

1 Program logics for ledgers


2 **Orestis Melkonian** ✉ 

3 University of Edinburgh, Scotland

4 Input Output, Global

5 **Wouter Swierstra** ✉ 

6 Utrecht University, The Netherlands

7 **James Chapman** ✉ 

8 Input Output, Global

9 — Abstract —

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

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

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

20 **2012 ACM Subject Classification** Theory of computation → Program reasoning; Theory of com-
21 putation → Separation logic

22 **Keywords and phrases** blockchain, distributed ledgers, UTxO, Hoare logic, separation logic, pro-
23 gram semantics, formal verification

24 **Digital Object Identifier** 10.4230/LIPIcs...

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

27 **1** Introduction

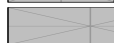
28 Ledger-based cryptocurrencies manage large amounts of money and record monetary trans-
29 fers in copious detail. The market capitalisation of the top ten cryptocurrencies is currently
30 valued at over 750B USD. The underlying blockchain that records transactions, gigabytes in
31 size, is an ever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

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

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



© Author: Please provide a copyright holder;
licensed under Creative Commons License CC-BY 4.0



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

XX:2 Program logics for ledgers

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

47 ■ We show how these same semantics—denotational, operational, and axiomatic—can be
48 given for blockchain ledgers (Section 4), in particular ones based on the *unspent trans-*
49 *action outputs* (or UTxO) model, such as Bitcoin [20] and Cardano [6]. Separation logic,
50 however, poses more of a challenge as the hash-based nature of the UTxO model adds
51 new side conditions to the frame rule that were not necessary for account-based ledgers.

52 ■ To address this problem, we propose a novel variant of UTxO, dubbed *Abstract UTxO*. In
53 contrast to regular UTxO, our Abstract UTxO model supports compositional reasoning
54 using separation logic without further side conditions (Section 5). The resulting logic
55 enables us to reason locally and safely about a limited number of transactions, sprinkled
56 arbitrarily throughout a larger ledger.

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

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

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

63 2 Ledgers & Semantics

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

$$70 \quad S := \mathcal{P} \mapsto \mathbb{Z}$$

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

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

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

79 Alice pays Bob 5;
80 Alice pays Carroll 10;
81 Dana pays Alice 2;
82 ...

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

$$85 \quad T := \mathcal{P} \xrightarrow{n} \mathcal{P}$$

$$86 \quad L := \epsilon \mid T; L$$

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

91 2.1 Denotational semantics

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

$$\begin{array}{l}
 \llbracket _ \rrbracket : L \rightarrow S \rightarrow S \\
 \llbracket \epsilon \rrbracket = \text{id} \\
 \llbracket t; l \rrbracket = \llbracket l \rrbracket \circ d(t)
 \end{array}
 \qquad
 d : T \rightarrow S \rightarrow S
 \qquad
 d(p_1 \xrightarrow{n} p_2)(\sigma) = \lambda p. \begin{cases} \sigma(p) - n & \text{if } p = p_1 \neq p_2 \\ \sigma(p) + n & \text{if } p = p_2 \neq p_1 \\ \sigma(p) & \text{otherwise} \end{cases}$$

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

99 ► **Theorem 1.** *For any ledgers l_1 and l_2 , we have $\llbracket l_1 \uparrow l_2 \rrbracket = \llbracket l_2 \rrbracket \circ \llbracket l_1 \rrbracket$.*

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

104 2.2 Operational semantics

105 Alternatively, we can describe an operational semantics for L . To do so, we define a relation:
 106 $\langle l, \sigma \rangle \rightarrow \tau$, denoting that running the ledger l in the state σ terminates in the state τ .

107 The definition of this relation is entirely straightforward:

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

109 It is straightforward to establish that these two semantics coincide:

110 ► **Theorem 2.** *For any ledger l and state σ , we have that $\llbracket l \rrbracket(\sigma) = \tau$ iff $\langle l, \sigma \rangle \rightarrow \tau$.*

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

114 ► **Corollary 3.** *For any ledgers l, l' and states σ, σ', τ we have that $\langle l, \sigma \rangle \rightarrow \sigma'$ and
 115 $\langle l', \sigma' \rangle \rightarrow \tau$ iff $\langle l \uparrow l', \sigma \rangle \rightarrow \tau$.*

116 2.3 Axiomatic semantics

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

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

XX:4 Program logics for ledgers

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

$$121 \quad \frac{P' \Rightarrow P \quad \{P\} l \{Q\} \quad Q \Rightarrow Q'}{\{P'\} l \{Q'\}} \text{CONSQ}$$

122 Once again, we can relate our axiomatic semantics to its operational and denotational coun-
123 terparts:

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

125 ► **Theorem 5.** $\{P\} l \{Q\}$ holds iff $P(\sigma)$ and $\llbracket l \rrbracket(\sigma) = \tau$ implies $Q(\tau)$ for all σ and τ .

