Algebras of UTxO blockchains

Murdoch J. Gabbay

We condense the theory of UTxO blockchains down to a simple and compact set of four type equations (Idealised EUTxO), and to an algebraic characterisation (abstract chunk systems), and exhibit an adjoint pair of functors between them. This gives a novel account of the essential mathematical structures underlying blockchain technology, such as Bitcoin.

Contents

1. INTRODUCTION

Blockchain is a young field — young enough that no consensus has yet developed as to its underlying mathematical structures. There are many blockchain implementations, but what (mathematically) are they implementations *of*?

Two major blockchain architectures exist:

— *UTxO-based* blockchains (like Bitcoin), and
— *accounts-based* blockchains (like Ethereum).

We consider UTxO-style blockchains in this paper, and specifically the *Extended* UTxO-style model [CCM+20], which as the name suggests extends the UTxO structure (how, is described in Remark 3.8.1) of Bitcoin, which is still the canonical blockchain application.

So our question becomes: what, mathematically speaking, is an EUTxO blockchain?

In the literature, Figure 3 of [CCM+20] exhibits Extended UTxO as an inductive datatype, designed with implementation and formal verification in mind. Most blockchains exist only in code, so to have an inductive specification to work from in a published academic paper is a luxury for which we can be grateful.

However, this does not answer our question.

It would be like answering *"What are numbers?"* with the inductive definition $\mathbb{N} = 1 + \mathbb{N}$: this an important structure (and to be fair, it yields an important inductive principle) but this does not tell us that $\mathbb{N}$ is a ring; or about primes and the fundamental theorem of arithmetic; or that $\mathbb{N}$ is embedded in $\mathbb{Q}$ and $\mathbb{R}$; or even about binary representations. In short: Figure 3 of [CCM+20] gives us the raw data structure of one particular blockchain implementation, which is certainly important, but this is not a *mathematics of blockchains*.

As we shall see, there is more to be said here.

1.1. Map of the paper

A map of this paper, and our answer, is as follows:

(1) In Section 3 we present *Idealised EUTxO* (Definition 3.1.1), which is four type equations (Figure 1).
This captures the essence of [CCM+20, Figure 3], but far more succinctly — four lines vs. one full page.[1]
So: EUTxO is a solution to the IEUTxO equations in Figure 1.

(2) The approach to blockchain in this paper is novel — we concentrate not on *blockchains* but on *blockchain segments*, which we call *chunks* (Definition 3.4.1).
Chunks have many properties that blockchains do not have: if you cut a blockchain into pieces you get chunks, not blockchains; and chunks have more structure, e.g. they form a partially-ordered partial monoid (Theorem 3.5.4) which communicate across *channels* (much like the π-calculus [Mil99]) and they display resource separation properties reminiscent of known systems such as separation logic (Remark 3.5.13).
A blockchain is the special case of a chunk with no active input channels (Definition 3.4.10).
So: EUTxO is a system of chunks (a partially-ordered partial monoid with channels).

(3) IEUTxO models form a category (Definition 3.7.1).
So now EUTxO is the category of partially-ordered partial monoid solutions to the IEUTxO models, and arrows between them.

(4) Our answers are still quite concrete, in the sense that objects are solutions to type equations. To go further, we use algebra.
We introduce *abstact chunk systems* (Definition 4.5.1), which are oriented atomic monoids of chunks (Definitions 4.4.1, 4.2.5, and 4.2.1). These too form a category (Definition 4.5.2), with objects and arrows.
Thus, we extract relevant properties of what makes solutions to the IEUTxO equations in Figure 1 *interesting*, as explicit and testable algebraic properties (see the discussion in Subsection 8.3).
Several design choices exist in this space: we discuss some of them in Remarks 6.4.4 and 7.1.5 and Proposition 7.3.6.
So now EUTxO is a bundle of abstract algebraic, testable properties, which exists in a clean design space which could be explored in future work.

(5) Finally, we pull this all together by constructing functors between the categories of IEUTxO models and of abstract chunk systems (Definitions 5.1.1 and 6.2.1), and we exhibit a cycle of categorical embeddings between them (Theorem 7.3.4).
So finally, EUTxO becomes a pair of categories — one of concrete solutions to some type equations, and this embedded in a category of abstract algebras — related by adjoint functors mapping between them; or if the reader prefers, it becomes a loop of embeddings (illustrated in Remark 7.3.5), cycling between the concrete and abstract algebras.

There are many definitions and results in this paper, and this leads to a broader point of it: that the mathematics we observe is *possible*. There is a mathematics of blockchain here which we have not seen commented on before.

This may help make blockchain more accessible and interesting to a mathematical audience, and improve communication — since there is no more effective language for handling complexity than mathematics. Furthermore, as we argue in Subsections 4.4.2 and 8.3, our analysis of blockchain structure in this paper is not just of interest to mathematicians — it may also be of practical interest to programmers — by suggesting ways to structure and transform code, and establishing properties for unit tests, property-based testing, and formal verification of correctness — and to designers of new UTxO-based systems, wishing to attain good design and security by working from a (relatively compact) mathematical reference model.

More exposition and discussion is in the body of the paper and in Section 8, including discussions of future work.[2]

[1] This is not a criticism of the original inductive definition.
[2] Tip: search the pdf for 'future work'.

2. SOME BACKGROUND

2.1. What this paper is (not) about

We are parametric across possible data.

Blockchains are best-known as stores of value but it is widely appreciated in the industry that a blockchain is just a particular kind of distributed database, and that the data stored on it, and consistency conditions imposed on its transactions, can vary with the application.

This paper (in common with many real-life blockchain implementations) is parametric in the type of data stored on it. Specifically, we include as a parameter an uninterpreted type β of 'data' in our 'Idealised IEUTxO', as a user-determined black box; and we admit a choice of admissible transactions and validators (this is the subset inclusions in the last two lines of Figure 1).[3]

We do not consider networks or security.

Real blockchains rightly work hard to be efficient and secure. In the real world we hash data, network latency matters, as do good cryptography, incentives, permissions, user training, passwords, privacy, and more.

Again we are parametric in these concerns. We include an uninterpreted type α of 'keys', but do not force any particular cryptographic content on them.[4]

We do have validators, but we elide their computational content.

The UTxO model of adding a block to a chain is that the chain has 'unspent outputs' — meaning output ports that have not yet engaged in interaction — and at each unspent output o is located some *data* d and a *validator* v.

A *validator* is a machine that takes data-key pairs as input, and decides whether they are 'good' or 'bad'. Good data-key pairs let the user interact with the blockchain; bad ones get rejected. In more detail, to append a block to the blockchain, we

— find a validator v on an unspent output o with data d, and
— present it with a suitable key k such v considers (d, k) to be 'good';

the output is now considered 'spent' and our block is attached to the chain.[5]

This is much like putting a key into a lock to open a door, except that appending a block is irreversible: an output, once spent, cannot be un-spent.[6] The idea is that our newly-attached block may introduce fresh outputs with validators which we designed and to which we have the keys;[7] these are new 'unspent' outputs on the chain.[8]

---

[3] E.g. currencies and smart contracts are eminently compatible with our models — and most likely other as-yet-unimagined extensions too. Algebra is good at accommodating examples; how many boolean algebras, groups, or vector spaces does the average person encounter in a lifetime (whether they know it or not)? So if this maths stimulates the reader to think *"Obviously this model could also accommodate* X*; why did the author not mention this?"*, then everything will be just as it should be.

[4] It's not that we don't care, or think these issues are trivial. But consider: Java disallows pointer arithmetic and presents the user with an abstraction of infinite memory addressed by uninterpreted abstract pointers. This does not imply that Java programmers believe that memory actually *is* infinite, or that RAM is not linearly addressed. It's just more productive — and also actually safer — to handle memory-management as a distinct design issue.

[5] In practice, the code of d and v is usually public and can be read directly off the blockchain. The cleverness of cryptography is to devise systems such that from d and v it is not trivial to deduce a k such that v will accept (d, k) — unless you already know a cryptographic secret.

[6] When we write 'irreversible' what we mean in real life is so-called *probabilistic finality*, that the practical chances of a block append getting reversed decreases rapidly and exponentially as other blocks are added after it.

[7] The case where we deliberately attach a block that consumes an output and introduces no fresh outputs on the new block to replace it, is in some contexts called a *burn*, because its effect is to destroy unspent outputs without replacing them with new ones.

[8] So for instance, if we had an oracle that could solve the cryptographic puzzles currently attached to validators on bitcoin, then we would be able to 'spend' bitcoin by attaching new blocks to the chain. As discussed, in the real world these puzzles are designed to be practically impossible to solve unless you have the key. Fun fact: somewhere in a rubbish dump in Newport, UK is a hard drive with the keys to 7,500 bitcoin.

We do model validators in this paper, but mathematically — meaning that a validator is not a code-script (as it would be in real life). Instead, we identify a validator directly with the set of data-key pairs that it accepts.[9]

REMARK 2.1.1. In summary:

> *Idealised (E)UTxO is a mathematical model of the structural act of UTxO block combination and validation*

and

> *Abstract chunk systems are an algebraic rendering of the same idea.*

This basic idea of block combination and validation might not seem like much.[10] However, this is *the* fundamental operation of the UTxO architecture, and we shall find many interesting and unexpected observations to make about it.

So if there is one question that this paper addresses, it is this:

> *What does it mean, mathematically, when we append a transaction to a UTxO blockchain?*

Our answer occupies the rest of this paper.

In the rest of this Section we will set up some basic mathematical machinery. The reader is welcome to skip or skim it, and refer back to it as required.

2.2. Basic data structures

DEFINITION 2.2.1.

(1) Fix a countably infinite set $\mathbb{A} = \{a, b, c, p, \dots\}$ of **atoms**.
(2) A **permutation** is a bijection on $\mathbb{A}$; write $\pi, \pi' \in Perm$ for permutations.

REMARK 2.2.2. Following the ideas in [Gab20a; GP01] atoms will be the atoms of ZFA of Zermelo-Fraenkel set theory with Atoms[11] — this is a fancy way to say that $\mathbb{A}$ is a type of atomic identifiers.

We will use atoms to locate inputs and outputs on a blockchain. More on this in Subsection 2.3.

NOTATION 2.2.3.

(1) Write $\mathbb{N} = \{0, 1, 2, \dots\}$.
(2) If X is a set, write $\mathit{fin}(\mathsf{X})$ for the finite powerset of X, and $\mathit{fin}_!(\mathsf{X})$ for the **pointed finite powerset**. In symbols:
$$\mathit{fin}_!(\mathsf{X}) = \{(X, x) \in \mathit{fin}(\mathsf{X}) \times \mathsf{X} \mid x \in X\}.$$
Above, (X, x) is a pair, and $\mathit{fin}(\mathsf{X}) \times \mathsf{X}$ is a cartesian product.
(3) If X and Y are sets, we use a convenient shorthand in Figure 1 by writing
$$(\mathit{fin}(\mathsf{X}), \mathsf{Y})_! \quad \text{as shorthand for} \quad (\mathit{fin}(\mathsf{X})_!, \mathsf{Y}).$$

[9] Contrast with [CCM+20, Figure 3], which is truer to how an efficient implementation would actually work: it assumes an "(opaque) type of scripts" and a generic script application function ⟦-⟧ which operates on a general-purpose type Data of data, into which it is assumed that inputs get encoded.

The validators-as-graphs-of-function paradigm of this paper rests on two basic observations: 1. we can identify a program-script with the function that it computes (then throw away the script and keep the function); and 2. a function can be represented as its **graph** $\{(x, y) \mid y = f(x)\}$. (We could further insist that a validator be a *computable* graph, but we don't need to for our results here, so we don't.)

We spell out points 1. and 2. here *precisely because* they may be taken as obvious by some readers, but *saying this* appears to be novel in the peer-reviewed blockchain literature; and I know from conversations that either point may be novel to some readers.

[10] ...neither did 'plus one' when I learned to count in school, and look what comes of *that* in university. Basic ideas can be tricky like that.

[11] Also called *urelemente* or *urelements* in the set-theoretic literature.

That is, we take $(\text{-})_!$ to act on a pair functorially, on the first component. We do this in Figure 1 when we write $\mathsf{Transaction}_!$ in the definition of $\mathsf{Validator}$.

(4) If X is a set then write $[\mathsf{X}]$ for the set of (possibly empty) finite lists of elements from X. We write $\cdot$ for list concatenation, so $l \cdot l'$ is l followed by l'.
More generally, we will write $\cdot$ for any monoid composition; list concatenation is one instance. It will always be clear what is intended.

(5) If X is a set then order $l, l' \in [\mathsf{X}]$ by the **sublist inclusion** relation, where $l \leq l'$ when l can be obtained from l' by deleting (but not rearranging) some of its elements.

(6) If X is a set and $x \in \mathsf{X}$ then we may call the one-element list $[x] \in [\mathsf{X}]$ a **singleton**.

(7) If $V \in \mathsf{X} \to \mathsf{Bool}$ and $x \in \mathsf{X}$ then we may write $V(x)$ or $V\,x$ for $V\,x = \mathsf{True}$.

2.3. The permutation action

Remark 2.3.1. We spend this Subsection introducing permutations and their action on elements. We will need this most visibly in two places:

(1) To state the key Definition 4.3.6.
(2) To prove Lemma 5.2.1, and thus Proposition 5.2.2.

Because we assume atoms and are working in a ZFA universe, everything has a standard permutation action. We describe it in Definition 2.3.3. Programmers can think of the permutation action as a *generic* definition in the ZFA universe (given below in this Remark), which is sufficiently generic that it exists for all the datatypes considered in this paper. By this perspective, Definition 2.3.3 specifies how this generic action interacts with the specific type-formers of interest for this paper.

