Program Logics for Ledgers

Orestis Melkonian
Input Output, London, UK

Wouter Swierstra
Utrecht University, The Netherlands

James Chapman
Input Output, London, UK

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 is less clear how to reason effectively about such ever-growing linear data structures. This paper demonstrates how distributed ledgers may be viewed as *computer programs*, that, when executed, transfer funds between various parties. As a result, familiar program logics, such as Hoare logic, are applied in a novel setting. Borrowing ideas from concurrent separation logic, this enables modular reasoning principles over arbitrary fragments of any ledger. All of our results have been mechanised in the Agda proof assistant.

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

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

Digital Object Identifier 10.4230/OASIcs.FMBC.2025.10

Supplementary Material
Software (Zenodo archive): `https://doi.org/10.5281/zenodo.15097592` [22]
Software (Source Code): `https://github.com/omelkonian/hoare-ledgers` [21]
archived at `swh:1:dir:fe2bce9b8779645c5a992156ea43604432ccc496`

Acknowledgements We would like to thank Philip Wadler, Marko Doko, Murdoch J. Gabbay, Manuel M.T. Chakravarty, Brian Campbell, and James Wood for their valuable feedback.

1 Introduction

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

To illustrate this point, consider the following simplified example, giving two simple lists of transactions, also known as *ledgers*:

```
Alice pays Bob 5;          Dana pays Bob 5;
Carroll pays Dana 3;       Alice pays Dana 3;
Dana pays Alice 2;         Carroll pays Dana 3;
```

Are these transactions "the same"? Although a simple calculation shows that they have the same net effect, one of the two might fail: for instance, if Dana has less than five funds available these two behave differently. Now suppose that these transactions are interleaved with an arbitrary number of other transactions, some even involving the same accounts. Can we still say anything about how they might behave? For many financial applications, such as fraud detection or ledger compression, it is crucial to study fragments of the ledger in isolation to ensure analyses remain computationally tractable.

This paper explores the application of program language semantics to the domain of financial ledgers, such as those underlying cryptocurrencies. Starting from simple ledger based accounts, we extend our study to cover the Unspent Transaction Outputs model (UTxO), underlying modern cryptocurrencies including Bitcoin [27] and Cardano [9]. Crucially, we explore how to employ *separation logic* 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 demonstrate how *the financial transactions stored in a ledger form a simple programming language.* We present denotational and axiomatic semantics of account-based ledgers, together with a *separation logic* that enables modular reasoning over ledger fragments (Section 2). The separation logic that 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 can be given for blockchain ledgers (Section 3), in particular ones based on the UTxO model. Separation logic, however, poses more of a challenge as the hash-based nature of UTxO 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 4). The resulting logic enables us to reason locally and safely about a limited number of transactions, sprinkled arbitrarily throughout a larger ledger.

It is important to emphasise that we do *not* study individual smart contracts or other such programs that might manipulate the ledger; the focus of this paper is the meaning of the ledger *as a whole.*

All the definitions and theorems presented in this paper have been mechanised in Agda [22]:

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

We use mathematical notation rather than "literate programming" style, but still provide hyperlinks to the actual mechanisation indicated by the Agda logo (). The proofs themselves are typically quite simple – the hard work is in finding the definitions that make them so.

While the semantics and logics may be unsurprising to program language experts, we feel their application in a novel setting is ample reason for excitement. Just as previous work has shown how complicated financial contracts are built from simple functional combinators [30], this work aims to discover mathematical structure where none is apparent.

2 Account-based ledgers [`ValueSepExact.Main`]

To start things off, we give a formal definition of the syntax and semantics of a simple account-based 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 model the state of all the participants' accounts as a (finite) map, mapping each participant to their current balance:

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

Note that this model does now allow negative account balances; one could generalise this to any fixed bound other than zero to model overdraft.