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

$$128 \quad \frac{\{P\} l_1 \{Q\} \quad \{Q\} l_2 \{R\}}{\{P\} l_1 ++ l_2 \{R\}} \text{APP}$$

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

134 Example specification

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

$$\begin{aligned} 137 \quad & \{\lambda\sigma. \sigma(A) = 2\} \\ 138 \quad & A \xrightarrow{1} B \\ 139 \quad & \{\lambda\sigma. \sigma(A) = 1\} \\ 140 \quad & A \xrightarrow{1} C \\ 141 \quad & \{\lambda\sigma. \sigma(A) = 0\} \end{aligned}$$

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

147 However, to prove such statements amounts to providing evidence for each Hoare triple
148 at each step, which involves predicates over the whole state although each transaction can
149 only refer to two distinct participants. In the case of a more complicated state space than
150 just a single participant, this approach is *non-compositional*, since you would need to talk
151 about the whole state you care about in one go. This is precisely the reason we now turn
152 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 $l_1 \uplus l_2$ 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 $S \rightarrow \text{Maybe } S$, 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:

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

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

$$\begin{aligned} \llbracket _ \rrbracket : L \rightarrow S \rightarrow \text{Maybe } S & & d' : T \rightarrow S \rightarrow \text{Maybe } S \\ \llbracket \epsilon \rrbracket = \text{just} & & d'(p_1 \xrightarrow{n} p_2)(\sigma) = \begin{cases} \text{just } d(p_1 \xrightarrow{n} p_2)(\sigma) & \text{if } \sigma(p_1) \geq n \\ \text{nothing} & \text{otherwise} \end{cases} \\ \llbracket t; l \rrbracket = d'(t) \ggg \llbracket l \rrbracket & & \end{aligned}$$

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:

XX:6 Program logics for ledgers

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

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:

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

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 weakened precondition $P \circ d(t)$ implies a transaction's validity, we ensure that adding a new transaction in a ledger always succeeds:

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

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 $\llbracket l \rrbracket(\sigma) = \text{just } \tau$ 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

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

$$226 \quad (\sigma_1 \oplus \sigma_2)(p) := \sigma_1(p) + \sigma_2(p)$$

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

$$229 \quad (P * Q)(\sigma) := \exists \sigma_1. \exists \sigma_2. P(\sigma_1) \wedge Q(\sigma_2) \wedge \sigma = \sigma_1 \oplus \sigma_2$$

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

$$232 \quad \frac{\{P\} l \{Q\}}{\{P * R\} l \{Q * R\}} \text{FRAME}$$

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

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

$$241 \quad \frac{}{\{p_1 \mapsto n\} p_1 \xrightarrow{n} p_2 \{p_2 \mapsto n\}} \text{SEND}$$

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

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

$$248 \quad \frac{}{\{emp\} \epsilon \{emp\}} \text{EMPTY} \qquad \frac{\{P\} l_1 \{Q\} \quad \{Q\} l_2 \{R\}}{\{P\} l_1 \# l_2 \{R\}} \text{APP}$$

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

251 Concurrent separation logic

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

$$255 \quad \frac{\{P_1\} l_1 \{Q_1\} \quad \{P_2\} l_2 \{Q_2\}}{\{P_1 * P_2\} l_1 \parallel l_2 \{Q_1 * Q_2\}} \text{PAR}$$

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

XX:8 Program logics for ledgers

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

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

4 UTxO

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

$$275 \quad \textit{Output} := \{\textit{validator} : \textit{DATA} \rightarrow \mathbb{B}, \textit{value} : \mathbb{N}\}$$

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

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

$$281 \quad \textit{Ref} := \{\textit{tx} : \textit{HASH}, \textit{index} : \mathbb{N}\}$$

$$282 \quad \textit{Input} := \{\textit{ref} : \textit{Ref}, \textit{redeemer} : \textit{DATA}\}$$

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

$$285 \quad \textit{T} := \{\textit{inputs} : [\textit{Input}], \textit{outputs} : [\textit{Output}]\}$$

$$286 \quad \textit{L} := \epsilon \mid \textit{T}; \textit{L}$$

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

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

$$297 \quad \textit{S} := \textit{Ref} \mapsto \textit{Output}$$

298 We again treat finite maps as functions from keys to values; we write $k \in \sigma$ when map σ
299 contains a mapping for reference k , $\sigma \setminus ks$ to remove a set of keys ks in a given map σ , and
300 $\sigma \uplus \sigma'$ 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 :**