Definition 2.3.2. For reference we write out the ZFA generic definition, which is by ϵ-induction on the sets universe:

$$\begin{array}{ll} \pi{\cdot}a = \pi(a) & a \in \mathbb{A} \\ \pi{\cdot}X = \{\pi{\cdot}x \mid x \in X\} & X \text{ a set.} \end{array}$$

More information on this sets inductive definition is in [Gab20a; Gab01]. Definition 2.3.3 can be usefully viewed as a collection of concrete instances of Definition 2.3.2 for the datatypes of interest in this paper:

Definition 2.3.3. Permuations π act concretely as follows:

(1) If $\pi \in \mathit{Perm}$ and $a \in \mathbb{A}$ then π acts on a as a function:
$$\pi{\cdot}a = \pi(a).$$

(2) If $\pi \in \mathit{Perm}$ and X is any set then π acts **pointwise** on X as follows:
$$\pi{\cdot}X = \{\pi{\cdot}x \mid x \in X\}.$$
Note as a corollary of this that $x \in X \iff \pi{\cdot}x \in \pi{\cdot}X$.

(3) If $\pi \in \mathit{Perm}$ and $(x_1, \ldots, x_n)$ is a tuple then π acts **pointwise** on $(x_1, \ldots, x_n)$ as follows:
$$\pi{\cdot}(x_1, \ldots, x_n) = (\pi{\cdot}x_1, \ldots, \pi{\cdot}x_n).$$
Note therefore that $\pi{\cdot}((x_1, \ldots, x_n)!!i) = \pi{\cdot}x_i$, where $1 \leq i \leq n$ and !! indicates lookup.

(4) If $\pi \in \mathit{Perm}$ and $i \in \mathbb{N}$ then $\pi{\cdot}i = i$.

(5) If $\pi \in \mathit{Perm}$ and (X, x) is a pointed set (Notation 2.2.3(2)) then π acts **pointwise** on (X, x) as follows:
$$\pi{\cdot}(X, x) = (\pi{\cdot}X, \pi{\cdot}x).$$
(This is indeed just a special case of the previous case, for tuples.)

(6) If $\pi \in \mathit{Perm}$ and f is a function then π has the **conjugation action** on f as follows:
$$(\pi{\cdot}f)(x) = \pi{\cdot}(f(\pi^{-1}{\cdot}x)).$$
Note therefore that $\pi{\cdot}(f(x)) = (\pi{\cdot}f)(\pi{\cdot}x)$, and $\pi{\cdot}f$ maps $\pi{\cdot}x$ to $\pi{\cdot}(f(x))$.

$$\begin{aligned}\mathsf{Input} &= \mathbb{A} \times \alpha\\ \mathsf{Output} &= \mathbb{A} \times \beta \times \mathsf{Validator}\\ \mathsf{Transaction} &\subseteq \big(\mathit{fin}(\mathsf{Input}) \times \mathit{fin}(\mathsf{Output})\big) \setminus \{(\varnothing,\varnothing)\}\\ \mathsf{Validator} &\subseteq \mathit{pow}(\beta \times \mathsf{Transaction}_!)\end{aligned}$$

Fig. 1. Type equations of Idealised EUTxO

(7) In particular, π acts as the above on the inputs, outputs, sets of inputs, sets of outputs, transactions, and validators from Figure 1.
(8) If $\pi \in \mathit{Perm}$ and R is a relation, then π acts **pointwise** such that

$$\pi{\cdot}R = \{(\pi{\cdot}x, \pi{\cdot}y) \mid (x,y) \in R\}$$

so that

$$x\ \pi{\cdot}R\ y \iff \pi^{-1}{\cdot}x\ R\ \pi^{-1}{\cdot}y.$$

We use Definition 2.3.4 in Definition 4.3.6, but it is a useful concept so we include it here:

Definition 2.3.4. If $a \in \mathbb{A}$ then write $\mathit{fix}(a)$ for the set of permutations $\pi \in \mathit{Perm}$ such that $\pi(a) = a$. In symbols:

$$\mathit{fix}(a) = \{\pi \in \mathit{Perm} \mid \pi(a) = a\}.$$

Definition 2.3.5. Call an element **equivariant** when $\pi{\cdot}x = x$ for every $\pi \in \mathit{Perm}$. Concretely:

(1) $\mathbb{A}$ is equivariant, and no individual atom $a \in \mathbb{A}$ is equivariant.
(2) A set X is equivariant when

$$\forall \pi{\in}\mathit{Perm}.(x \in X \iff \pi{\cdot}x \in X).$$

In words:

A set is equivariant precisely when it is closed in the orbits of its elements under the permutation action.

(3) $(x_1, \ldots, x_n)$ is equivariant precisely when x_i is equivariant for every $1 \leq i \leq n$.
(4) $\mathbb{N}$ is equivariant, and every $i \in \mathbb{N}$ is equivariant.
(5) A pointed set (X, x) is equivariant precisely when X and x are equivariant.
(6) A function f is equivariant when $\pi{\cdot}(f(x)) = f(\pi{\cdot}x)$ for every x and every π.
(7) A relation R is equivariant when

$$\forall \pi{\in}\mathit{Perm}.\forall x, y.(x\ R\ y \iff \pi{\cdot}x\ R\ \pi{\cdot}y).$$

3. IDEALISED EUTXO: IEUTXO

3.1. IEUTxO equations and solutions

Definition 3.1.1. Let **idealised EUTxO** be the type equations in Figure 1 ($\mathsf{Transaction}_!$ is from Notation 2.2.3(3)).

Definition 3.1.2. Let a **solution** or **model** of the IEUTxO type equations in Figure 1 be a tuple

$$\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator}, \nu : \mathsf{Validator} \hookrightarrow \mathit{pow}(\beta \times \mathsf{Transaction}_!))$$

where:

(1) α is an uninterpreted[12] equivariant (Definition 2.3.5(2)) set of *keys*.
(2) β is an uninterpreted equivariant set of *data*.
(3) $\mathsf{Validator}$ is an equivariant set of *validators*.

[12] *Uninterpreted* means: "Pick whatever you want. We'll never look inside it so we don't care. Just make sure this set is large enough and has whatever structure that *you* need for *your* application, and the maths will work for that choice."

(4) Transaction is an equivariant subset of $\mathit{fin}(\mathbb{A} \times \alpha) \times \mathit{fin}(\mathbb{A} \times \beta \times \mathsf{Validator})$ and we disallow the empty transaction, having no inputs or outputs.
(5) ν is an equivariant injective function (Definition 2.3.5(6)) from Validator to $\mathit{pow}(\beta \times \mathsf{Transaction}_!)$.

Some notation will be helpful:

NOTATION 3.1.3. If $tx = (I, O) \in \mathsf{Transaction}$ and $i \in I$ then write $tx@i \in \mathsf{Transaction}_!$ for the input-in-context $((I, i), O)$ obtained by pointing I at $i \in I$ (Notation 2.2.3). In symbols:

$$\text{if } tx = (I, O) \in \mathsf{Translation} \text{ and } i \in I \quad \text{then} \quad tx@i = ((I, i), O) \in \mathsf{Transaction}_!$$

EXAMPLE 3.1.4. In Figures 2 and 3 we take a short diagrammatic tour of Definition 3.1.2, before continuing with the technical development. Since we have yet to build our machinery, this discussion is informal and intuitive. We include precise forward pointers to later definitions where appropriate; additional diagrams will be discussed in detail in Example 3.4.11.

$tx \in \mathsf{Transaction}$ in Figure 2 is a pair of a set of three inputs, and a set of two outputs:

$$tx \;=\; \Big(\{(a, x_1), (b, x_2), (c, x_3)\} \;,\; \{(d, y_1, v_1), (e, y_2, v_2)\}\Big).$$

Above:

— a, b, c, d, and e are atoms in $\mathbb{A}$. They serve as unique labels for the inputs and outputs.
Note that every input and output in tx has a distinct label. This is not directly enforced in the raw datatype in Figure 1, but it will follow as a consequence of well-formedness conditions which we introduce later, in Definition 3.4.1.
Since each input is labelled with a unique atom, and similarly for each output, we may treat atoms as *positions*, *locations*, or *channels*. Thus for example the input (a, x_1) is labelled with a and so we can think intuitively that it is located at position a; or waiting to communicate its data x_1 on channel a.
— x_1 and x_2 are elements of α. This is just an uninterpreted datatype, but a good intuition is that these are cryptographic keys.
— y_1 and y_2 are elements of β. This is just an uninterpreted datatype: the intuition is that y_1 and y_2 are fragments of state data.

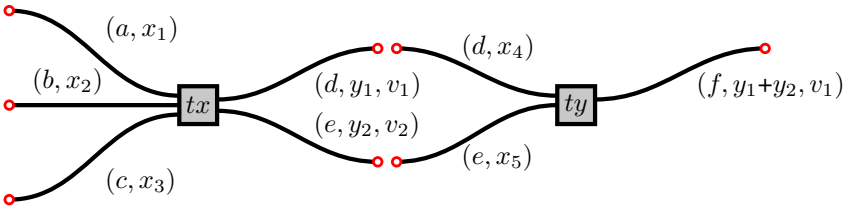


Fig. 2. A pair of transactions tx and ty

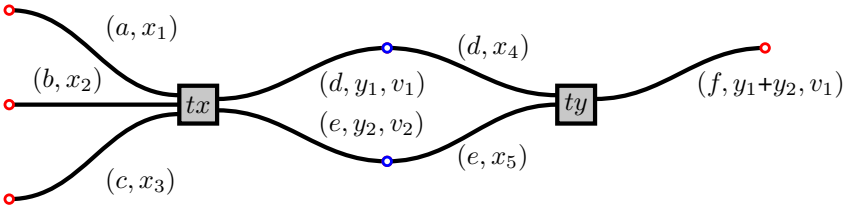


Fig. 3. A pair of transactions tx and ty, successfully validated and combined

Note that state data is stored per-output and not per-transaction. We suppose for concreteness that $\beta = \mathbb{N}$, so state data can be summed (as we do in the output f of ty; more in this shortly).
Note that $\mathsf{Transaction}$ is a *subset* of $fin(\mathsf{Input}) \times fin(\mathsf{Output})$ in Figure 2; which subset, is a parameter of the model selected. So for example we could enforce that all state data on all inputs and outputs should be equal, and this would in effect ensure that state data is a per-transaction quantity. This may or may not be what we want: the definition allows us to choose.

— v_1 and v_2 are validators. Their role is, intuitively, to decide which transactions tx will interact with (precise definition in Definition 3.4.1(4)).

$ty \in \mathsf{Transaction}$ is another transaction, with two inputs and one output. Note that its inputs are located at d and e, meaning that the inputs of ty *point to* the outputs of tx (Definition 3.2.6). Note also that it performs an addition, in the sense that the state data of its single output is the sum of the state data on its two inputs.

If in addition the following validation conditions are satisfied (@ from Notation 3.1.3)

— $(y_1, ty@(d, x_4)) \in \nu(v_1)$
(in words: validator v_1 in state y_1 validates the pointed transaction $ty@(d, x_4)$)), and
— $(y_2, ty@(e, x_5)) \in \nu(v_2)$
(in words: validator v_2 in state y_2 validates the pointed transaction $ty@(e, x_5)$)

then the two-element sequence $[tx, ty]$ is considered to be a valid combination. In the terminology we define later, this is called a *chunk* (Definition 3.4.1).

Supposing this is so. Then we can join the two transactions as illustrated in Figure 3, using blue circles to indicate successful validations. Congratulations: we have performed our first blockchain concatenation.

Note that *both* validations must succeed for the combination $[tx, ty]$ to be considered a valid chunk. More diagrams and discussion follow in Example 3.4.11. We now return to the definitions.

Notation 3.1.5. Sets that do not include atoms — including $\mathbb{N}$, inductive types built using $\mathbb{N}$, and function types built using $\mathbb{N}$ — are automatically equivariant. Thus, if the reader unfamiliar with nominal techniques and ZFA wonders whether particular choices they need for α, β, and $\mathsf{Validator}$ are equivariant — then the answer is 'yes'.

We may elide equivariance conditions Definition 3.1.2 henceforth. Any such type-like definition will be equivariant — i.e. closed under taking orbits of the permutation action — unless stated otherwise.

Remark 3.1.6. Equivariance comes from the underlying ZFA universe. Notation 3.1.5 can be viewed as an assertion that the definition exists in the category of equivariant ZFA sets and equivariant functions between them (or if the reader prefers: sets with a permutation action, and equivariant functions between them).

This paper will be light on sets and categorical foundations: we use just enough so that readers from various backgrounds get a hook on the ideas that speaks to them, and so it is always clear what is meant and how it could be made fully formal.

Note that just because a set is equivariant does not mean all its elements must be; for instance, $\mathbb{A}$ is equivariant (and consists of a single permutation orbit), but none of its elements $a \in \mathbb{A}$ are equivariant.

Remark 3.1.7. Compare Figure 1 with [CCM+20, Figure 3]:[13] α here corresponds to *redeemer* there, β here corresponds to *datum* there (though α here lives on inputs and β lives on outputs, whereas there both redeemer and datum live on inputs); the by-hash referencing there is replaced here with a nominal treatment using atoms; and validators exist on the output here and on the input there. There is less to this latter difference (validator moving from input in [CCM+20] to output here) than meets

[13] These correspondences are not intended to be read as mathematically precise; they illustrate how a shared intuitive underlying structure manifests itself across the two different definitions.

the eye: what is key is the interaction between an output and a later input, so it a matter of perspective and our convenience whether we view the output as validating the input, or vice versa, or indeed both.

Remark 3.1.8. Definition 3.1.2(4) uses a subset inclusion, whereas Definition 3.1.2(5) uses an injection ν. Why?