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. As we saw in the introduction, a *ledger* records the history of transfers between accounts. We view such a ledger as a *program*, describing updates to the state of the accounts modelled by S. The abstract syntax of our ledger is defined as:

$$\begin{aligned} T &:= \mathcal{P} \xrightarrow{n} \mathcal{P} \\ L &:= \epsilon \ \mid T; L \end{aligned}$$

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 [`ValueSepExact.Ledger`]

We give the denotational semantics of a ledger by mapping L to a function of type $S \to \mathit{Maybe}\ S$, executing all the transactions in the ledger starting from a given state with given account balances. The optional return type is used to model the case where a transaction fails due to insufficient funds: the result is just a new state after successful execution or $\mathsf{nothing}$ to signal an error.

This semantics for ledgers is straightforward to define by iterating its transactions:

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

$$d : T \to S \to \mathit{Maybe}\ S$$

$$d(p_1 \xrightarrow{n} p_2)(\sigma) = \begin{cases} \mathsf{just}\ \lambda p. \begin{cases} \sigma(p) - n & \text{if } p = p_1 \neq p_2 \\ \sigma(p) + n & \text{if } p = p_2 \neq p_1 \\ \sigma(p) & \text{otherwise} \end{cases} & \text{if } \sigma(p_1) \geq n \\ \mathsf{nothing} & \text{otherwise} \end{cases}$$

The Kleisli arrow ($\ggg$) composes partial functions by collapsing to $\mathsf{nothing}$ when the first function fails. A transaction's semantics (d) checks the validity of each transfer and fails if not enough funds are available, otherwise updates the state accordingly. 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.

We 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* $[\![l_1 +\!\!+ l_2]\!] = [\![l_1]\!] \ggg [\![l_2]\!]$.

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, we may be interested in an arbitrary subset of the transactions that are related to a specific smart contract in the blockchain setting.

2.2 Axiomatic semantics [`ValueSepExact.HoareLogic`]

We 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.

$$\frac{}{\{P\}\ \epsilon\ \{P\}}\ \textsc{stop} \qquad\qquad \frac{\{P\}\ l\ \{Q\}}{\{\uparrow P \circ d(t)\}\ t; l\ \{Q\}}\ \textsc{step}$$

The base rule dictates that executing an empty ledger leaves the state unchanged, while the inductive step rule provides the *weakest pre-condition* by viewing the transaction as a *predicate transformer* [14].

Also note the necessary operation $\uparrow$, lifting 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, and the **strong** lifting that collapses to false upon failure. Since we wish to observe failing transactions, we opt for the *strong* version, which we prove *sound* with respect to the denotational semantics:

▶ **Theorem 2.** $\{P\}\ l\ \{Q\}$ *holds iff* $(P(\sigma) \wedge [\![l]\!](\sigma) = \mathsf{just}\ \tau)$ *implies* $Q(\tau)$ *for all* σ *and* τ.

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

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

The soundness theorem also allows us to derive a sequencing rule as a corollary of the equivalent statement about ledgers in the denotational semantics:

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

▶ Remark 3. 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 semantics. Moreover, any subsequent **derived** inference rules (e.g. APP above) are implicitly proven using either the axioms or directly appealing to their denotational counterparts.

Example specification

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

$$\{\lambda\sigma.\ \sigma(A) = 2\} \quad A \xrightarrow{1} B \quad \{\lambda\sigma.\ \sigma(A) = 1\} \quad A \xrightarrow{1} C \quad \{\lambda\sigma.\ \sigma(A) = 0\}$$

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* [32].

2.3 Separation logic [`ValueSepExact.SL`]

Both of our denotational and axiomatic semantics rely 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 memory state (i.e. the *heap* mapping variable addresses to their value) into two disjoint parts. The separating conjunction, $P * Q$, is then defined as follows:

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

Here, σ is the resulting heap of combining two smaller heaps σ_1 and σ_2 using the *disjoint union* operation ($\uplus$).

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 divide the *funds* from both maps into two distinct parts! To do so, we begin by defining the following operation of combining ledger states by pointwise addition of their funds:

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

Using this operation, we 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 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:

$$\frac{}{\{emp\}\ \epsilon\ \{emp\}}\ \textsc{Empty} \qquad \frac{\{P\}\ l_1\ \{Q\} \qquad \{Q\}\ l_2\ \{R\}}{\{P\}\ l_1 +\!\!+\ l_2\ \{R\}}\ \textsc{App}$$

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.