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

■ **there is no double spending:**

$$\forall(i, j \in t.inputs). i \neq j \rightarrow i.ref \neq j.ref$$

■ **value is preserved:**

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

■ **all inputs validate:**

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

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:

$$d : T \rightarrow S \rightarrow S$$

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

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

$$\llbracket _ \rrbracket : L \rightarrow S \rightarrow \text{Maybe } S$$

$$d' : T \rightarrow S \rightarrow \text{Maybe } S$$

$$\llbracket \epsilon \rrbracket = \text{just}$$

$$\llbracket t; l \rrbracket = d'(t) \gg \llbracket l \rrbracket$$

$$d'(t)(\sigma) = \begin{cases} \text{just } d(t)(\sigma) & \text{if } t \text{ valid in } \sigma \\ \text{nothing} & \text{otherwise} \end{cases}$$

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,

XX:10 Program logics for ledgers

336 funds are locked by a validator script and must be consumed as a whole: we cannot readily
337 split and combine funds in the same way as we saw previously. Therefore, predicates such as
338 $t_3^\# \mapsto v * t_3^\# \mapsto v'$ no longer make sense, since the third output of transaction t can only be
339 spent once. Consequently, our separating conjunction has to be restricted only to *disjoint*
340 fragments of the global state, as is typical in separation logics reasoning about mutable
341 memory:

$$342 \quad (P * Q)(\sigma) := \exists \sigma_1. \exists \sigma_2. P(\sigma_1) \wedge Q(\sigma_2) \wedge \sigma = \sigma_1 \uplus \sigma_2$$

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

$$346 \quad \frac{\{P\} l \{Q\} \quad l \# R}{\{P * R\} l \{Q * R\}} \text{FRAME}$$

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

$$350 \quad l \# R := \forall s. R(s) \leftrightarrow R(s \setminus \{i.\text{ref} \mid i \in l.\text{inputs}\})$$

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

$$352 \quad \frac{\{P_1\} l_1 \{Q_1\} \quad \{P_2\} l_2 \{Q_2\} \quad l_1 \# P_2 \quad l_2 \# P_1}{\{P_1 * P_2\} l_1 || l_2 \{Q_1 * Q_2\}} \text{PAR}$$

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

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

363 **5 Abstract UTxO**

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

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

$$370 \quad \text{Ref} := \text{Output}$$

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

$$374 \quad S := \text{Bag}\langle \text{Output} \rangle$$

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

$$380 \quad (\sigma_1 \oplus \sigma_2)(p) := \sigma_1(p) + \sigma_2(p)$$

$$381 \quad \sigma \subseteq \tau := \forall x. \sigma(x) \leq \tau(x)$$

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

385 5.1 Denotational semantics

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

395 ■ **there are sufficient funds in σ :**

$$396 \quad t.\text{inputs} \subseteq \sigma$$

397 ■ **value is preserved:**

$$398 \quad \sum_{i \in t.\text{inputs}} i.\text{ref.value} = \sum_{o \in t.\text{outputs}} o.\text{value}$$

399 ■ **all inputs validate:**

$$400 \quad \forall (i \in t.\text{inputs}). i.\text{ref.validator}(i.\text{redeemer}) = \text{true}$$

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

404 The denotational semantics of a single transaction removes previously unspent transac-
 405 tion outputs, replacing them with the outputs of the new transaction:

$$406 \quad d : T \rightarrow S \rightarrow S$$

$$407 \quad d(t)(\sigma) = \sigma \setminus \{i.\text{ref} \mid i \in t.\text{inputs}\} \oplus t.\text{outputs}$$

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

$$411 \quad \llbracket _ \rrbracket : L \rightarrow S \rightarrow \text{Maybe } S \quad d' : T \rightarrow S \rightarrow \text{Maybe } S$$

$$\llbracket \epsilon \rrbracket = \text{just}$$

$$\llbracket t; l \rrbracket = d'(t) \gg \llbracket l \rrbracket \quad d'(t)(\sigma) = \begin{cases} \text{just } d(t)(\sigma) & \text{if } t \text{ valid in } \sigma \\ \text{nothing} & \text{otherwise} \end{cases}$$