First, in practice we would expect $\nu(v)$ to be a *computable* subset of $\beta \times \mathsf{Transaction}_!$, since we have implementations in mind (though nothing in the mathematics to follow will depend on this).

Also, sets are well-founded, so a pure subset inclusion solution to Figure 1, for both clauses, would be difficult; we use ν to break the downward chain of sets inclusions.[14]

Yet just as we may write

$$\mathbb{N} \subseteq \mathbb{Q} \subseteq \mathbb{R},$$

eliding (or neglecting to consider) that their realisations may differ (finite cardinals vs. equivalence classes of pairs vs. Dedekind cuts), so — since ν is an injection — it may be convenient to treat $\mathsf{Validator}$ as a literal subset of $pow(\beta \times \mathsf{Transaction}_!)$.[15] This is standard, provided we clearly state what is intended and are confident that we could unroll the injections if required.

Thus Definition 3.1.9 rephrases Definition 3.1.2, with a focus on extensional sets behaviour rather than on internal structure (and for reference, see Definition 6.2.1 for an example of where the internal structure is required):

Definition 3.1.9. An IEUTxO model $\mathbb{T}$ from Definition 3.1.2 can be presented modulo Remark 3.1.8 as a tuple

$$\mathbb{T} = (\alpha_{\mathbb{T}}, \beta_{\mathbb{T}}, \mathsf{Transaction}_{\mathbb{T}}, \mathsf{Validator}_{\mathbb{T}})$$

of sets that solves the equations in Figure 1. We may drop the $\mathbb{T}$ subscripts where these are clear from the context.

3.2. Positions

Remark 3.2.1. The *positions* of a transaction are intuitively the interfaces or channels along which it may connect with other transactions to form a chunk (see next subsection). The notion of connection is called *pointing to* (Definition 3.2.6).

In Figure 3, the positions of tx and ty are $\{a, b, c, d, e\}$ and $\{d, e, f\}$ respectively, and input d of ty points to output d of tx. Also, tx is *earlier* than ty, and ty is *later* than tx.

Notation 3.2.2.

(1) If $tx \in \mathsf{Transaction}$ appears in $txs \in [\mathsf{Transaction}]$ then write $tx \in txs$.
(2) If $tx, tx' \in txs$ and tx appears before tx' in txs, then call tx **earlier** than tx' and tx' **later** than tx (in txs).
(3) If $tx = (I, O) \in \mathsf{Transaction}$ and $o \in \mathsf{Output}$, say o **appears in** tx and write $o \in tx$ when $o \in O$; similarly for an input $i \in \mathsf{Input}$.
We may silently extend this notation to larger data structures, writing e.g. $i \in txs$ when $i \in tx \in txs$ for some tx.

Definition 3.2.3. Suppose $\mathbb{T}$ is an IEUTxO model. We define **positions of**, written pos, as in Figure 4 (see also Definition 3.4.5).

Remark 3.2.4. Intuitively, $pos(x)$ collects the positions mentioned explicitly on the inputs or outputs of a structure. Note that validators may also act depending on positions of their inputs, but this

[14] A universe of non-wellfounded sets [Acz88] would also solve this. That might be overkill for this paper, but this might become relevant if the theorems of this paper are implemented in a universe with more direct support for non-wellfounded objects than ZFA provides.

[15] This can also depend on which aspects of a structure we wish to emphasise. For example, $\mathbb{N}$ and ω are isomorphic, as are $pow(X)$ and 2^X: the usage chosen in context suggests which aspects of the underlying entity (arithmetic vs. ordinals; sets vs. functions) we find most relevant.

$$\begin{array}{rl} i = (p,k) \in \mathsf{Input} & pos(i) = \{p\} \\ o = (p,d,V) \in \mathsf{Output} & pos(o) = \{p\} \\ tx = (I,O) \in \mathsf{Transaction} & pos(tx) = \bigcup\{pos(x) \mid x \in I \cup O\} \\ ctx = ((I,i),O) \in \mathsf{Transaction}_! & pos(ctx) = pos(I,O) \\ txs \in [\mathsf{Transaction}] & pos(txs) = \bigcup\{pos(tx) \mid tx \in txs\} \end{array}$$

Fig. 4. Positions of

information is not detected by pos. For instance, consider a (arguably odd, but imaginable) output o having the form

$$o = (b, 0, \{i \in \mathsf{Input} \mid pos(i) = \{a\}\}).$$

So this output is at position b, $\beta = \mathbb{N}$ and o carries data 0, and o has a validator that validates an input precisely when it is at position a. Then $pos(o) = \{b\}$.[16]

Lemma 3.2.5. *Suppose* $\mathbb{T}$ *is an IEUTxO model and* $txs \in [\mathsf{Transaction}]$. *Then*

$$pos(txs) = \varnothing \quad \textit{implies} \quad txs = [].$$

Proof. Recall from Definition 3.1.2(4) that the empty transaction is disallowed. Now examining Figure 4 we see that the only way txs can mention no positions *at all*, is by having no transactions and so being empty. □

Definition 3.2.6. (1) Suppose $i \in \mathsf{Input}$ and $o \in \mathsf{Output}$. Then say that i **points to** o and write $i \mapsto o$ when they share a position:

$$i \mapsto o \quad \text{when} \quad pos(i) = pos(o).$$

(The use of 'point' here is unrelated to the 'pointed sets' from Notation 2.2.3.)

(2) Recall the notation $tx@i$ from Notation 3.1.3. Suppose that:

$$\begin{array}{l} i = (p,k) \in \mathsf{Input} \\ i \in tx \in \mathsf{Transaction} \text{ and} \\ o = (p,d,V) \in \mathsf{Output}. \end{array}$$

Then write

$$validates(o, tx@i) \quad \text{when} \quad (d, tx@i) \in V \tag{1}$$

and say that the output o **validates** the input-in-context $tx@i$.

3.3. Why we have α and β

We are now ready to more rigorously explain the roles of α and β in Figure 1. We touched on this already in Subsection 2.1 (α is 'keys'; β is 'data'), but now we can be more specific in our discussion:

— α is intuitively a datatype for *keys*.
If $i = (a,k)$, then k is supposed to be a cryptographic secret that we need to attach to a suitable unspent output (e.g. a solution to a cryptographic puzzle posted by its validator).
— β is intuitively a datatype of abstract data.
If $o = (p,d,V)$ then d stores some kind of state (for instance, an account balance).

We should note:

(1) Nothing mathematical enforces this usage. α and β are abstract type parameters, to instantiate as we please.

[16] Note to experts in nominal techniques: so the support of validators, if any, is not counted in pos. See also the discussion in Subsection 8.4.

(2) β is redundant and can be isomorphically removed.
We briefly sketch a suitable isomorphism: We set $\mathsf{Output} = \mathbb{A} \times \mathsf{Validator}$ and $\mathsf{Validator} \subseteq pow(\mathsf{Transaction}_!)$ in Figure 1. α is replaced by $\alpha \times \beta$, and information that was stored as $d \in \beta$ on an output would instead be stored in the validator of that output, which would be set to accept only inputs with β-component d.
(3) If we only care about countable datatypes and computable data (a reasonable simplification), then everything could be Gödel encoded into $\mathbb{N}$. So we could then just fix $\alpha = \mathbb{N}$.

But there is a design tension here: we want something that is compact, but also implementable (e.g. as proof-of-concept reference code; see Remark 3.3.1). Having explicit types of keys α and data β is useful for clarity, so even though α and β introduce (a little) redundancy, the cost is minimal and the practical returns worthwhile.

Furthermore, in the real world validators are generated by code. This validator code is typically given state data which is carried on the same output as the validator is stored, to help the validator decide whether to validate prospective input. So when we write $\mathsf{Validator} \subseteq pow(\beta \times \mathsf{Transaction}_!)$, this effectively makes the validator into a function over *local state data* and a transaction input.

We see this happen concretely in equation (1) in Definition 3.2.6: we have $o = (p, d, V)$ and we pass the state data d of o to V the validator of o —

$$(d, tx@i) \in V$$

— even though this is mathematically redundant, since d and V are both in o so the d could in principle be curried into the V. We retain β and set $\mathsf{Validator} \subseteq pow(\beta \times \mathsf{Transaction}_!)$ rather than just $\mathsf{Validator} \subseteq pow(\mathsf{Transaction}_!)$, to reflect this practical architecture.

Remark 3.3.1. The discussion above is more than hypothetical and corresponds to the experience of creating executable Haskell code. The IEUTxO type equations in this paper have been realised as a Haskell implementation; a reference system has been implemented following it; and some of the results of this paper converted into QuickCheck properties. The package is on Hackage and GitHub [Gab20b].[17]

3.4. Chunks and blockchains

3.4.1. Chunks. Definition 3.4.1 (chunks) is a central concept in this paper, so we will summarise it twice: once now, and again after the definition:

> A list of transactions is a *chunk* when all input positions are distinct, all output positions are distinct, and all inputs point to at most one earlier validating output.

Examples are illustrated in Example 3.4.11 (along with examples of blockchains). In formal detail:

Definition 3.4.1. Suppose $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator})$ is an IEUTxO model. Call a transaction-list $txs \in [\mathsf{Transactions}]$ a **chunk** when:

(1) Distinct outputs appearing in txs have distinct positions.
(2) Distinct inputs appearing in txs have distinct positions.
It may be that the position of an input $i \in tx \in txs$ equals that of some output $o \in tx' \in txs$, or using the notation of Definition 3.2.6(1): $i \mapsto o$. If so, by condition 1 there is at most one such. We write the **unique output pointed to by** i

$$txs(i) \in \mathsf{Output}$$

[17] There is no claim to have created a practical blockchain implementation (there is no cryptographic assurance, for example), but the Haskell code comes out as a surprisingly direct translation of the maths in this paper, and demonstrates that this paper has enough executable content to be a plausible reference implementation, for example, within the framework of its own abstractions.

In particular, the design of Figure 1 was informed by the experience of creating executable code, and having types α and β explicitly available, was useful.

where it exists, so $o = txs(i) \in tx' \in txs$.
(3) For every input $i \in tx \in txs$, if the output $txs(i)$ is defined then that output must occur in a transaction tx' that is strictly earlier than tx in txs.
(4) For every $i \in tx \in txs$, if $txs(i)$ is defined then $validates(txs(i), tx@i)$ (Definition 3.2.6(2)).

REMARK 3.4.2. So to sum Definition 3.4.1 up in a single line:

> a *chunk* is a list of transactions such that a position can only ever be shared between a single pair of an *earlier* output o to a *later* input i, which o *validates* — and otherwise positions are distinct.

NOTATION 3.4.3. (1) Write

$$\mathsf{Chunk}_{\mathbb{T}} \subseteq [\mathsf{Transaction}_{\mathbb{T}}] \quad \text{for the set of chunks over } \mathbb{T}.$$

We may drop the $\mathbb{T}$ subscripts; the meaning will always be clear.
(2) We may also call a transaction-list **valid**, when it is a chunk. That is: chunks are precisely the *valid* transaction-lists.

REMARK 3.4.4. The way we have formulated the structure of chunks in Definition 3.4.1 reminds this author of the π-calculus, where positions correspond to π-calculus channel names (and outputs are outputs and inputs are inputs).

When we have this intuition in mind, we may occasionally call positions **channels**, as in the *blocked channels* of Subsection 5.5. See also the discussion in Remark 4.3.4.

With the intuition of Remark 3.4.4 in mind, we give a simple definition, which refines Definition 3.2.3:

DEFINITION 3.4.5. Suppose $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator})$ is an IEUTxO model and suppose $tx = (I, O) \in \mathsf{Transaction}$ is a transaction. Then define

— the **input channels** or **positions** $input(tx) \subseteq_{fin} \mathbb{A}$ of tx to be the finite set of atoms that are positions of inputs in tx, and
— the **output channels** or **positions** $output(tx) \subseteq_{fin} \mathbb{A}$ of tx to be the finite set of atoms that are positions of outputs in tx.

In symbols:

$$input(tx) = \{p \mid (p, k) \in I\}$$
$$output(tx) = \{p \mid (p, d, V) \in O\}.$$

An important special case of Notation 3.4.3 is when the chunk is a singleton list, i.e. it contains just one transaction:

LEMMA 3.4.6. *Suppose $\mathbb{T}$ is an IEUTxO model and suppose $tx \in \mathsf{Transaction}$. Then*

$$[tx] \in \mathsf{Chunk} \quad \textit{if and only if} \quad input(tx) \cap output(tx) = \varnothing.$$

Proof. We consider the conditions in Definition 3.4.1 and see that condition 3 forces the input and output channels of the transaction to be disjoint, and then none of the other conditions are relevant. □

REMARK 3.4.7. Definition 3.4.5 refines Definition 3.2.3, and another way to phrase Lemma 3.4.6 is that $[tx]$ is a chunk precisely when $pos(tx) = inputs(tx) \uplus outputs(tx)$, where $\uplus$ denotes disjoint sets union.

3.4.2. UTxOs, UTxIs ...

DEFINITION 3.4.8. Suppose $\mathbb{T}$ is an IEUTxO model and $txs \in [\mathsf{Transaction}]$.