2.4 Concurrent separation logic [`ValueSepExact.CSL`]

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:

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

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. We refer the reader to Appendix A for example derivations.

3 UTxO [`UTxOErr.Main`]

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. The UTxO model is quite different: rather than develop a model based on accounts and transactions between them, the UTxO model focuses on "unspent funds". Such unspent funds are locked by a *validator script*. These funds can be spent by anyone, provided they can provide the *redeemer data*, that is, data mapped to true by the associated validator script:

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

Typically, such a validator script might use public-private key pairs to allow access only to the person holding the private key. This model is more general than the account-based model we have studied so far: funds might be shared by different parties that each must provide their private key to unlock the corresponding funds. The overall state of the ledger is a collection of unspent transaction outputs (UTxOs) – we will make this more precise shortly, but first need to describe how transactions work in the UTxO model.

If all unspent funds are locked by a validator script – how can we possibly move funds? In the UTxO model, each transaction consumes unspent funds from *inputs*, producing new unspent *outputs*:

$$T := \{\mathsf{inputs} : [Input],\ \mathsf{outputs} : [Output]\}$$

Each input needs to refer to the unspent output being consumed *and* provide the redeemer data required to unlock the corresponding funds:

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

There is a subtle point to consider here: how should an input refer to an unspent transaction output? This is usually done by referring to the transaction's *hash*. As each transaction produces a list of unspent transaction outputs, we also require an index into this list to refer to a specific transaction output. We write $t^{\#}_{k}$ to refer to the k-th output of the transaction t. Finally, the ledger itself consists of a list of transactions – as we have seen before.

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

For the sake of clarity, we have elided some additional fields and validator arguments that do not play a significant role in our semantics:

- adding a single transaction field `forge` : ℕ to create new currency immediately gets us to Bitcoin's UTxO model [3];

  $$T := \{\ldots,\ \mathsf{forge} : \mathbb{N}\}$$

- an additional field `datum` : *DATA* in outputs and extending validators with additional context further brings us to the Extended UTxO model employed by Cardano [9] that supports fully expressive smart contracts;

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

- generalising output values from ℕ to *token bundles* and including policies to control how to mint these tokens enables native tokens and multi-currency support [11, 10].

  $$\begin{aligned} \mathit{TokenBundle} &:= \mathit{HASH} \mapsto \mathit{HASH} \mapsto \mathbb{N} \\ T &:= \{\ldots,\ \mathsf{forge} : \mathit{TokenBundle}, \mathsf{mintingPolicies} : \ldots\} \\ \mathit{Output} &:= \{\ldots,\ \mathsf{value} : \mathit{TokenBundle}\} \end{aligned}$$

At any given point, the state records the currently unspent transaction outputs:

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

We again treat finite maps as functions from keys to values; we write $k \in \sigma$ to check for membership in the map; $\sigma \setminus ks$ to remove a set of keys; $\sigma \uplus \sigma'$ for the *disjoint union* on maps.

For some examples of transactions, see Appendix B.

3.1 Denotational semantics [`UTxOErr.Ledger`]

In the previous section, a transaction could fail due to insufficient funds. Similarly 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**:

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

- **there is no double spending**:

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

- **value is preserved**:

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

- **all inputs validate**:

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

All other parts remain identical to the previous semantics, Hoare logic rules included, except the denotation of a single transaction: instead of updating account balances, it instead removes all previous UTxOs consumed by its inputs and then inserts new UTxOs for each output:

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

3.2 Separation logic [UTxOErr.SL]

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* (i.e. the hash of the enclosing transaction), while the previous model for account-based ledger transferred funds 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. Thus our separating conjunction has to be restricted only to *disjoint* fragments of the state:

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

As a result, we have to extend the frame rule with a *disjointness* side-condition, familiar from the semantics of imperative programs that mutate memory:

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

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

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

3.3 Concurrent separation logic [UTxOErr.CSL]

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

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

This is the point in our development where we have lost the stronger *compositionality* properties that the previous semantics enjoyed. To use the frame rule to reason about UTxO-based blockchains, this side condition requires checking disjointness (see Appendix B for examples) – where previously we were free to split off arbitrary funds from the account state.