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

414 **5.2 Separation Logic**

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

418
$$(P * Q)(\sigma) := \exists \sigma_1. \exists \sigma_2. P(\sigma_1) \wedge Q(\sigma_2) \wedge \sigma = \sigma_1 \oplus \sigma_2$$

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

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

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

431
$$\frac{\{P\} l \{Q\}}{\{P * R\} l \{Q * R\}} \text{FRAME} \qquad \frac{\{P_1\} l_1 \{Q_1\} \quad \{P_2\} l_2 \{Q_2\}}{\{P_1 * P_2\} l_1 || l_2 \{Q_1 * Q_2\}} \text{PAR}$$

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

436 **Example use case**

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

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

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

451 **5.3 Sound abstraction**

452 The relation between AUTxO and UTxO is not yet satisfying, as we need some kind of *full*
 453 *abstraction* [17] result that lets us conduct compositional proofs at the *abstract* (\mathbb{A}) level
 454 which then translate to properties about an actual *concrete* (\mathbb{C}) ledger. One can informally

455 see that all properties that do not observe the implementation details of the concrete model
 456 (i.e. the order of transaction outputs and their specific hashes), should be derivable from
 457 their abstract counterparts.

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

$$460 \quad \text{abs}^S : \mathbb{C}.S \rightarrow \mathbb{A}.S$$

$$461 \quad \text{abs}^S(\sigma) = \{\sigma(k) \mid k \in \sigma\}$$

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

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

$$470 \quad \frac{t \text{ valid in } \sigma \quad \mathbb{C} \llbracket t \rrbracket (\sigma) = \text{just } \tau}{\mathbb{A} \llbracket \text{abs}^T(t) \rrbracket (\text{abs}^S(\sigma)) = \text{just } \text{abs}^S(\tau)}$$

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

$$473 \quad \frac{l \text{ valid in } \sigma \quad \mathbb{C} \llbracket l \rrbracket (\sigma) = \text{just } \tau}{\mathbb{A} \llbracket \text{abs}^L(l) \rrbracket (\text{abs}^S(\sigma)) = \text{just } \text{abs}^S(\tau)}$$

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

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

$$479 \quad \frac{\mathbb{A}\{P\} \text{abs}^L(l) \{Q\} \quad l \text{ valid in } \sigma}{\mathbb{C}\{P \circ \text{abs}^S\} l \{Q \circ \text{abs}^S\}} \text{SOUNDNESS}$$

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

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

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

496 **6 Discussion**497 **6.1 Related Work**498 **Blockchain Theory**

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

508 On the Bitcoin side, there is a mechanised program logic for reasoning about Bitcoin’s
 509 script language [1] based on *predicate transformer* semantics [11]; the striking similarity with
 510 our work lies in the use of weakest preconditions to model access control, which is essentially
 511 what we use to define the STEP rule for our Hoare logic, i.e. in the calculated pre-condition
 512 $\uparrow P \circ d'(t)$.

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

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

523 **Concurrency Theory**

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

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

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

535 **6.2 Future Work**536 **Decompositionality**

537 One aspect that fails to translate to the UTxO setting is the treatment of separated conjunc-
 538 tions as arithmetic formulas, where equivalences such as $A \mapsto 2 \approx A \mapsto 1 * A \mapsto 1$ hold by

539 definition. We can refer to this property as *decompositionality*, since it lets us automatically
540 decompose a large resource into its constituent parts.

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

547 Connection with existing separation logics

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

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

559 6.3 Conclusion

560 We have presented a compositional approach to reasoning about UTxO ledgers, made pos-
561 sible by exploiting the analogy between programs mutating memory and transactions trans-
562 ferring funds between accounts. The key methodological insight is that the ledger can be
563 viewed as a (restricted) programming language, thus opening up the possibility of develop-
564 ing program logics to reason about (sequences of) transactions. We have demonstrated how
565 ideas from separation logic in particular provide the modularity principle to reason about
566 ledger fragments independently of one another.

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

572 References

- 573 1 Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer. Verification of Bit-
574 coin script in Agda using weakest preconditions for access control. In Henning Basold, Jesper
575 Cockx, and Silvia Ghilezan, editors, *27th International Conference on Types for Proofs and
576 Programs, TYPES 2021, June 14-18, 2021, Leiden, The Netherlands (Virtual Conference)*,
577 volume 239 of *LIPICs*, pages 1:1–1:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik,
578 2021. doi:10.4230/LIPICs.TYPES.2021.1.
- 579 2 Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of
580 Bitcoin transactions. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography
581 and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February
582 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer
583 Science*, pages 541–560. Springer, 2018. doi:10.1007/978-3-662-58387-6_29.