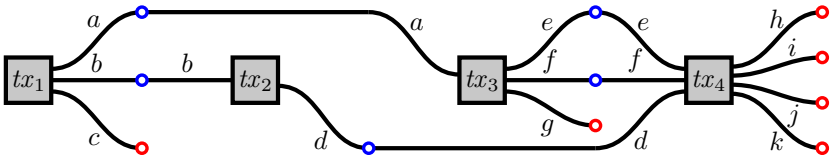


Fig. 5. A blockchain $\mathcal{B} = [tx_1, tx_2, tx_3, tx_4]$

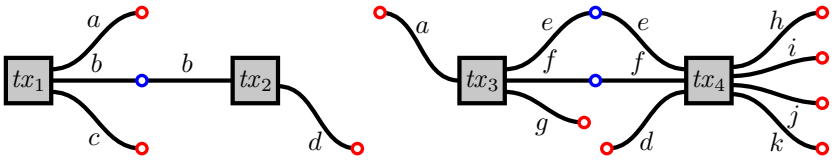


Fig. 6. $\mathcal{B}$ chopped up as a blockchain $[tx_1, tx_2]$ and a chunk $[tx_3, tx_4]$

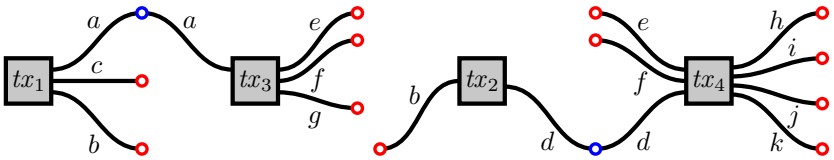


Fig. 7. $\mathcal{B}$ chopped up as a blockchain $[tx_1, tx_3]$ and a chunk $[tx_2, tx_4]$

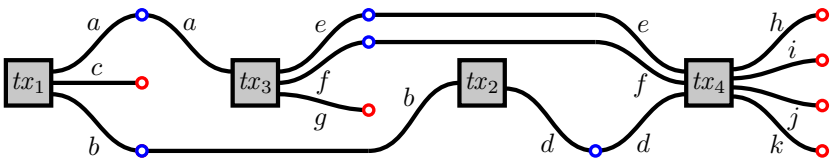


Fig. 8. The blockchain $\mathcal{B}' = [tx_1, tx_3, tx_2, tx_4]$

(1) If $i \in tx \in txs$ and $txs(i)$ is not defined[18] then call the unique atom $a \in pos(i) \subseteq \mathbb{A}$ an **unspent transaction input**, or **UTxI**. Write

$$utxi(txs) \subseteq_{fin} \mathbb{A} \quad \text{for the UTxIs of } txs.$$

(2) If $o \in tx \in txs$ and $o \neq txs(i)$ for all later $i \in tx \in txs$[19] then call the unique atom $a \in pos(o) \subseteq \mathbb{A}$ an **unspent transaction output**, or **UTxO**. Write

$$utxo(txs) \subseteq_{fin} \mathbb{A} \quad \text{for the set of UTxOs of } txs.$$

(3) If $o \in tx \in txs$ and $o = txs(i)$ for some later $i \in tx \in txs$, then call the atom $a \in pos(o) \subseteq \mathbb{A}$ a **spent transaction channel**, or **STx**. Write

$$stx(txs) \subseteq_{fin} \mathbb{A} \quad \text{for the set of STxs of } txs.$$

REMARK 3.4.9. Some comments on the interpretation of $utxi$ and $utxo$ and stx from Definition 3.4.8:

[18] ...so i does not point to an earlier validating output in txs...
[19] ...so o does not validate some later input in txs ...

$utxi(txs)$, $utxo(txs)$, and $stx(txs)$ are all finite sets of atoms, but we interpret them somewhat differently:

(1) Intuitively, an atom $a \in utxo(txs)$ identifies an output $o \in tx \in txs$ with position a. So $utxo(txs)$ is a finite set of names of outputs in txs.
(2) Intuitively, an atom $a \in utxi(txs)$ identifies an input-in-context $tx@i$, for $i \in tx \in txs$ with $pos(i) = a$.
We say this because the validator of an output takes as argument an input-in-context $tx@i \in \mathsf{Transaction}_!$. So $utxi(txs)$ is a finite set of names for inputs-in-contexts.
(3) Intuitively, an atom $a \in stx(txs)$ identifies a pair of an output and the input-in-context that spends it. Thus a could be thought of as this pair, or a could be thought of as an edge in a graph that joins a node representing the output, to a node representing the input.
So $stx(txs)$ is a finite set of internal names of already-spent communications between outputs and inputs-in-context within txs.

3.4.3. . . . and blockchains. With the machinery we have now have, it is quick and easy to define blockchains:

Definition 3.4.10. A **blockchain** is a chunk $ch \in \mathsf{Chunk}$ such that $utxi(ch) = \varnothing$. In words:

> a blockchain is a chunk with no unspent inputs.

Diagrammatic examples follow in Example 3.4.11:

Example 3.4.11. Recall we observed in Subsection 2.1 that (in the terminology that we now have) the key operation of an IEUTxO is to attach a transaction's inputs to a chunk's outputs.

Example transaction-lists, blockchains, and chunks are illustrated in Figures 2, 3, 5, 6, 7, and 8.[20]

In the Figures, a blue circle denotes a validator on an output at some position $(a, b, c, \dots)$ that has accepted an input and connected to it; and a red circle denotes an unspent input or output, meaning one that has not connected up with a validator to form a spent output-input pair:

(1) $\mathcal{B}$, $\mathcal{B}'$, $[tx_1, tx_2]$ and $[tx_1, tx_3]$ are blockchains, because they have unspent outputs (in red) but no unspent inputs.
In these blockchains, tx_1 is what is called the **genesis block**, meaning the first block in the chain. It follows from the definitions that the genesis block has no inputs.[21]
(2) $[tx]$ (Figure 2) and $[tx_1]$, $[tx_3, tx_4]$ and $[tx_2, tx_4]$ are chunks, but not blockchains because they have unspent inputs (in red).
(3) $[tx_2, tx_1]$ is neither a blockchain nor chunk, because the b-input of tx_2 points to the later b-output of tx_1. It is just a list of transactions.

We note two alternative characterisation of blockchains (Definition 3.4.10):

Lemma 3.4.12. *A chunk is a blockchain when . . .*

(1) . . . the 'at most one' in Definition 3.4.1(2) is strengthened to 'precisely one'.
(2) . . . the function $i \mapsto txs(i)$ (Definition 3.4.1(2)) is a total function on the inputs in txs (so that every input points to precisely one output in an earlier transaction).

Remark 3.4.13. We step back to reflect on Definition 3.4.10. This is supposed to be a paper about blockchains; why did it take us this long to get to them? Because they are a special case of something better and more pertinent: chunks.

[20] These diagrams are adapted from [BG20], with my coauthor's agreement.
[21] If we permitted loops, i.e. connections from a later output to an input that is on the same block or earlier, then a genesis block might have an input that addresses an output on the same or a later block. See Subsection 8.6(2). But for the definitions as set up in this paper, that is not allowed.
Note that our definitions admit a blockchain with two genesis blocks; it suffices to have more than one transaction with no inputs. Whether this is a feature or a bug depends on the application.

A blockchain is just a left-closed chunk. There is nothing wrong with blockchains, but mathematically, chunks seem more interesting:

(1) A sublist of a blockchain is a chunk, not a blockchain (we prove this in a moment, in Corollary 3.5.2).
(2) A composition of blockchains is possible, but uninteresting; whereas composition of chunks is clearly an interesting operation.[22]
(3) If we cut a blockchain into n pieces then we get one *blockchain* (the initial segment) ... and $n-1$ *chunks*.
(4) Chunks can in any case be viewed as a natural generalisation of blockchains, to allow UTxIs as well as UTxOs.

Definitions and results like Definition 3.5.3, Theorem 3.5.4, and Lemma 3.5.9 inhabit a universe of chunks, not blockchains.

Even in implementation, where we care about real blockchains on real systems, a lot of development work goes into allowing users in practice to download only partial histories of the blockchain rather than having to download and store a complete record — the motivation here is practical, not mathematical — and in the terminology of this paper, we would say that for efficiency we may prefer to work with chunks where possible, because they can be partial and so can be more lightweight.

So the focus of this paper is on chunks: they generalise blockchains, have better mathematical structure; and chunks are in any case where we arrive even if we *start off* asserting (de)compositional properties of blockchains; and finally — though this is not rigorously explored in this paper, but we would suggest that — chunks are also where we arrive when we consider space-efficient blockchain implementations.

Finally, we mention that blockchains have a right monoid action given by concatenating chunks. Thus, by analogy here with rings and modules, we could imagine for future work a mathematics of blockchains generalising Definition 3.4.10 such that a 'blockchain set' is just any set with a suitable chunk action.

3.5. Properties of chunks and blockchains

3.5.1. Algebraic and closure properties of chunks. Lemma 3.5.1 expresses that a list of transactions is a valid chunk if and only if every sublist of it of length at most two, is a valid chunk. In this sense, (in)validity is a *local* phenomenon:

LEMMA 3.5.1. *Suppose* $\mathbb{T} = (\alpha_{\mathbb{T}}, \beta_{\mathbb{T}}, \mathsf{Transaction}_{\mathbb{T}}, \mathsf{Validator}_{\mathbb{T}})$ *is an IEUTxO model, and suppose* $[tx_1, \ldots, tx_n] \in [\mathsf{Transaction}]$. *Then the following conditions are equivalent:*

— $[tx_i, tx_j] \in \mathsf{Chunk}$ *for every* $1 \leq i < j \leq n$ [23]
— $[tx_1, \ldots, tx_n] \in \mathsf{Chunk}$

Proof. We note of the well-formedness conditions on chunks from Definition 3.4.1 that they all concern relationships involving at most two transactions. □

COROLLARY 3.5.2 (Validity is down-closed). *Suppose we have an IEUTxO model* $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator})$ *and* $l, l' \in [\mathsf{Transaction}]$.

Recall from Notation 2.2.3 that we order lists by sublist inclusion, so $l' \leq l$ *when* l' *is a sublist of* l. *Then*

$$ch \in \mathsf{Chunk} \ \wedge \ l' \in [\mathsf{Transaction}] \ \wedge \ l' \leq ch \quad \textit{implies} \quad l' \in \mathsf{Chunk}.$$

In words: every sublist of a chunk, is itself a chunk.

[22] Blockchains have no unspent inputs, so if composed they just sit side-by-side and do not communicate. Contrast this with chunks, for which the partial monoidal composition is clearly natural.
[23] This condition holds trivially if $n = 0$, i.e. for the empty list.

Proof. From Lemma 3.5.1. □

We can wrap up Corollary 3.5.2 in a nice mathematical package:

DEFINITION 3.5.3. Suppose $(X, \leq, \bullet, \mathsf{e})$ has the following structure:

(1) X is a set.
(2) $(X, \leq)$ is a partial order, for which e is a bottom element.
(3) $\bullet$ is a **partial monoid** action on X, meaning that $(x \bullet y) \bullet z$ exists if and only if $x \bullet (y \bullet z)$ exists, and if both exist then they are equal.[24]
(4) $\bullet$ is **down-closed**, meaning that if $x' \leq x$ and $x \bullet y$ exists, then so does $x' \bullet y$, and similarly for $y \bullet x$ and $y \bullet x'$.
(5) $\bullet$ is **monotone** where defined, meaning that if $x' \leq x$ then $x' \bullet y \leq x \bullet y$ (provided $x \bullet y$ exists), and similarly for $y \bullet x$ and $y \bullet x'$.

In this case, call $(X, \leq, \bullet, \mathsf{e})$ a **partially-ordered partial monoid**.

THEOREM 3.5.4. *Suppose $\mathbb{T}$ is an IEUTxO model (Definition 3.1.9).*
Then its set of chunks $\mathsf{Chunk}_{\mathbb{T}}$ (Definition 3.4.1) forms a partially-ordered partial monoid (Definition 3.5.3), where

— $\leq$ is sublist inclusion,
— $\bullet$ is list concatenation, and
— the unit element is $[]$ the empty set.

Proof. By facts of lists, and Corollary 3.5.2. □

3.5.2. Some observations on observational equivalence

REMARK 3.5.5. Lemmas 3.5.6 and 3.5.9 apply to IEUTxO models and essentially give criteria for observational equivalence when positions are disjoint.
We find them echoed in the theory of abstract chunk systems as Definitions 4.4.1(5) and 4.4.1(4), and we need them for Proposition 4.4.8.

LEMMA 3.5.6. *Suppose $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator})$ is an IEUTxO model and $ch, ch' \in \mathsf{Chunk}_{\mathbb{T}}$. Then*

$$pos(ch) \cap pos(ch') = \varnothing \quad \textit{implies} \quad ch \bullet ch' \in \mathsf{Chunk}_{\mathbb{T}}.$$

Proof. By routine checking of possibilities, using the fact that if $pos(ch) \cap pos(ch') = \varnothing$ (Definition 3.2.3) then they have no positions in common, so no output in one can be called on to validate an input in the other. □

DEFINITION 3.5.7. Suppose $ch, ch' \in \mathsf{Chunk}_{\mathbb{T}}$. Then call ch and ch' **commuting** when

$$ch \bullet ch' \in \mathsf{Chunk}_{\mathbb{T}} \iff ch' \bullet ch \in \mathsf{Chunk}_{\mathbb{T}}.$$

REMARK 3.5.8. Definition 3.5.7 is clearly a notion of observational equivalence between $ch \bullet ch'$ and $ch' \bullet ch$ where the observable is 'forms a valid chunk with'. This observable does not depend on internal structure, so we will develop it further once we have abstract chunk systems; see Definition 4.3.16.
For now, Definition 3.5.7 gives us just enough of the background theory of observational equivalence, to state and prove Lemma 3.5.9, Proposition 3.5.12, and Theorem 3.6.1.

