 1\ at\ C * t_0^1 \mapsto 1\ at\ B\} & \\
\quad \approx & \\
\{t_0^1 \mapsto 1\ at\ B * t_0^2 \mapsto 1\ at\ C\} & \\
\quad t^3 & \dashv \textsc{frame}(t_0^2 \mapsto 1\ at\ C, \dots, \textsc{send}) \\
\{t_0^3 \mapsto 1\ at\ A * t_0^2 \mapsto 1\ at\ C\} & \\
\quad \approx & \\
\{t_0^2 \mapsto 1\ at\ C * t_0^3 \mapsto 1\ at\ A\} & \\
\quad t^4 & \dashv \textsc{frame}(t_0^3 \mapsto 1\ at\ A, \dots, \textsc{send}) \\
\{t_0^4 \mapsto 1\ at\ D * t_0^3 \mapsto 1\ at\ A\} & \\
\quad \approx & \\
\{t_0^3 \mapsto 1\ at\ A * t_0^4 \mapsto 1\ at\ D & \blacktriangleleft
\end{array}$$

Notice the additional proof obligations marked with ... all over the place, which require tedious reasoning about disjointness.

UTxO example using PAR

The PAR can slightly improve the situation by composing smaller proofs, but is no longer a scalable solution since we still need to provide evidence that the interleaved ledgers are disjoint:

$$
\begin{array}{ll}
\{t_0^0 \mapsto 1 \; at \; A * t_1^0 \mapsto 1 \; at \; D\} & \\
\quad t^1 \ldots t^4 & \dashv \textsc{par}(\ldots, H^{AB}, H^{CD}) \\
\{t_0^3 \mapsto 1 \; at \; A * t_0^4 \mapsto 1 \; at \; D\} & \blacktriangleleft
\end{array}
$$

where

$H^{AB} :=$

$$
\begin{array}{ll}
\{t_0^0 \mapsto 1 \; at \; A\} & \\
\quad t^1 & \dashv \textsc{send} \\
\{t_0^1 \mapsto 1 \; at \; B\} & \\
\quad t^3 & \dashv \textsc{send} \\
\{t_0^3 \mapsto 1 \; at \; A\} & \blacktriangleleft
\end{array}
$$

$H^{CD} :=$

$$
\begin{array}{ll}
\{t_1^0 \mapsto 1 \; at \; D\} & \\
\quad t^1 & \dashv \textsc{send} \\
\{t_0^2 \mapsto 1 \; at \; C\} & \\
\quad t^4 & \dashv \textsc{send} \\
\{t_0^4 \mapsto 1 \; at \; D & \blacktriangleleft
\end{array}
$$

AUTxO example using FRAME

In the case of AUTxO, we can once again think of validators as a replacement for participant identifiers A, B, C, D, assuming transactions $t_1 \ldots t_4$ that have the corresponding structure that enacts the transfers we defined in the initial non-blockchain example.

Unsurprisingly, the Hoare conditions remain identical and only the enclosed transactions change from the initial proof:

$$
\begin{array}{ll}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \\
\quad t^1 & \dashv \textsc{frame}(C \mapsto 0 * D \mapsto 1, \textsc{send}) \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 0 * D \mapsto 1\} & \\
\quad \approx & \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 0 * B \mapsto 1\} & \\
\quad t^2 & \dashv \textsc{frame}(A \mapsto 0 * B \mapsto 1, \textsc{send}) \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 0 * B \mapsto 1\} & \\
\quad \approx & \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 1 * D \mapsto 0\} & \\
\quad t^3 & \dashv \textsc{frame}(C \mapsto 1 * D \mapsto 0, \textsc{send}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 1 * D \mapsto 0\} & \\
\quad \approx & \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 1 * B \mapsto 0\} & \\
\quad t^4 & \dashv \textsc{frame}(A \mapsto 1 * B \mapsto 0, \textsc{send}) \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 1 * B \mapsto 0\} & \\
\quad \approx & \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \blacktriangleleft
\end{array}
$$

AUTxO example using PAR

We finally demonstrate how we have regained compositionality in the AUTxO setting:

$$\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\}$$
$$t^1 \dots t^4 \qquad \dashv \textsc{par}(H^{AB}, H^{CD})$$
$$\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \qquad \blacktriangleleft$$

where

$H^{AB} :=$

$$\{A \mapsto 1 * B \mapsto 0\}$$
$$t^1 \qquad \dashv \textsc{send}$$
$$\{A \mapsto 0 * B \mapsto 1\}$$
$$t^3 \qquad \dashv \textsc{send}$$
$$\{A \mapsto 1 * B \mapsto 0\} \qquad \blacktriangleleft$$

$H^{CD} :=$

$$\{C \mapsto 0 * D \mapsto 1\}$$
$$t^2 \qquad \dashv \textsc{send}$$
$$\{C \mapsto 1 * D \mapsto 0\}$$
$$t^4 \qquad \dashv \textsc{send}$$
$$\{C \mapsto 0 * D \mapsto 1\} \qquad \blacktriangleleft$$