4 Abstract UTxO [ValueSepUTxO.Main]

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 where we abstract away from hash-based references and refer to unspent outputs by *value*:

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

It is important to emphasise what changes here: rather than refer to an unspent transaction output using the transaction hash (and the index in the list of outputs it produces), each input refers directly to the *values* of the unspent outputs it consumes.

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 (there can be multiple outputs with the same exact value).

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

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*:

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

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

4.1 Denotational semantics [ValueSepUTxO.Ledger]

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 3.1, 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:

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

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

- **all inputs validate**:

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

- **value is preserved**:

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

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:

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

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

4.2 Separation logic [ValueSepUTxO.SL]

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 separating conjunction:

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

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

The resulting inference rules are identical to the ones presented previously for account-based ledgers in Section 2, where we now use the monoidal actions on bags of values instead of the pointwise sum on finite maps.

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

In particular, the PAR rule enables us to reason about separate parts of the ledger independently. We can now prove properties at the AUTxO level in a modular fashion (see Appendix C for an example), and have confidence that they also hold in an equivalent UTxO ledger with hash references and ordered outputs.

Note that the elements of each bag are pairs of a validator function and available funds. While previously in the account-based ledger model, we used the monoidal action on the monetary funds – adding their monetary value when splitting the state into smaller parts. In the UTxO model, however, we cannot split locked funds: 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 will discuss this limitation in more detail later (Section 7), but leave its resolution for future work.

4.3 Sound abstraction [ConcreteToAbstract]

The relation between AUTxO and UTxO is not yet satisfying, as we need some kind of *full abstraction* [24] 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 should be derivable from their abstract counterparts. To formalise this intuition, we first define the abstraction of a concrete state as viewing its *range* as a bag:

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

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 4.** *Given a UTxO ledger l 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 ledger:*

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

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 5.** *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 proofs 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. We feel that this will not be problematic in practice: these predicates mention implementation details that arguably *should* be kept abstract.

▶ Remark 6. 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 to be done, we note that there have been some initial experiments that explore similar relaxations of the UTxO model [25], as employed in the IOTA distributed ledger [26].

5 Perspectives

Why care about semantics? Oftentimes blockchains are designed with cryptographic security guarantees in mind; the intended ledger semantics is usually clear, but as is often the case, the devil is in the details. This shows up when considering separation logic for ledgers built on the UTxO model, where cryptographic details about hashing and the model's implementation leak into the logic. Exposing the underlying mathematics, as our AUTxO model does, nails down the exact behaviour once and for all. We believe there are numerous applications that would be difficult, if not impossible, to reason about without a clearly specified formal semantics. This section sketches several such directions.

Formal verification of smart contracts

So far we have only tackled the verification of individual transactions, but there is nothing preventing us from reasoning about resources in the more elaborate setting of *smart contracts*.

Given the extensions to the UTxO model outlined in Section 3 to allow for fully-fledged smart contracts to be expressed in validator scripts, it is crucial to observe that such contracts will again manifest as transaction outputs holding a certain amount of funds. In other words, smart contracts would appear as another kind of participant in Hoare triples, and we can reason about its resources in the usual manner. Moreover, smart contracts would now appear in a *sequence* of related transactions, while possibly also interacting with other non-contract resources; the modularity of our Hoare-style framework gives us the ability to focus on exactly the subset of transactions we are interested in.

One immediate application of this method to EUTxO smart contracts would be to prevent the common issue of **double satisfaction** [38], arising when a *single* input resource is used to satisfy *multiple* constraints coming from different validators/scripts. Within our framework, the resource in question would be precisely characterised in a Hoare triple, either to rule out double satisfaction or exhibit that it occurs (in case it was deliberate).

Ledger compression

Another application of this work would be in *ledger compression*: denotation semantics give us a natural algorithm for minimising the ledger, while concurrent separation logic lets us reason under the substitution of the original ledger with the compressed one.