LEMMA 3.5.9. *Suppose $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator})$ is an IEUTxO model. Then:*

(1) If $tx, tx' \in \mathsf{Transaction}$ and $pos(tx) \cap pos(tx') = \varnothing$ (Definition 3.2.3) then the following all hold:

$$[tx, tx'] \in \mathsf{Chunk} \iff [tx', tx] \in \mathsf{Chunk} \iff [tx], [tx'] \in \mathsf{Chunk}$$

[24] A slightly weaker possibility is that $(x \bullet y) \bullet z$ and $x \bullet (y \bullet z)$ need not exist or not exist together, but *if* they both exist, then they are equal. This is the natural notion *if* we bind names of spent output-input pairs. More discussion in Subsection 8.2.

(2) *As a corollary, if* $ch, ch' \in \mathsf{Chunk}$ *and* $pos(ch) \cap pos(ch') = \varnothing$ *then* ch *and* ch' *are commuting (Definition 3.5.7).*

Proof. (1) By routine checking of possibilities, using the fact that if $pos(tx) \cap pos(tx') = \varnothing$ then they have no positions in common, so no output in one can be called upon to validate an input in the other.
(2) It is a fact that if $tx \in l$ then $pos(tx) \subseteq pos(l)$ and similarly for $tx' \in l'$. The corollary now follows by a routine argument from part 1 of this result and Lemma 3.5.1.

□

3.5.3. Properties of UTxOs and UTxIs. We return to Definition 3.4.8: Lemma 3.5.10 uses Definition 3.4.8 to note some simple properties of Definition 3.4.1.

Lemma 3.5.10. *Suppose* $\mathbb{T}$ *is an IEUTxO model and* $ch, ch' \in \mathsf{Chunk}_{\mathbb{T}}$ *Then:*

(1) $utxi(ch) \cap utxo(ch) = \varnothing$
(2) *If* $ch \cdot ch' \in \mathsf{Chunk}_{\mathbb{T}}$ *then* $pos(ch) \cap pos(ch') \subseteq utxo(ch) \cap utxi(ch')$.
(3)
$$\begin{aligned} \varnothing &= utxi(ch) \cap stx(ch) \\ \varnothing &= utxo(ch) \cap stx(ch) \\ pos(ch) &= utxi(ch) \uplus utxo(ch) \uplus stx(ch) \qquad \uplus \textit{ is disjoint union} \end{aligned}$$

Proof. (1) An input cannot point to a later output, because of Definition 3.4.1(3), and if it points to an earlier output then by construction in Definition 3.4.8 this position no longer labels a UTxO or UTxI. Furthermore a position can be used at most once in an input-output pair, by Definition 3.4.1(2).
(2) From Definition 3.4.1(3), as for the previous case.
(3) All facts of Definition 3.4.8 and Figure 4.

□

Remark 3.5.11. Proposition 3.5.12 can be viewed as a stronger version of Lemma 3.5.6. It is an important result because it relates the following apparently different observables:

(1) A statically observable property, that ch and ch' mention disjoint sets of positions.
(2) A locally observable property, that ch and ch' compose in both directions.
(3) An abstract global observable, that $ch \cdot ch'$ and $ch' \cdot ch$ can be commuted in any larger chunk.

Compare also with Proposition 4.4.8, which is a similar result but for the differently-constructed abstract chunk systems.

Proposition 3.5.12. *Suppose* $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator})$ *is an IEUTxO model and* $ch, ch' \in \mathsf{Chunk}_{\mathbb{T}}$. *Then the following are equivalent:*

(1) $pos(ch) \cap pos(ch') = \varnothing$
(2) $ch \cdot ch' \in \mathsf{Chunk}_{\mathbb{T}} \wedge ch' \cdot ch \in \mathsf{Chunk}_{\mathbb{T}}$.
(3) ch *and* ch' *are commuting.*

Proof. The top-to-bottom implication is Lemma 3.5.6.

For the bottom-to-top implication, suppose that $ch \cdot ch', ch' \cdot ch \in \mathsf{Chunk}_{\mathbb{T}}$. From Lemma 3.5.10(2) we have

$$pos(ch) \cap pos(ch') \subseteq (utxo(ch) \cap utxi(ch')) \cap (utxo(ch') \cap utxi(ch)).$$

We can rearrange this:

$$pos(ch) \cap pos(ch') \subseteq (utxi(ch) \cap utxo(ch)) \cap (utxi(ch') \cap utxo(ch')).$$

Now we use Lemma 3.5.10(1).

The final part is direct from Lemma 3.5.9(2). □

REMARK 3.5.13. Proposition 3.5.12 is a *resource separation* result: if two chunks depend on disjoint resources (disjoint sets of positions) then they commute. This is in the spirit of separation logic [Rey02], which is a family of logics for reasoning about programs with resource separation in programs — intuitively, that if two programs depend on disjoint resources (e.g. channels or pointers), then they should not interfere with one another, just as we see in Proposition 3.5.12.

It is also in the spirit of a short paper [Nes21] (which appeared after this paper went into initial review) which uses monoidal categories to reason on resources in a broadly similar spirit, albeit using different methods. A quote from that paper makes a related point: *We have seen how the resource theoretic interpretation of monoidal categories, and in particular their string diagrams, captures the sort of material history that concerns ledger structures for blockchain systems.* More on this in the Conclusions.

We conclude with Lemma 3.5.14, which we will need later in Lemma 5.2.1:

LEMMA 3.5.14. *Suppose* $\mathbb{T} \in \mathsf{IEUTxO}$ *is an IEUTxO model and* $\pi \in Perm$ *is a permutation of atoms and* $txs \in [\mathsf{Transaction}_{\mathbb{T}}]$. *Then:*

$$\begin{aligned} f(\pi \cdot txs) &= \pi \cdot f(txs) && \text{for } f \in \{utxi, utxo, stx, pos\} \\ &= \{\pi(a) \mid a \in f(txs)\} \end{aligned}$$

In the terminology of Definition 2.3.5: $utxi$, $utxo$, stx, *and* pos *are all equivariant.*

Proof. Direct from Figure 4 and Definitions 3.4.8 and 2.3.3. □

3.6. An application: UTxO systems are 'Church-Rosser', in a suitable sense

We now come to Theorem 3.6.1 which is an application of our machinery so far:

THEOREM 3.6.1 (Church-Rosser for UTxO). *Suppose* $\mathbb{T} \in \mathsf{IEUTxO}$ *and* $y, x, x' \in \mathsf{Chunk}_{\mathbb{T}}$. *Suppose further that*

$$y \bullet x \bullet x' \in \mathsf{Chunk}_{\mathbb{T}} \quad \text{and} \quad utxi(y \bullet x') = utxi(y \bullet x \bullet x').$$

Then we have that:

(1) x *and* x' *are commuting (Definition 3.5.7).*
(2) $y \bullet x' \bullet x \in \mathsf{Chunk}_{\mathbb{T}}$.

Proof. We know by Corollary 3.5.2 (because $y \bullet x \bullet x'$ is a chunk) that $x \bullet x'$ is a chunk, so $pos(x) \cap pos(x') \subseteq utxo(x) \cap utxi(x')$.

We also know that $utxi(y \bullet x') = utxi(y \bullet x \bullet x')$ and it follows from Definition 3.4.8 that $(utxo(y) \cap utxi(x) = \varnothing$ and) $utxo(x) \cap utxi(x') = \varnothing$.[25]

Therefore $pos(x) \cap pos(x') = \varnothing$. By Proposition 3.5.12, x and x' are commuting, and it follows (since $y \bullet x' \bullet x \in \mathsf{Chunk}_{\mathbb{T}}$) that $y \bullet x \bullet x' \in \mathsf{Chunk}_{\mathbb{T}}$. □

Recall the definition of a blockchain (Definition 3.4.10) as being a chunk with empty $utxi$. Then we can specialise Theorem 3.6.1 as follows:

COROLLARY 3.6.2. *Suppose* $\mathbb{T} \in \mathsf{IEUTxO}$ *and* $y, x, x' \in \mathsf{Chunk}_{\mathbb{T}}$ *and suppose* $y \bullet x'$ *is a blockchain. Then:*

(1) *If* $y \bullet x \bullet x'$ *is a blockchain then* x *and* x' *commute.*
(2) *If* x *and* x' *do not commute then* $y \bullet x \bullet x'$ *is not a blockchain.*

Proof. Direct from Theorem 3.6.1, for the case that $utxi(y \bullet x') = utxi(y \bullet x \bullet x') = \varnothing$. □

REMARK 3.6.3. Corollary 3.6.2 models a situation where someone designs a chunk x' that will successfully attach to a blockchain y, but because this is a distributed system, somebody else gets in and attaches x first.

[25] We really use here the fact that names can only be used to link an output to an input *once*.

Then this can only happen if x and x' are commuting; conversely, if x and x' are not commuting then one of $y \cdot x \cdot x'$ or $y \cdot x'$ must fail to be a blockchain.

What makes this interesting is that it is an important correctness property from the point of view of the user who created x': *if* x' is accepted onto both y and $y \cdot x$ without failure, then it *does not matter* that x got in first — $y \cdot x \cdot x'$ and $y \cdot x' \cdot x$ are equivalent up to observable behaviour.

Note that:

— Theorem 3.6.1 and Corollary 3.6.2 do not state that IEUTxO systems are insensitive to order, and
— Theorem 3.6.1 and Corollary 3.6.2 do not state that all transactions always commute (this is simply not what is written on the page).

Theorem 3.6.1 and Corollary 3.6.2 can be viewed as a *purity* property, in the sense of functional programming: if x' successfully combines with y, then inputs to x' may not be modified by an intervening environment x — there are no side-effects!

More specifically, these results can be viewed as playing a role analogous to a 'Church-Rosser' or 'confluence' property. To see why, contrast with the situation in an accounts-based system — corresponding to an imperative paradigm — where a transaction may be successfully appended *even if* parameters to it on the blockchain get modified by intervening transactions.

To take a concrete scenario: I could check my bank account, observe I have enough money for a purchase, submit my transaction — and then go into overdraft and be subject to overdraft fees, because a direct debit happened to arrive in-between (a) my checking my balance and designing my purchase transaction and (b) the payment request for the purchase transaction arriving at my account. This error clearly comes from the use of a stateful, imperative programming style, and Theorem 3.6.1 expresses a rigorous sense in which a corresponding phenomenon is impossible in a UTxO-style system.[26]

With this comparison in mind, we see that Theorem 3.6.1 is a purity result: state is local, and composition of chunks succeeds or fails locally.

3.7. The category IEUTxO of IEUTxO models

We can organise our IEUTxO models into a category:

Definition 3.7.1. Let IEUTxO be a category such that:

(1) Objects $\mathbb{S}, \mathbb{T}$ are IEUTxO models (Definition 3.1.2).
(2) An arrow $f : \mathbb{S} \to \mathbb{T}$ is a map

$$f : \mathsf{Transaction}_{\mathbb{S}} \to \mathsf{Chunk}_{\mathbb{T}}$$

such that if $tx, tx' \in \mathsf{Transaction}_{\mathbb{S}}$ then

$$[tx, tx'] \in \mathsf{Chunk}_{\mathbb{S}} \quad \text{implies} \quad f(tx) \cdot f(tx') \in \mathsf{Chunk}_{\mathbb{T}}. \tag{2}$$

Above, $\cdot$ denotes monoid composition, which on chunks is list concatenation; see Notation 2.2.3(4).
(3) The identity arrow maps tx to $[tx]$.
(4) Composition of arrows is pointwise, meaning that if

$$\begin{array}{rr} f : \mathbb{S} \to \mathbb{S}' & tx \in \mathsf{Transaction}_{\mathbb{S}} \\ f' : \mathbb{S}' \to \mathbb{S}'' & f(tx) = [tx'_1, \ldots, tx'_n] \end{array}$$

then $f'f : \mathbb{S} \to \mathbb{S}''$ is such that

$$tx \in \mathsf{Transaction}_{\mathbb{S}} \longmapsto f'(tx'_1) \cdot \ldots \cdot f'(tx'_n) \in \mathsf{Chunk}_{\mathbb{S}''}.$$

We prove this mapping is indeed an arrow — thus, it maps to chunks — in Corollary 3.7.4.

[26] In practice, programmers of smart contracts on accounts-based blockchains may write explicit tests into their smart contracts that double-check values of input variables *at time of attachment to the blockchain*, if they anticipate this might be an issue. More discussion of this is in [BG20].

LEMMA 3.7.2. *An arrow $f : \mathbb{S} \to \mathbb{T}$ (Definition 3.7.1(2)) induces a mapping of chunks $\mathsf{Chunk}_{\mathbb{S}} \to \mathsf{Chunk}_{\mathbb{T}}$, by acting on the individual transactions and composing the results:*

$$f([tx_1, \ldots, tx_n]) = f(tx_1) \bullet \ldots \bullet f(tx_n).$$

Proof. The nontrivial part is to check that

— if $ch = [tx_1, \ldots, tx_n]$ is a valid chunk in $\mathsf{Chunk}_{\mathbb{S}}$,
— then $f(tx_1) \bullet \ldots \bullet f(tx_n)$ is a valid chunk in $\mathsf{Chunk}_{\mathbb{T}}$.