- 584 **3** Massimo Bartoletti and Roberto Zunino. Formal models of Bitcoin contracts: A survey.
585 *Frontiers Blockchain*, 2:8, 2019. doi:10.3389/fbloc.2019.00008.
- 586 **4** John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor,
587 *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13,*
588 *2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer,
589 2003. doi:10.1007/3-540-44898-5_4.
- 590 **5** Cristiano Calcagno, Peter W. O’Hearn, and Hongseok Yang. Local action and Abstract
591 Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-*
592 *12 July 2007, Wroclaw, Poland, Proceedings*, pages 366–378. IEEE Computer Society, 2007.
593 doi:10.1109/LICS.2007.30.
- 594 **6** Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Mi-
595 chael Peyton Jones, and Philip Wadler. The Extended UTXO model. In Matthew Bernhard,
596 Andrea Bracciali, L. Jean Camp, Shin’ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and
597 Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2020 International*
598 *Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February*
599 *14, 2020, Revised Selected Papers*, volume 12063 of *Lecture Notes in Computer Science*, pages
600 525–539. Springer, 2020. doi:10.1007/978-3-030-54455-3_37.
- 601 **7** Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann
602 Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native custom tokens
603 in the Extended UTXO model. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging*
604 *Applications of Formal Methods, Verification and Validation: Applications - 9th International*
605 *Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece,*
606 *October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer*
607 *Science*, pages 89–111. Springer, 2020. doi:10.1007/978-3-030-61467-6_7.
- 608 **8** Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann
609 Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnent-
610 ferner. UTXO_{ma}: UTXO with multi-asset support. In Tiziana Margaria and Bernhard Steffen,
611 editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*
612 *- 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020,*
613 *Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes*
614 *in Computer Science*, pages 112–130. Springer, 2020. doi:10.1007/978-3-030-61467-6_8.
- 615 **9** Bob Coecke, Tobias Fritz, and Robert W. Spekkens. A mathematical theory of resources. *Inf.*
616 *Comput.*, 250:59–86, 2016. doi:10.1016/j.ic.2016.02.008.
- 617 **10** Thibault Dardinier, Peter Müller, and Alexander J. Summers. Fractional resources in
618 unbounded separation logic. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1066–1092, 2022.
619 doi:10.1145/3563326.
- 620 **11** Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs.
621 *Communications of the ACM*, 18(8):453–457, 1975.
- 622 **12** Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable
623 binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002. doi:10.1007/s001650200016.
- 624 **13** Murdoch James Gabbay. Algebras of UTxO blockchains. *Math. Struct. Comput. Sci.*,
625 31(9):1034–1089, 2021. doi:10.1017/S0960129521000438.
- 626 **14** Maurice Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*,
627 62(2):78–85, 2019. doi:10.1145/3209623.
- 628 **15** Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek
629 Dreyer. Iris from the ground up: A modular foundation for higher-order Concurrent Separation
630 Logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 631 **16** Orestis Melkonian, Wouter Swierstra, and Manuel M. T. Chakravarty. Formal investigation of
632 the Extended UTxO model. In *4th ACM SIGPLAN International Workshop on Type-Driven*
633 *Development (TyDe)*, 2019. URL: [https://omelkonian.github.io/data/publications/](https://omelkonian.github.io/data/publications/formal-utxo.pdf)
634 [formal-utxo.pdf](https://omelkonian.github.io/data/publications/formal-utxo.pdf).