Inspired by the technique of *normalisation by evaluation* [5], we proceed as follows for a given ledger l that we wish to compress:

1. Compute the denotational semantics $[\![l]\!]$, i.e. a function that transforms states.
2. Read back a *minimal* ledger by observing the net change to participant accounts. The resulting compressed ledger is guaranteed to have the same semantics as the ledger we started with, therefore we can instead prove a Hoare triple for the smaller ledger.
3. Complete our reasoning by utilising the PAR rule to embed our results from step (2) in the context of the whole ledger L, which is expected to be way larger than l.

While the above works for the account-based ledger of Section 2, things become trickier in the UTxO case: the hash references appearing in the resulting state will always necessarily differ after compression. However, if we ignore the hashes and instead consider the values that they refer to, we would be able to see that the associated values indeed remain identical. In other words, this application requires reasoning *modulo hashes*, which is exactly what the AUTxO model of Section 4 provides.

It is also worth mentioning that the same technique can be used to detect fraud when “mixers” obfuscate the provenance of certain funds by moving them around between different accounts. Since the net effect of such “rings” would be zero, ledger compression would then optimise them away, enabling us to read off the fraudulent addresses that were involved.

6 Related work

Blockchain theory

The entire line of research on UTxO-based ledgers starts from Bitcoin [27, 3, 4], later extended in the Cardano blockchain to *Extended UTxO* (EUTxO) [39] so as to enable the full expressivity of smart contracts. Thankfully, there are mechanised formalisations for the meta-theory of both Bitcoin [35] and EUTxO [9, 10], 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 [23] 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 [14]; 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 in Section 2.

Alternative approaches to solving the modularity problem include the algebraic model of *Idealised UTxO* [16] where ledgers are generalised to *ledger chunks* with open-ended inputs rather than an inductive structure and naming is handled using *nominal techniques* [15], as well as the categorical treatment of Nester's material history [28, 29] where one reasons about resources and ownership in the intuitive graphical language of *symmetric monoidal categories* [33, 12].

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 [31].

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 [18, 34].

One particular separation logic in existing work bears close resemblance to the one developed in this paper, namely that of *fractional permissions* [6, 13] 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 [36].

Type Theory

The resource-oriented nature of our logics also echoes efforts in type theories that track resource usage in some way or another, for instance *Quantitative Type Theory* [20, 2], currently supported by the Idris programming language [7].

Blockchains have to do with monetary resources above everything else, thus provide a natural setting to apply these resource-oriented frameworks, and indeed there is already research supporting this in the context of the Tezos blockchain [17].

7 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* [8], where we could prove that the various ledger states across our development 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 [19] which supports a wide variety of separation logics in the Rocq proof assistant [37]. Given how extensible Iris is and the relative simplicity of our program logics, the transliteration of our Agda formalisation to Iris should be straightforward and quickly give us a practical verification tool.

8 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 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, or even offer 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 Robert Atkey. Syntax and semantics of Quantitative Type Theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. `doi:10.1145/3209108.3209189`.

3 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`.

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

5 Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 203–211. IEEE Computer Society, 1991. `doi:10.1109/LICS.1991.151645`.

6 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`.

7 Edwin C. Brady. Idris 2: Quantitative Type Theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.9`.

8 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`.

9 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`.

10 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`.

11 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`.

12 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`.

13 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.

14 Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. doi:10.1145/360933.360975.

15 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.

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

17 Christopher Goes. Compiling Quantitative Type Theory to Michelson for compile-time verification and run-time efficiency in juvix. 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 146–160. Springer, 2020. doi:10.1007/978-3-030-61467-6_10.

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

19 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.

20 Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

21 Orestis Melkonian. omelkonian/hoare-ledgers. Software, swhId: swh:1:dir:fe2bce9b8779645c5a992156ea43604432ccc496 (visited on 2025-04-14). URL: https://github.com/omelkonian/hoare-ledgers, doi:10.4230/artifacts.23004.

22 Orestis Melkonian. Agda formalisation for "Program logics for ledgers". https://github.com/omelkonian/hoare-ledgers, March 2025. doi:10.5281/zenodo.15097592.

23 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.

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

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

26 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.

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

28 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.

29 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.