This follows by combining condition 2 of Definition 3.7.1 with Lemma 3.5.1 and Corollary 3.5.2. □

COROLLARY 3.7.3. *Condition 2 of Definition 3.7.1 is equivalent to either of the following conditions:*

(1) $[tx_1, \ldots, tx_n] \in \mathsf{Chunk}_{\mathbb{S}}$ *implies* $f(tx_1) \bullet \ldots \bullet f(tx_n) \in \mathsf{Chunk}_{\mathbb{T}}$.
(2) *f induces a monoid homomorphism on chunks.*

Proof. The equivalence of conditions 1 and 2 above is routine, given that every chunk factors into singletons. Then condition (2) in part 2 of Definition 3.7.1 is just a special case of condition 1 above, and the reverse implication is Lemma 3.7.2. □

COROLLARY 3.7.4. *Composition of arrows as given in Definition 3.7.1(4) is well-defined; that is, the composition $f'\,f$ really is a map from transactions to chunks.*

Proof. Continuing the notation of Definition 3.7.1(4), by assumption f maps $tx \in \mathsf{Transaction}_{\mathbb{S}}$ to some $f(tx) \in \mathsf{Chunk}_{\mathbb{S}'}$, and then by Lemma 3.7.2 the action of f' maps $f(tx)$ to a valid chunk $f'(f(tx)) \in \mathsf{Chunk}_{\mathbb{S}''}$. □

REMARK 3.7.5 (Comment on design). We briefly discuss the design decisions embedded in Definition 3.7.1:

(1) The conditions in Corollary 3.7.3 are more readable than condition (2) of Definition 3.7.1(2), but this comes at the cost of an additional universally quantified parameter n. It is a matter of taste which version we take as primitive: the one in the Definition has fewest parameters and is easiest to check (a higher-level view will be taken later when we develop abstract chunk systems in Section 4).
(2) We could relax the condition to allow f to be a partial map.
This would exhibit IEUTxO as a subcategory of a larger category with the same objects but more arrows, and in particular it would allow chunks in $\mathbb{S}$ to cease to be valid when mapped to $\mathbb{T}$ — we would still insist that f be a *partial* monoid homomorphism on chunks, where everything is defined.
We did not choose this design for this paper, but it might be useful for future work; e.g. following an intuition that $\mathbb{S}$ is a liberal universe of chunks, and f maps it to a stricter universe $\mathbb{T}$ in which additional restrictions are appended to validators. Thus, chunks in the liberal world might cease to be valid in the stricter universe.
(3) We could also restrict f further so that $f : \mathsf{Transaction}_{\mathbb{S}} \to \mathsf{Transaction}_{\mathbb{T}}$.
This would yield fewer arrows, and we prefer to allow the flexibility of mapping a single transaction in $\mathbb{S}$ to a chunk of transactions in $\mathbb{T}$; following an intuition that $\mathbb{S}$ is a coarse-grained representation which f maps into a finely-grained representation where something that was considered a single transaction is now a chunk.

3.8. Idealised UTxO

One special case of IEUTxO deserves its own discussion:

REMARK 3.8.1 (Idealised UTxO). Recall from Figure 1 that validators take as input a *pointed* transaction:

$$\mathsf{Transaction}_{!} \subseteq \mathit{fin}_{!}(\mathsf{Input}) \times \mathit{fin}(\mathsf{Output}).$$

Recall also from Notation 2.2.3(2) and that a *pointed transaction* is a transaction with one distinguished input of that transaction. For convenience we will call this the **input-point** of the transaction.

The UTxO model — on which Bitcoin is based — is the special case of EUTxO where validators just examine the input-point. So intuitively, in the UTxO model a validator of an output sees just the input that points to that output, in the sense of Notation 3.1.3, and it does not pay any attention to the transaction in which that input occurs.

We therefore obtain an **Idealised UTxO** (**IUTxO**) model from Figure 1 just by changing the line for validators to:

$$\mathsf{Validator} \subseteq pow(\beta \times \mathsf{Input}).$$

There is an easy embedding map which we can write 1, taking an IUTxO model to an IEUTxO model, derived from the embedding

$$pow(\beta \times \mathsf{Input}) \longrightarrow pow(\beta \times \mathsf{Transaction}_!)$$

which is itself derived from the projection taking a pointed transaction to its input-point:

$$\mathsf{Transaction}_! \longrightarrow \mathsf{Input}.$$

Then we can define a category of **IUTxO models** such that

— objects are IUTxO models, and
— arrows are functions exactly as defined in Definition 3.7.1.

Proposition 3.8.2. *The mapping* e *extends to a categorical embedding*[27] $\mathsf{IUTxO} \to \mathsf{IEUTxO}$.

Proof. Direct from the construction, since an IUTxO model is identified with an IEUTxO model whose validators ignore the transaction and just look at the input-point. □

Remark 3.8.3. For convenience, we may treat IUTxO as a direct subset of IEUTxO — abusing notation we could write $\mathsf{IUTxO} \subseteq \mathsf{IEUTxO}$ — thus identifying an IUTxO model with an IEUTxO model whose validators only check the input-point of their transaction. Thus for instance we wrote 'is identified with' in Proposition 3.8.2. It will always be clear what is intended and we could always unroll the injections if required.

4. ABSTRACT CHUNK SYSTEMS: ACS

4.1. Basic definitions

Remark 4.1.1. IEUTxO models are good, because they abstract key features of blockchain architectures in a simple and (I would argue) clear manner: output, input, and (valid) combination of transactions to form chunks and then blockchains.

However, IEUTxOs are concrete. An IEUTxO model is full of internal structure, by its very construction as a solution to type equations in Figure 1. We will now set about developing an axiomatic, algebraic account of the essential features that make IEUTxOs interesting.

We recall some basic definitions:

Definition 4.1.2. Suppose X is a set and $\leq \subseteq \mathsf{X}^2$ is a relation on X. Call $(\mathsf{X}, \leq)$ a **well-ordering** when:

(1) $\leq$ is a partial order (reflexive, transitive, anti-symmetric), and
(2) $\leq$ is well-founded (every descending chain is eventually stationary).[28]

As per Notation 3.1.5, X and $\leq$ are also assumed equivariant.

[27] A functor that is injective on objects and bijective on arrows.
[28] Alternative and equivalent definition: every *strictly* descending chain is finite.

EXAMPLE 4.1.3. This should be familiar, but we give examples:

— $(\mathbb{Z}, \leq)$ is not well-founded.
— $(pow(\mathbb{A}), \subseteq)$ and $(fin(\mathbb{A}), \subseteq)$ are well-founded.
— $[\mathbb{N}]$ (lists of numbers) with sublist inclusion is well-founded.

DEFINITION 4.1.4. Suppose $(\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq)$ is a partial order with an equivariant least element e and an equivariant greatest element f. Call $x \in \mathsf{X}$ **atomic** when

(1) $\mathsf{e} \lneq x \lneq \mathsf{f}$ and
(2) for every $x' \in \mathsf{X}$ if $x' \leq x$ then either $x' = \mathsf{e}$ or $x' = x$.

Write $atomic(\mathsf{X})$ for the set of atomic elements of X (see also Definition 4.2.5).

If we call $x \in \mathsf{X}$ **proper** when it is neither e nor f (following the standard terminology of *proper subset*), then an atomic element is "a minimal proper element".

REMARK 4.1.5. The set of atomic elements $atomic(\mathsf{X})$ is not to be confused with the set of atoms $\mathbb{A}$ from Definition 2.2.1. This name collision is just a coincidence.

Lemma 4.1.6 will be useful later:

LEMMA 4.1.6. *Suppose* $\mathbb{T} \in \mathsf{IEUTxO}$ *and consider* $\mathsf{Chunk}_{\mathbb{T}}$ *(valid lists of transactions; see Definition 3.4.1) as a partial order under sublist inclusion* $\leq$.

Then the atomic elements in $(\mathsf{Chunk}_{\mathbb{T}}, \leq)$ *are precisely the singleton chunks (Notation 3.4.3).*

Proof. Using Corollary 3.5.2. □

4.2. Monoid of chunks

DEFINITION 4.2.1. Assume we have equivariant data $(\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \bullet)$ where:

— X is a set.
— $\mathsf{e}, \mathsf{f} \in \mathsf{X}$ are called **unit** and **fail** respectively.
— $\leq \subseteq \mathsf{X}^2$ is a relation.
— $\bullet : \mathsf{X}^2 \to \mathsf{X}$ is a **composition**.

Call $(\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \bullet)$ a **monoid of chunks** when:

(1) $\mathsf{e} \bullet x = x = x \bullet \mathsf{e}$.
(2) $\mathsf{f} \bullet x = \mathsf{f} = x \bullet \mathsf{f}$.
(3) $\leq$ is a well-ordering for which the unit e is a bottom element and the paradoxical element f is a top element.
(4) Composition $\bullet$ is **associative**, and **monotone** in both components, meaning that
$$x' \leq x \quad \text{implies} \quad x' \bullet y \leq x \bullet y \quad \text{and}$$
$$y \leq y' \quad \text{implies} \quad x \bullet y \leq x \bullet y'.$$
(5) Composition is **increasing** in the sense that
$$x \leq x \bullet y \quad \text{and} \quad y \leq x \bullet y.$$
(6) If $x_1, \ldots, x_n \in \mathsf{X}$ and $x_1 \bullet \ldots \bullet x_n = \mathsf{f}$, then there must exist $1 \leq i < j \leq n$ such that $x_i \bullet x_j = \mathsf{f}$.

REMARK 4.2.2. A few comments on Definition 4.2.1:

(1) This is a clearly an abstraction of IEUTxO structure, where $\bullet$ is chunk composition and $\leq$ is list inclusion (proof in Proposition 5.1.3).
This is the key instance of the axioms that motivates the definition — IEUTxOs have more structure, but monoids of chunks is where we start. See also Example 4.2.6(6).
(2) $x \bullet y$ is not necessarily a least upper bound for $\{x, y\}$.
Take $X = \{1, 2\}$ and $x = [1]$ and $y = [2]$ in Example 4.2.6(4) (finite lists with a top element). Then $x \bullet y$ and $y \bullet x$ are distinct and incomparable, so both are upper bounds for $\{x, y\}$ but $x \bullet y \not\leq y \bullet x$ and $y \bullet x \not\leq x \bullet y$.

(3) We see that condition 6 of Definition 4.2.1 closely resembles Lemma 3.5.1, and indeed the condition is inspired by that very Lemma. We will use this in Proposition 5.1.3.

NOTATION 4.2.3. As is standard, we may write X for both a monoid of chunks and its carrier set. See for instance the first line of Definition 4.2.4.

DEFINITION 4.2.4. Suppose $\mathsf{X} = (\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \cdot)$ is a monoid of chunks.

(1) If $x \in \mathsf{X}$ and $[x_1, \ldots, x_n] \in [atomic(\mathsf{X})]$ is a finite list of atomic elements[29] in X and
$$x = x_1 \cdot \ldots \cdot x_n \quad \text{then say that} \quad x \textbf{ factorises as } [x_1, \ldots, x_n].$$
(2) Say that X is **generated by its atomic elements** when every $x \in \mathsf{X} \setminus \{\mathsf{f}\}$ has a (possibly non-unique) factorisation into atomic elements.

DEFINITION 4.2.5.

(1) Call a monoid of chunks $\mathsf{X} = (\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \cdot)$ **atomic** when:
   (a) X is generated as a monoid by its atomic elements (Definition 4.2.4).
   (b) There exists a **factorisation function** $factor : \mathsf{X} \setminus \{\mathsf{f}\} \to [atomic(\mathsf{X})]$ such that for every $x, y \in \mathsf{X} \setminus \{\mathsf{f}\}$
      i. $factor(x)$ factorises x (Definition 4.2.4) and
      ii. $factor(x \cdot y) = factor(x) \cdot factor(y)$ (the right-hand $\cdot$ denotes list concatentation; the left-hand $\cdot$ is the monoid action in X).

   In words we say that X is atomic when there is a homomorphism of partially-ordered monoids from $\mathsf{X} \setminus \{\mathsf{f}\}$ to the space of possible factorisations of its elements. The relevance of this condition is discussed in Remark 6.4.4.
(2) Call X **perfectly atomic** when it is atomic and furthermore:
   (a) factorisations into atom elements are unique and
   (b) if $x \leq y < \mathsf{f}$ and $x = x_1 \cdot \ldots \cdot x_m$ and $y = y_1 \cdot \ldots \cdot y_n$ then $[x_1, \ldots, x_m] \leq [y_1, \ldots, y_n]$ (sublist inclusion).

   The relevance of this condition is discussed in Proposition 7.3.6.

EXAMPLE 4.2.6. Suppose X is an equivariant set. Then:

(1) $pow(X)$ forms a monoid of chunks, where $\mathsf{e} = \varnothing$ and $\mathsf{f} = X$, and $\leq$ is subset inclusion, and composition $\cdot$ is sets union. It is atomic if and only if X is finite (recall: factorisations must be finite).
   We obtain a factorisation function by choosing any order on X, and listing elements of any $X' \subseteq X$ in order.
(2) $pow(X)$ forms a monoid of chunks, where:
   — $\mathsf{e} = \varnothing$ and $\mathsf{f} = X$.
   — $\leq$ is subset inclusion.
   — $x \cdot y = x \cup y$ if $x \cap y = \varnothing$, and $x \cdot y = \mathsf{f}$ otherwise.
   It is atomic if and only if X is finite.
(3) $fin(X) \cup \{X\}$ (finite sets of atoms, with a top element) forms an atomic monoid of chunks, using either of the two definitions above for $pow(X)$.
(4) Finite lists with a top element $[X]^\top$ — meaning finite lists of elements from X, plus one extra 'top' element $\top$ — form a perfectly atomic monoid of chunks as follows:
   — $\mathsf{e} = []$ and $\mathsf{f} = \top$.
   — $\leq$ is sublist inclusion (Notation 2.2.3(5)) and $l \leq \mathsf{f}$ for every finite list l.

[29] Functional programmers, who may be used to distinguishing between types (which are primitive) and sets (which inhabit powerset types), may perceive this definition as subtly broken, since it appears to apply a type-former $[\ldots]$, to a set $atomic(\mathsf{X})$. This is a culture clash and is not an issue with the maths as set up in this paper.