- 635 17 Robin Milner. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci.*, 4(1):1–22,
636 1977. doi:10.1016/0304-3975(77)90053-6.
- 637 18 Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and
638 Hans Moog. Reality-based UTXO ledger. *CoRR*, abs/2205.01345, 2022. arXiv:2205.01345,
639 doi:10.48550/arXiv.2205.01345.
- 640 19 Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and
641 Hans Moog. Tangle 2.0 leaderless nakamoto consensus on the heaviest DAG. *IEEE Access*,
642 10:105807–105842, 2022. doi:10.1109/ACCESS.2022.3211422.
- 643 20 S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [https://bitcoin.org/en/
644 bitcoin-paper](https://bitcoin.org/en/bitcoin-paper), October 2008.
- 645 21 Chad Nester. A foundation for ledger structures. In Emmanuelle Anceaume, Christophe
646 Bisière, Matthieu Bouvard, Quentin Bramas, and Catherine Casamatta, editors, *2nd Interna-
647 tional Conference on Blockchain Economics, Security and Protocols, Tokenomics 2020, Octo-
648 ber 26-27, 2020, Toulouse, France*, volume 82 of *OASICs*, pages 7:1–7:13. Schloss Dagstuhl -
649 Leibniz-Zentrum für Informatik, 2020. doi:10.4230/OASICs.Tokenomics.2020.7.
- 650 22 Chad Nester. The structure of concurrent process histories. In Ferruccio Damiani and Ornella
651 Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Con-
652 ference, COORDINATION 2021, Held as Part of the 16th International Federated Conference
653 on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021,
654 Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 209–224. Springer,
655 2021. doi:10.1007/978-3-030-78142-2_13.
- 656 23 George Pirlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with own-
657 ership and commutativity analysis. In Stephen N. Freund and Eran Yahav, editors, *PLDI
658 '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and
659 Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1327–1341. ACM, 2021.
660 doi:10.1145/3453483.3454112.
- 661 24 John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *17th
662 IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen,
663 Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.
664 1029817.
- 665 25 Peter Selinger. A survey of graphical languages for monoidal categories. *New structures for
666 physics*, pages 289–355, 2011.
- 667 26 Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In Mi-
668 chael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa
669 Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, edi-
670 tors, *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC,
671 BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Pa-
672 pers*, volume 10323 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2017.
673 doi:10.1007/978-3-319-70278-0_30.
- 674 27 Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. URL: [http://arxiv.
675 org/abs/1804.06398](http://arxiv.org/abs/1804.06398), arXiv:1804.06398.
- 676 28 Wouter Swierstra and Andres Löb. The semantics of version control. In Andrew P. Black,
677 Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014,
678 Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and
679 Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October
680 20-24, 2014*, pages 43–54. ACM, 2014. doi:10.1145/2661136.2661137.
- 681 29 Joachim Zahnentferner. An abstract model of UTxO-based cryptocurrencies with scripts.
682 *IACR Cryptol. ePrint Arch.*, page 469, 2018. URL: <https://eprint.iacr.org/2018/469>.

XX:18 Program logics for ledgers

683 **A** Examples

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

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

$$688 \frac{}{\{A \mapsto n\} A \xrightarrow{n} B; B \xrightarrow{n} A \{A \mapsto n\}} \text{CANCEL} \quad \frac{}{\{P * Q\} \approx \{Q * P\}} \text{SWAP}$$

690
691

692 **Simple example using FRAME**

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

$$\begin{aligned} 695 & \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\ 696 & \quad A \xrightarrow{1} B \quad \quad \quad \dashv \text{FRAME}(C \mapsto 0 * D \mapsto 1, \text{SEND}) \\ 697 & \{A \mapsto 0 * B \mapsto 1 * C \mapsto 0 * D \mapsto 1\} \\ 698 & \quad \approx \\ 699 & \{C \mapsto 0 * D \mapsto 1 * A \mapsto 0 * B \mapsto 1\} \\ 700 & \quad D \xrightarrow{1} C \quad \quad \quad \dashv \text{FRAME}(A \mapsto 0 * B \mapsto 1, \text{SEND} \circ \text{SWAP}) \\ 701 & \{C \mapsto 1 * D \mapsto 0 * A \mapsto 0 * B \mapsto 1\} \\ 702 & \quad \approx \\ 703 & \{A \mapsto 0 * B \mapsto 1 * C \mapsto 1 * D \mapsto 0\} \\ 704 & \quad B \xrightarrow{1} A \quad \quad \quad \dashv \text{FRAME}(C \mapsto 1 * D \mapsto 0, \text{SEND} \circ \text{SWAP}) \\ 705 & \{A \mapsto 1 * B \mapsto 0 * C \mapsto 1 * D \mapsto 0\} \\ 706 & \quad \approx \\ 707 & \{C \mapsto 1 * D \mapsto 0 * A \mapsto 1 * B \mapsto 0\} \\ 708 & \quad C \xrightarrow{1} D \quad \quad \quad \dashv \text{FRAME}(A \mapsto 1 * B \mapsto 0, \text{SEND}) \\ 709 & \{C \mapsto 0 * D \mapsto 1 * A \mapsto 1 * B \mapsto 0\} \\ 710 & \quad \approx \\ 711 & \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft \\ 712 & \end{aligned}$$