30 Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). *ACM SIGPLAN Notices*, 35(9):280–292, 2000. doi:10.1145/351240.351267.

31 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.

32 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.

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

34 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.

35 Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. arXiv:1804.06398.

36 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.

37 The Coq Development Team. The Coq proof assistant (*renamed to Rocq*), September 2024. doi:10.5281/zenodo.14542673.

38 Polina Vinogradova and Orestis Melkonian. Message-passing in the Extended UTxO ledger. In *Financial Cryptography and Data Security. FC 2024 International Workshops: Voting, DeFI, WTSC, CoDecFin, Willemstad, Curaçao, March 4–8, 2024, Revised Selected Papers*, pages 150–169, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-69231-4_11.

39 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 Account-based examples [`ValueSepExact.Example`]

Example derivation using FRAME

We now introduce an example derivation that will act as a running example across the various logics we will develop throughout the paper, in order to demonstrate the relative strengths and weaknesses of each approach.

We will have two transactions between participants A and B exchanging a single unit of currency back and forth, interleaved with another two transactions of the same form but now between different participants C and D. Overall, this set of transaction leave the state of account balances unchanged after execution.

Apart from the aforementioned basic rules, we will also make use of the fact that separating conjunction is *symmetric*, in the form of the following derived rule:

$$\frac{}{\{P * Q\} \approx \{Q * P\}}\ \textsc{swap}$$

The FRAME rule lets us focus on a small part of a larger separating conjunction and apply the rule locally only on the part of the state that concerns the two participants of the transaction at hand:

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

Example derivation 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 is 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

$H^{AB} :=$

$$\begin{array}{ll}
\{A \mapsto 1 * B \mapsto 0\} & \\
\qquad A \xrightarrow{1} B & \dashv \textsc{send} \\
\{A \mapsto 0 * B \mapsto 1\} & \\
\qquad B \xrightarrow{1} A & \dashv \textsc{send} \circ \textsc{swap} \\
\{A \mapsto 1 * B \mapsto 0\} & \blacktriangleleft
\end{array}$$

$H^{CD} :=$

$$\begin{array}{ll}
\{C \mapsto 0 * D \mapsto 1\} & \\
\qquad D \xrightarrow{1} C & \dashv \textsc{send} \circ \textsc{swap} \\
\{C \mapsto 1 * D \mapsto 0\} & \\
\qquad C \xrightarrow{1} D & \dashv \textsc{send} \\
\{C \mapsto 0 * D \mapsto 1\} & \blacktriangleleft
\end{array}$$

B UTxO examples [UTxOErr.Example]

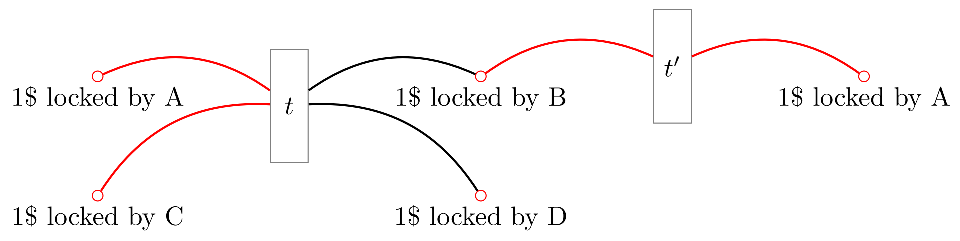
Example transaction

Let us define an example transaction in the UTxO model. First, we associate the notion of a participant – which does not inherently exist in the UTxO model – with a validator script that restricts access to the funds solely to said participant.[1]

Therefore, an output of the form "1$ locked by A" carries a single unit of currency that only A can unlock. Moreover, a single transaction can pack together multiple inputs and outputs, thus immediately performing exchange between multiple "participants".

Given a starting state where participants A and C hold a single unit of currency (in the form of two unspent outputs assumed to already exist in previous transactions), a transaction t can transfer these funds to another set of participants B and D, and another transaction t' give B's funds back to A.