We are working in ZFA; the carrier set X is a set and so is $atomic(\mathsf{X})$ (and both are equivariant); the list set-former $[\ldots]$ is a *set*-former, not a type-former. Thus, $[atomic(\mathsf{X})]$ is well-defined by Notation 2.2.3(4) as the set of finite lists of atomic elements from X.

— Composition $\bullet$ is list concatenation on lists, and $x \bullet \mathsf{f} = \mathsf{f} = \mathsf{f} \bullet x$ for any x (list, or f).

(5) Finite lists with a top element $[X]^\top$ form an atomic (but not perfectly atomic) monoid of chunks as above, where $\leq$ and $\bullet$ are defined as follows:
— $l \leq l'$ holds when l is not a singleton list and l is a sublist of l'.
So $[] \leq [x]$ and $[] \leq [x,z] \leq [x,y,z]$ but $[x] \not\leq [x,y]$; and the proper atomic elements are singleton and two-element lists.
— $[] \bullet l = l \bullet [] = l$ for any list.
— $\mathsf{f} \bullet x = \mathsf{f} = x \bullet \mathsf{f}$ for any x.
— If l and l' are non-empty lists, then $l \bullet l'$ is l_{init} concatenated with l'_{tail}, where l_{init} is everything except for the last element of l, and l'_{tail} is everything except for the first element of l'.

(6) As touched on above, if $\mathbb{T}$ is an IEUTxO model then $\mathbb{T}$ gives rise to a perfectly atomic monoid of chunks. See Proposition 5.1.3.

Remark 4.2.7. It might seem counterintuitive to make failure f a *top* element in Definition 4.2.1, especially if we are used to seeing domain models where 'failure' is intuitively 'non-termination' and features $\bot$ as a bottom element.

We have a concrete reason for this: our canonical IEUTxO models are based on lists ordered by sublist inclusion, so bottom is already occupied by the empty list $[]$ which plays the role of e (see Definition 5.1.1).

But also we have abstract justifications: if we think of a chunk system as a many-valued logic (in which truth-values are chunks or blockchains and $\leq$ reflects how they accumulate transactions over time), then to exhibit a $\top$ is to *fail* to exhibit a concrete witness. Or (thinking perhaps of callCC [CFW86]) we can think of f as a 'final' or 'escape' element.

4.3. Behaviour, positions, and equivalence

4.3.1. Left- and right-behaviour

Definition 4.3.1. Suppose $\mathsf{X} = (\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \bullet)$ is a monoid of chunks. Then we have natural **left-** and **right-behaviour** functions:

$$\begin{array}{ll} \mathit{leftB} : \mathsf{X} \to \mathit{pow}(\mathsf{X}) & \mathit{rightB} : \mathsf{X} \to \mathit{pow}(\mathsf{X}) \\ \mathit{leftB} : x \mapsto \{y{\in}\mathsf{X} \mid y \bullet x < \mathsf{f}\} & \mathit{rightB} : x \mapsto \{y{\in}\mathsf{X} \mid x \bullet y < \mathsf{f}\} \end{array}$$

Lemma 4.3.2. *Suppose* X *is a monoid of chunks. Then we have:*

(1) $\mathit{leftB}(\mathsf{e}) = \mathit{rightB}(\mathsf{e}) = \mathsf{X} \setminus \{\mathsf{f}\}$.
(2) $\mathit{leftB}(\mathsf{f}) = \mathit{rightB}(\mathsf{f}) = \varnothing$.

Proof. A fact of Definition 4.2.1(1&2). □

Remark 4.3.3. If we think of f as a failure element, and we think of $x \bullet y$ as being a composition of which we can observe whether it fails or succeeds, then

— rightB maps $x \in \mathsf{X}$ to its right-observational behaviour, and
— leftB maps $x \in \mathsf{X}$ to its left-observational behaviour.

Remark 4.3.4. Parallels can be made in Definition 4.3.1 with the λ-calculus [Bar84] and the π-calculus [Mil99]:

(1) In the λ-calculus, a standard observable is non-termination.
Here we are doing something similar, except that (as noted in Remark 4.3.3) instead of failure to *terminate* ($\bot$) we observe failure to *compose* (f), and we consider combination both to the left and to the right.
Continuing the analogy, in the untyped λ-calculus, the left-behaviour set of a term t would be those s such that st terminates; and the right-behaviour set of t would be those s such that ts terminates.

(2) The π-calculus has notions of communications across channels, and as noted in Remark 3.4.4 we see a resemblance with communication of an input and output on a position. However there are differences, including:
   (a) *Validation* is not primitive in the π-calculus but it is a core precept here.
   (b) Communication in the plain π-calculus (without considering dialects) is non-deterministic — one channel name can be invoked by multiple inputs and outputs — whereas here a key assumption is that every channel name (i.e. position) must have one input and one output — and if not, the chunk collapses to a failure error-state f (cf. the conditions in Definitions 3.4.1 and 3.4.10).
   (c) Name-restriction in the π-calculus is not automatic but instead is managed by an explicit restriction term-former. In contrast here a communicating channel (an output-input pair) automatically closes when used once. We say 'closed' and not 'bound' because the name remains visible in *up* (see also *stx* in the IEUTxO models); it is just that no further communication may occur along it. We discuss garbage-collecting names in chunks in Subsection 8.2.

REMARK 4.3.5. Definitions 4.3.6 and 4.3.9 will build on Definitions 4.2.1 and 4.3.1 to derive a full notion of an *observable interface* of a monoid element, all derived just from the partiality of composition. We will make good use of this in the rest of the development, for instance Definition 6.2.1 depends on it.

4.3.2. Positions

DEFINITION 4.3.6. Suppose X is a monoid of chunks. Define $\mathit{posi}(x) \subseteq \mathbb{A}$ the **positions** of $x \in \mathsf{X}$ as follows:

$$\begin{array}{l}\mathit{posi}(\mathsf{f}_{\mathsf{X}}) = \varnothing\\ \mathit{posi}(x) = \{a \in \mathbb{A} \mid \forall \pi{\in}\mathit{fix}(a).\pi{\cdot}x \notin \mathit{leftB}(x) \cup \mathit{rightB}(x)\} \quad (x \in \mathsf{X} \setminus \{\mathsf{f}_{\mathsf{X}}\}).\end{array}$$

($\mathit{fix}(a)$ from Definition 2.3.4.)

Thus $a \in \mathit{posi}(x)$ when $x \neq \mathsf{f}_{\mathsf{X}}$ and $\pi{\cdot}x \notin \mathit{leftB}(x) \cup \mathit{rightB}(x)$, for any π such that $\pi(a) = a$.

REMARK 4.3.7. Note that the $\cdot$ in $\pi{\cdot}x$ in Definition 4.3.6 above refers to the atoms-permutation action from Definition 2.3.2, not to the partial monoid action $\bullet$ from Definition 3.5.3.

REMARK 4.3.8. In words, $\mathit{posi}(x)$ from Definition 4.3.6 is those atoms such that there is no permutation fixing a such that $\pi{\cdot}x$ can be successfully combined (left or right) with x.

What is the intuition here?

The name *posi* reminds us of *pos* from Definition 3.2.3, though the definitions are quite different. They are indeed related; in fact, they are equal in a sense made formal in Proposition 5.2.2 (see also Lemma 6.3.2(2)).

We do not have all the machinery in place yet, so it may be helpful to point forwards here and observe that conditions 3 and 4 of Definition 4.4.1 can be read as a way to make name-clash into an observable.

So intuitively, Definition 4.3.6 — once combined with the notion of an oriented monoid from Definition 4.4.1 — can use permutations to observe name-clash: it measures the live communication channels $a \in \mathbb{A}$ in an element x by forcing name-clashes between a-channels with π-renamed variants $\pi{\cdot}x$ for $\pi \in \mathit{fix}(a)$. More details will follow, and see Remark 5.2.3.

DEFINITION 4.3.9. Suppose $\mathsf{X} = (\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \bullet)$ is a monoid of chunks, and suppose $x \in \mathsf{X}$ and $a \in \mathbb{A}$.

(1) If

$$a \in \mathit{posi}(x) \quad \text{and} \quad \exists y{\in}\mathit{leftB}(x).a \in \mathit{posi}(y),$$

then say that a **points left** in x.

Write $\mathit{left}(x) \subseteq \mathbb{A}$ for the set of atoms that point left in x.

(2) If

$$a \in \mathit{posi}(x) \quad \text{and} \quad \exists y{\in}\mathit{rightB}(x).a \in \mathit{posi}(y),$$

then say that a **points right** in x.
Write $\mathit{right}(x) \subseteq \mathbb{A}$ for the set of atoms that point right in x.

(3) If a points neither left nor right in a and yet $a \in \mathit{posi}(x)$, so that

$$a \in \mathit{posi}(x) \quad \text{and} \quad \forall y \in \mathit{leftB}(x) \cup \mathit{rightB}(x).a \notin \mathit{posi}(y),$$

then say that a **points up** in x.
Write $\mathit{up}(x) \subseteq \mathbb{A}$ for the set of atoms that point up in x.

Lemma 4.3.10 expresses intuitively that atoms that point 'up' in a transaction cannot engage in successful (non-failing) combination; they are 'stuck interfaces':

LEMMA 4.3.10. *Suppose* X *is a monoid of chunks and* $x, y \in \mathsf{X}$ *and* $a \in \mathit{up}(x)$. *Then*

$$a \in \mathit{posi}(y) \quad \textit{implies} \quad x \bullet y = y \bullet x = \mathsf{f}.$$

Proof. Direct from Definition 4.3.9(3). □

REMARK 4.3.11. The reader who sees similarities between the *left*, *right*, and *up* of Definition 4.3.9, and the *utxi*, *utxo*, and *stx* of Definition 3.4.8 is right: see Propositions 5.2.2, Lemma 5.3.1, and Proposition 5.5.4.

A simple lemma will be helpful:

LEMMA 4.3.12. *Suppose* X *is a monoid of chunks and* $x \in \mathsf{X}$. *Then:*

$$\begin{aligned} \mathit{up}(x) &= \mathit{posi}(x) \setminus (\mathit{left}(x) \cup \mathit{right}(x)) \\ \varnothing &= \mathit{left}(x) \cap \mathit{up}(x) \\ \varnothing &= \mathit{right}(x) \cap \mathit{up}(x) \\ \mathit{posi}(x) &= \mathit{left}(x) \cup \mathit{right}(x) \cup \mathit{up}(x) \end{aligned}$$

Proof. This just rephrases clause 3 of Definition 4.3.9. □

REMARK 4.3.13. Lemmas 4.3.12 and 3.5.10(3) are similar but note that the status of the underlying datatypes is somewhat different:

— A chunk $ch \in \mathsf{Chunk}$ of an IEUTxO model is full of internal structure, and operations on it are defined in terms of that structure, whereas
— an element $x \in \mathsf{X}$ in a monoid of chunks is an abstract entity and we assume nothing about its internal structure.

Thus, a similarity between them has significance: it is a sanity check on our model and indicates that something rather abstract (monoids of chunks) is accurately following the behaviour of something more concrete (IEUTxO models).

REMARK 4.3.14. Lemma 4.3.12 expresses that every position in some $x \in \mathsf{X}$ (Definition 4.3.6) must point in a direction in $\{\mathit{left}, \mathit{right}, \mathit{up}\}$, and it cannot point both left and up, or both right and up.

Note that Definition 4.3.9 admits a possibility that an atom could point both left and right; this *cannot* happen in the IEUTxO models (see Lemma 3.5.10(1)). We will exclude this when we introduce the notion of an oriented monoid of chunks; see Corollary 4.4.5.

LEMMA 4.3.15.

(*1*) $\mathit{left}(\mathsf{e}) = \mathit{right}(\mathsf{e}) = \mathit{up}(\mathsf{e}) = \varnothing$.
(*2*) $\mathit{left}(\mathsf{f}) = \mathit{right}(\mathsf{f}) = \mathit{up}(\mathsf{f}) = \varnothing$.
(*3*) *As a corollary using Lemma 4.3.12,* $\mathit{posi}(\mathsf{e}) = \mathit{posi}(\mathsf{f}) = \varnothing$.[30]

Proof. We check the behaviour of e and f as specified in Definition 4.2.1 against the definitions of *left*, *right*, and *up* in Definition 4.3.9 and see that this is true. □

[30] $\mathit{posi}(\mathsf{f}) = \varnothing$ is also immediate from Definition 4.3.6.

4.3.3. Observational equivalence

Definition 4.3.16. Suppose X is a monoid of chunks.

(1) Call x and x' in X **observationally equivalent** and write
$$x \sim x' \quad \text{when} \quad \mathit{leftB}(x) = \mathit{leftB}(x') \wedge \mathit{rightB}(x) = \mathit{rightB}(x').$$
(2) Say that x and y **commute (up to observational equivalence)** when
$$x \cdot y \sim y \cdot x.$$

We start with a simple but useful sanity check:

Lemma 4.3.17. *Suppose X is a monoid of chunks and $x, y \in \mathsf{X}$. Then if x and y commute then*
$$x \cdot y < \mathsf{f} \iff y \cdot x < \mathsf{f} \quad \textit{and} \quad x \cdot y = \mathsf{f} \iff y \cdot x = \mathsf{f}.$$

Proof. We unpack Definitions 4.3.16(1&2) and 4.3.1 and conclude that
$$x \cdot y \cdot \mathsf{e} < \mathsf{f} \iff y \cdot x \cdot \mathsf{e} < \mathsf{f}.$$
The result follows, since e is the unit for $\cdot$. □

Lemma 4.3.18. *Suppose X is a monoid of chunks. Then if $x \sim x'$ (Definition 4.3.16) then*
$$\mathit{left}(x) = \mathit{left}(x') \quad \textit{and} \quad \mathit{right}(x) = \mathit{right}(x') \quad \textit{and} \quad \mathit{up}(x) = \mathit{up}(x').$$

Proof. A fact of Definitions 4.3.16(1) and 4.3.9. □

4.4. Oriented monoids

4.4.1. Definition and properties

Definition 4.4.1. Suppose $\mathsf{X} = (\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \cdot)$ is a monoid of chunks.
Call X **oriented** when for all $x, y \in \mathsf{X}$:

(1) $\mathit{posi}(x) \subseteq_{\mathit{fin}} \mathbb{A}$.
(2) If $\mathit{posi}(x) = \varnothing$ then $x \in \{\mathsf{e}, \mathsf{f}\}$.
(3) If $\mathit{left}(x) \cap \mathit{right}(y) \neq \varnothing$ then $x \cdot y = \mathsf{f}$.
(4) If $\mathit{posi}(x) \cap \mathit{posi}(y) = \varnothing$ then x and y commute up to observational equivalence (Definition 4.3.16(2)).
(5) If $\mathit{posi}(x) \cap \mathit{posi}(y) = \varnothing$ and $\mathsf{f} \notin \{x, y\}$ then $x \cdot y < \mathsf{f}$.