713 **Simple example using PAR**

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

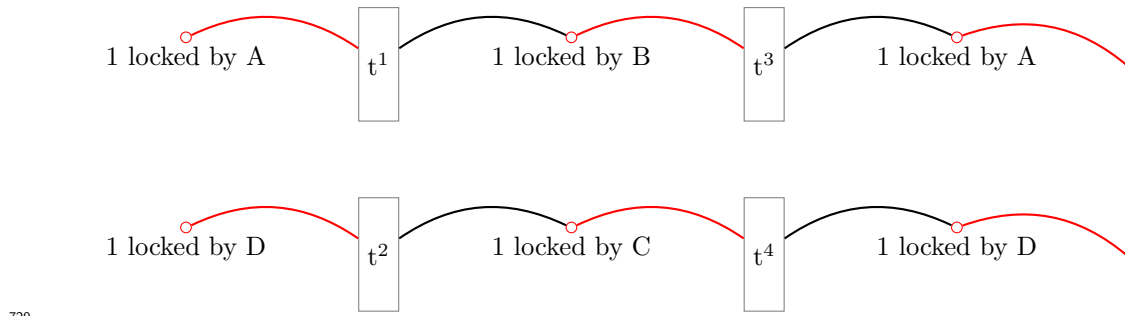
$$\begin{aligned} 717 & \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\ 718 & \quad (A \xrightarrow{1} B; B \xrightarrow{1} A) | (D \xrightarrow{1} C; C \xrightarrow{1} D) \\ 719 & \quad \ni (A \xrightarrow{1} B; D \xrightarrow{1} C; B \xrightarrow{1} A; C \xrightarrow{1} D) \quad \dashv \text{PAR}(H^{AB}, H^{CD}) \\ 720 & \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft \\ 721 & \end{aligned}$$

722 where

$$\begin{array}{c}
 722 \quad H^{AB} := \\
 723 \quad \{A \mapsto 1 * B \mapsto 0\} \\
 \quad A \xrightarrow{1} B; B \xrightarrow{1} A \quad \dashv \text{CANCEL} \\
 \{A \mapsto 1 * B \mapsto 0\} \quad \blacktriangleleft
 \end{array}
 \quad \Bigg| \quad
 \begin{array}{c}
 H^{CD} := \\
 \{C \mapsto 0 * D \mapsto 1\} \\
 \quad D \xrightarrow{1} C; C \xrightarrow{1} D \quad \dashv \text{CANCEL} \\
 \{C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft
 \end{array}$$

724 **UTxO example using FRAME**

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



$$\begin{array}{c}
 730 \quad \{t_0^0 \mapsto 1 \text{ at } A * t_1^0 \mapsto 1 \text{ at } D\} \\
 731 \quad \quad \quad t^1 \quad \quad \quad \dashv \text{FRAME}(t_1^0 \mapsto 1 \text{ at } D, \dots, \text{SEND}) \\
 732 \quad \{t_0^1 \mapsto 1 \text{ at } B * t_1^0 \mapsto 1 \text{ at } D\} \\
 733 \quad \quad \quad \approx \\
 734 \quad \{t_1^0 \mapsto 1 \text{ at } D * t_1^1 \mapsto 1 \text{ at } B\} \\
 735 \quad \quad \quad t^2 \quad \quad \quad \dashv \text{FRAME}(t_1^0 \mapsto 1 \text{ at } B, \dots, \text{SEND}) \\
 736 \quad \{t_0^2 \mapsto 1 \text{ at } C * t_1^1 \mapsto 1 \text{ at } B\} \\
 737 \quad \quad \quad \approx \\
 738 \quad \{t_0^1 \mapsto 1 \text{ at } B * t_2^0 \mapsto 1 \text{ at } C\} \\
 739 \quad \quad \quad t^3 \quad \quad \quad \dashv \text{FRAME}(t_2^0 \mapsto 1 \text{ at } C, \dots, \text{SEND}) \\
 740 \quad \{t_0^3 \mapsto 1 \text{ at } A * t_2^0 \mapsto 1 \text{ at } C\} \\
 741 \quad \quad \quad \approx \\
 742 \quad \{t_0^2 \mapsto 1 \text{ at } C * t_3^0 \mapsto 1 \text{ at } A\} \\
 743 \quad \quad \quad t^4 \quad \quad \quad \dashv \text{FRAME}(t_3^0 \mapsto 1 \text{ at } A, \dots, \text{SEND}) \\
 744 \quad \{t_0^4 \mapsto 1 \text{ at } D * t_3^0 \mapsto 1 \text{ at } A\} \\
 745 \quad \quad \quad \approx \\
 746 \quad \{t_0^3 \mapsto 1 \text{ at } A * t_4^0 \mapsto 1 \text{ at } D\} \quad \blacktriangleleft \\
 747
 \end{array}$$