This can be compactly depicted as a *directed acyclic graph*, whose *left fringe* contains dangling outputs that comprise the current state of unspent outputs and the *right fringe* corresponds to the resulting state, commonly referred to as the *UTxO set*:

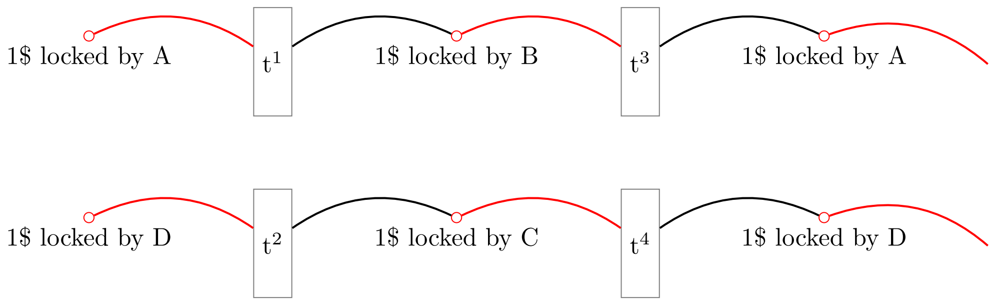


Note that the actual transaction fields are omitted in the node in the graph, since these can be easily deduced from the incoming and outgoing edges. Therefore, the above transactions denote a transition from the source state $\{ta^{\#}_i \mapsto 1\$ \text{ locked by A},\ tc^{\#}_j \mapsto 1\$ \text{ locked by C}\}$ (left fringe) to the resulting state $\{t'^{\#}_0 \mapsto 1\$ \text{ locked by A},\ t^{\#}_1 \mapsto 1\$ \text{ locked by D}\}$ (right fringe), assuming previous outputs in ta and tc holding the initial funds for A and C.

Example derivation using FRAME

Back to our running example, we can prove similar derivations 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.

[1] This could be achieved by naively having the redeemer be a password only known to the participant, or via a public-key approach where the validator script verifies a signature with respect to the participant's private key.



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

Note that this derivation now requires additional proof obligations, marked with $\ldots$, ensuring the disjointness of relevant transactions.

Example derivation 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

$$H^{AB} := \quad \{t_0^0 \mapsto 1 \ \mathit{at}\ A\} \ t^1 \dashv \text{SEND} \ \{t_0^1 \mapsto 1 \ \mathit{at}\ B\} \ t^3 \dashv \text{SEND} \ \{t_0^3 \mapsto 1 \ \mathit{at}\ A\} \blacktriangleleft$$

$$H^{CD} := \quad \{t_1^0 \mapsto 1 \ \mathit{at}\ D\} \ t^2 \dashv \text{SEND} \ \{t_0^2 \mapsto 1 \ \mathit{at}\ C\} \ t^4 \dashv \text{SEND} \ \{t_0^4 \mapsto 1 \ \mathit{at}\ D\} \blacktriangleleft$$

C AUTxO examples [ValueSepUTxO.Example]

As was the case for UTxO, we consider validator scripts 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 on account-based ledgers (Appendix A):

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

Most importantly, we no longer need to provide disjointness proofs as in the UTxO case.

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

$$\begin{array}{ll}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \\
\quad t^1 \ldots t^4 & \dashv \text{PAR}(H^{AB}, H^{CD}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & \blacktriangleleft
\end{array}$$

where

$H^{AB} :=$

$$\begin{array}{ll}
\{A \mapsto 1 * B \mapsto 0\} & \\
\qquad t^1 & \dashv \textsc{send} \\
\{A \mapsto 0 * B \mapsto 1\} & \\
\qquad t^3 & \dashv \textsc{send} \\
\{A \mapsto 1 * B \mapsto 0\} & \blacktriangleleft
\end{array}$$

$H^{CD} :=$

$$\begin{array}{ll}
\{C \mapsto 0 * D \mapsto 1\} & \\
\qquad t^2 & \dashv \textsc{send} \\
\{C \mapsto 1 * D \mapsto 0\} & \\
\qquad t^4 & \dashv \textsc{send} \\
\{C \mapsto 0 * D \mapsto 1\} & \blacktriangleleft
\end{array}$$