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

XX:20 Program logics for ledgers

750 UTxO example using PAR

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

$$\begin{array}{l}
 754 \quad \{t_0^0 \mapsto 1 \text{ at } A * t_1^0 \mapsto 1 \text{ at } D\} \\
 755 \quad \quad t^1 \dots t^4 \\
 756 \quad \{t_0^3 \mapsto 1 \text{ at } A * t_0^4 \mapsto 1 \text{ at } D\} \\
 757
 \end{array}
 \quad \dashv \text{PAR}(\dots, H^{AB}, H^{CD}) \quad \blacktriangleleft$$

758 where

$$\begin{array}{l}
 H^{AB} := \\
 \quad \{t_0^0 \mapsto 1 \text{ at } A\} \\
 \quad \quad t^1 \quad \dashv \text{SEND} \\
 \quad \{t_0^1 \mapsto 1 \text{ at } B\} \\
 \quad \quad t^3 \quad \dashv \text{SEND} \\
 \quad \{t_0^3 \mapsto 1 \text{ at } A\} \quad \blacktriangleleft \\
 \\
 H^{CD} := \\
 \quad \{t_1^0 \mapsto 1 \text{ at } D\} \\
 \quad \quad t^1 \quad \dashv \text{SEND} \\
 \quad \{t_0^2 \mapsto 1 \text{ at } C\} \\
 \quad \quad t^4 \quad \dashv \text{SEND} \\
 \quad \{t_0^4 \mapsto 1 \text{ at } D\} \quad \blacktriangleleft
 \end{array}$$

760 AUTxO example using FRAME

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

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

$$\begin{array}{l}
 766 \quad \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
 767 \quad \quad t^1 \\
 768 \quad \{A \mapsto 0 * B \mapsto 1 * C \mapsto 0 * D \mapsto 1\} \\
 769 \quad \quad \approx \\
 770 \quad \{C \mapsto 0 * D \mapsto 1 * A \mapsto 0 * B \mapsto 1\} \\
 771 \quad \quad t^2 \\
 772 \quad \{C \mapsto 1 * D \mapsto 0 * A \mapsto 0 * B \mapsto 1\} \\
 773 \quad \quad \approx \\
 774 \quad \{A \mapsto 0 * B \mapsto 1 * C \mapsto 1 * D \mapsto 0\} \\
 775 \quad \quad t^3 \\
 776 \quad \{A \mapsto 1 * B \mapsto 0 * C \mapsto 1 * D \mapsto 0\} \\
 777 \quad \quad \approx \\
 778 \quad \{C \mapsto 1 * D \mapsto 0 * A \mapsto 1 * B \mapsto 0\} \\
 779 \quad \quad t^4 \\
 780 \quad \{C \mapsto 0 * D \mapsto 1 * A \mapsto 1 * B \mapsto 0\} \\
 781 \quad \quad \approx \\
 782 \quad \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
 783
 \end{array}
 \quad \dashv \text{FRAME}(C \mapsto 0 * D \mapsto 1, \text{SEND}) \\
 \dashv \text{FRAME}(A \mapsto 0 * B \mapsto 1, \text{SEND}) \\
 \dashv \text{FRAME}(C \mapsto 1 * D \mapsto 0, \text{SEND}) \\
 \dashv \text{FRAME}(A \mapsto 1 * B \mapsto 0, \text{SEND}) \quad \blacktriangleleft$$

784 **AUTxO example using PAR**

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

$$\begin{array}{l}
786 \quad \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
787 \quad \quad \quad t^1 \dots t^4 \quad \quad \quad \neg \text{PAR}(H^{AB}, H^{CD}) \\
788 \quad \{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft
\end{array}$$

790 where

$$\begin{array}{l}
H^{AB} := \\
\quad \{A \mapsto 1 * B \mapsto 0\} \\
\quad \quad \quad t^1 \quad \quad \quad \neg \text{SEND} \\
791 \quad \{A \mapsto 0 * B \mapsto 1\} \\
\quad \quad \quad t^3 \quad \quad \quad \neg \text{SEND} \\
\quad \{A \mapsto 1 * B \mapsto 0\} \quad \blacktriangleleft
\end{array}
\quad \Bigg| \quad
\begin{array}{l}
H^{CD} := \\
\quad \{C \mapsto 0 * D \mapsto 1\} \\
\quad \quad \quad t^2 \quad \quad \quad \neg \text{SEND} \\
\quad \{C \mapsto 1 * D \mapsto 0\} \\
\quad \quad \quad t^4 \quad \quad \quad \neg \text{SEND} \\
\quad \{C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft
\end{array}$$