Remark 4.4.2. We discuss the conditions of Definition 4.4.1 in turn:

(1) An element $x \in \mathsf{X}$ can only be accessible on finitely many channel interfaces.
(2) The only elements without any interface (meaning atoms that point left right or up) are the unit element ($\leq$-bottom) and the failure element ($\leq$-top). Compare with the IEUTxO property Lemma 3.2.5.
We use this in Lemmas 4.4.6 and 6.3.1.
(3) Interfaces always try to connect, but can only *successfully* connect if the directions of their interfaces match up; if not, the whole combination fails.
We use this in Lemma 4.4.6, which is required for Proposition 4.4.8.
(4) This condition echoes Lemma 3.5.9(2). We use it in Proposition 4.4.8.
(5) Elements with no channels in common, cannot fail to compose.

We will show later that the IEUTxO models from Definition 3.1.9 are models of Definition 4.4.1 in a suitable sense; see Proposition 5.3.3.

We can strengthen Definition 4.4.1(2) to a logical equivalence:

Lemma 4.4.3. *Suppose X is an oriented monoid of chunks and $x \in \mathsf{X}$. Then*
$$\mathit{posi}(x) = \varnothing \quad \textit{if and only if} \quad x \in \{\mathsf{e}, \mathsf{f}\}.$$

Proof. The right-to-left implication is direct from Definition 4.4.1(2). The left-to-right implication is Lemma 4.3.15. □

Lemma 4.4.4 is a nice way to repackage Definition 4.4.1(3) in a slightly more accessible wrapper. In its form it resembles Lemma 3.5.10(2), and we use it for Corollary 4.4.5:

LEMMA 4.4.4. *Suppose* X *is an oriented monoid of chunks and suppose* $x, y \in \mathsf{X}$. *Then*

$$x \cdot y < \mathsf{f} \quad \textit{implies} \quad \textit{posi}(x) \cap \textit{posi}(y) \subseteq \textit{right}(x) \cap \textit{left}(y).$$

Proof. We consider the possibilities, using Lemma 4.3.12:

— *Suppose* $a \in \textit{left}(x) \cap \textit{posi}(y)$. From Definition 4.4.1(3) $a \notin \textit{right}(x)$, and by Lemma 4.3.12 $a \in \textit{posi}(x)$. It follows from Definition 4.3.9(2) that $x \cdot y = \mathsf{f}$.
— *Suppose* $a \in \textit{right}(y) \cap \textit{posi}(x)$. From Definition 4.4.1(3) $a \notin \textit{left}(y)$, and by Lemma 4.3.12 $a \in \textit{posi}(y)$. It follows from Definition 4.3.9(1) that $x \cdot y = \mathsf{f}$.
— *Other cases* are from Lemma 4.3.10 (or by direct reasoning from Definition 4.3.9(3)).

□

Corollary 4.4.5 is a slightly magical result, in the sense that it is perhaps not immediately obvious that it should follow from our definitions so far. In its form, if not its proof, it clearly resembles Lemma 3.5.10(1). We need it for Lemma 6.2.5, that atomic elements in X generate valid singleton chunks under a mapping to IEUTxO models F:

COROLLARY 4.4.5. *Suppose* X *is an oriented monoid of chunks and* $x \in \mathsf{X}$. *Then:*[31]

$$\textit{left}(x) \cap \textit{right}(x) = \varnothing.$$

Proof. Suppose $a \in \textit{left}(x)$; we will show that $a \in \textit{right}(x)$ is impossible. Consider some $y \in \mathsf{X}$ with $a \in \textit{posi}(y)$, so that by Lemma 4.3.12 $a \in \textit{left}(y) \cup \textit{right}(y) \cup \textit{up}(y)$. Then:

— If $a \in \textit{left}(y)$ then $a \in \textit{left}(x) \cap \textit{left}(y)$ and by Lemma 4.4.4 $x \cdot y = \mathsf{f}$.
— If $a \in \textit{right}(y)$ then $a \in \textit{left}(x) \cap \textit{right}(y)$ and by Lemma 4.4.4 $x \cdot y = \mathsf{f}$.
— If $a \in \textit{up}(y)$ then $a \in \textit{left}(x) \cap \textit{up}(y)$ and by Lemma 4.4.4 $x \cdot y = \mathsf{f}$.

Thus $a \in \textit{posi}(y)$ implies $x \cdot y = \mathsf{f}$ and so $y \notin \textit{rightB}(x)$. It follows from Definition 4.3.9(2) that $a \notin \textit{right}(x)$ as required. □

We use Lemma 4.4.6 for Proposition 4.4.8:

LEMMA 4.4.6. *Suppose* $\mathsf{X} = (\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \cdot)$ *is an oriented monoid of chunks. Then at least one of the following must hold:*

$$x \cdot y = \mathsf{f} \qquad y \cdot x = \mathsf{f} \qquad \textit{posi}(x) \cap \textit{posi}(y) = \varnothing$$

Proof. If x or y are equal to e or f then $\textit{posi}(x) \cap \textit{posi}(y) = \varnothing$ is immediate from Lemma 4.4.3.

So suppose $x, y \notin \{\mathsf{e}, \mathsf{f}\}$, from which it follows by Lemma 4.4.3 (or direct from Definition 4.4.1(2)) that $\textit{posi}(x) \neq \varnothing$ and $\textit{posi}(y) \neq \varnothing$. Suppose we have some $a \in \textit{posi}(x) \cap \textit{posi}(y)$. We reason by cases using our assumption that X is oriented (Definition 4.4.1):

— If $a \in \textit{left}(x) \cap \textit{right}(y)$ then $x \cdot y = \mathsf{f}$ by Definition 4.4.1(3).
— If $a \in \textit{right}(x) \cap \textit{left}(y)$ then $y \cdot x = \mathsf{f}$ by by Definition 4.4.1(3).
— If $a \in \textit{up}(x)$ or $a \in \textit{up}(y)$ then $x \cdot y = y \cdot x = \mathsf{f}$ by Lemma 4.3.10.
— Other cases are no harder.

□

[31] It would be nice to write this as $\textit{left}(x) \# \textit{right}(x)$ or $\textit{left}(x) \perp \textit{right}(x)$, but we prefer to trade off notational elegance for clarity and explicitness here, so we will write out our sets disjointness conditions in full.

Remark 4.4.7. Proposition 4.4.8 is a partial converse to Definition 4.4.1(4) (compare also with Proposition 3.5.12, which is the same result but for a different structure, and with a very different proof). It is significant because it equates

— a static property of positions, with
— a local property of being combinable in either order, with
— a global operational property of being commutative up to observation.

Commutativity is of particular interest in the context of blockchains, because they are by design intended to be distributed, so that we cannot in general know or assume in what order transactions get appended.

Proposition 4.4.8. *Suppose* $\mathsf{X} = (\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \bullet)$ *is an oriented monoid of chunks, and* $x, y \in \mathsf{X}$. *Suppose further that* $x \bullet y < \mathsf{f}$ *or* $y \bullet x < \mathsf{f}$ *(not necessarily both). Then the following conditions are equivalent:*

(1) $posi(x) \cap posi(y) = \varnothing$.
(2) $x \bullet y < \mathsf{f} \wedge y \bullet x < \mathsf{f}$.
(3) x *and* y *commute up to observational equivalence (Definition 4.3.16(2)).*

Proof. First, note that since $x \bullet y < \mathsf{f}$ or $y \bullet x < \mathsf{f}$, it must from Definition 4.2.1(2) be the case that $x < \mathsf{f}$ and $y < \mathsf{f}$.

If $x \bullet y < \mathsf{f} \wedge y \bullet x < \mathsf{f}$ then $posi(x) \cap posi(y) = \varnothing$ by Lemma 4.4.6. Conversely if $posi(x) \cap posi(y) = \varnothing$ then (since $x < \mathsf{f}$ and $y < \mathsf{f}$) by Definition 4.4.1(5) $x \bullet y < \mathsf{f}$ and $y \bullet x < \mathsf{f}$.

If $posi(x) \cap posi(y) = \varnothing$ then x and y commute by Definition 4.4.1(4). Conversely, if $posi(x) \cap posi(y) \neq \varnothing$ then by Lemma 4.4.6 (since $x \bullet y < \mathsf{f}$ or $y \bullet x < \mathsf{f}$) we have that $y \bullet x = \mathsf{f}$ or $x \bullet y = \mathsf{f}$ respectively, and in particular we have that $x \bullet y \neq y \bullet x$. Using Lemma 4.3.17 we conclude that x and y do not commute. □

Remark 4.4.9 (Comment on design). There is design freedom to Definition 4.4.1, and we mention this briefly for future work. One plausible condition is:

$$up(x \bullet y) \subseteq up(x) \cup up(y) \cup (right(x) \cap left(y)).$$

Intuitively, this states that combining x and y can *only* bind positions that point right in x and point left in y. An even stricter variant would be to insist on equality provided that $x \bullet y \neq \mathsf{f}$.

4.4.2. A brief discussion. We might ask:

Why bother with monoids of chunks? Why not just work with IEUTxOs?

The answer is that we need both: IEUTxOs are the motivating concrete model, and monoids of chunks are their abstraction.

Of course, the IEUTxO equations from Figure 1 themselves are an abstraction and generalisation of a concrete inductive definition in [CCM+20, Figure 3], so in overview this paper has the following hierarchy of models, given in increasing order of generality —

— The inductive EUTxO structures from [CCM+20].
— IEUTxO models from Definition 3.1.2.
— Abstract chunk systems (ACS) from Definition 4.5.1.

— though there is also more going on, because we also map from ACS back down to IEUTxO (Theorem 7.3.4 and surrounding discussion).

Programmers can think of APIs, which abstract away from internal structure of a concrete implementation; this makes programs more modular, and easier to test and document.

Modern programming languages make it easier to program efficiently using abstract denotations, instantiating only when needed — one might call this *just-in-time instantiation*. So even if we have

just one concrete model and want one implementation, a good algebraic theory is still relevant to producing working code, because

— it may help structure the mathematics and the code — the *algebraic graphs with class* Haskell package illustrates how effective this marriage of theory and practice can be [Mok17] — and
— an abstraction can be read as a library of testable properties, against which implementations can be checked. If an implementation fails an axiom — it's wrong.

So perhaps a better question is this:

> *What are the essential properties of* IEUTxO *that make it interesting? How are these properties layered, and nested; and how can they be compactly represented?*

Definitions 4.2.1 and 4.4.1 are one set of answers to these questions. And then, a further question is this:

> *What other models, aside from* IEUTxO *models, exist of the ACS axioms?*

This paper contains one answer (plus examples; see Subsection 4.6): in Definitions 6.2.1 and 6.4.1 and Proposition 6.4.3 we give a functor taking *any* oriented monoid of chunks to IEUTxOs. We also map functorially in the other direction in Definitions 5.1.1 and 5.4.1 and Theorem 5.4.4.

The overall results are packaged up in Theorem 7.3.4.

4.5. The category ACS of abstract chunk systems

We are now ready to give the algebraic account of IEUTxOs promised in Remark 4.1.1:[32]

Definition 4.5.1. An **abstract chunk system** (**ACS**) is an oriented atomic monoid of chunks (Definitions 4.4.1, 4.2.5, and 4.2.1).

Definition 4.5.2. Define ACS the **category of abstract chunk systems** by:

(1) Objects are abstract chunk systems (Definition 4.5.1).
(2) Arrows $g : \mathsf{X} \to \mathsf{Y}$ are sets functions from X to Y such that:
  (a) $g(\mathsf{e}_\mathsf{X}) = \mathsf{e}_\mathsf{Y}$ and $g(\mathsf{f}_\mathsf{X}) = \mathsf{f}_\mathsf{Y}$
  (b) $x \leq y < \mathsf{f}_\mathsf{X}$ implies $g(x) \leq g(y) < \mathsf{f}_\mathsf{Y}$
  (c) $g(x) \bullet g(y) = g(x \bullet y)$

Composition of arrows is composition of functions, and the identity arrow is the identity function.

Remark 4.5.3 (Comment on design).

(1) We do not insist in Definition 4.5.2 that $g(x)$ must be atomic if x is. This corresponds to our choice in Definition 3.7.1(2) to let f map from transactions to chunks, and not from transactions to transactions.
(2) We do insist in Definition 4.5.2(2b) that if x is not the failure element f_X in X then $g(x)$ is also not the failure element f_Y in Y. This corresponds to our choice in Definition 3.7.1(2) to make f a total function from transactions to chunks, rather than a partial one (cf. discussion in Remark 3.7.5(2)). We could relax this condition by allowing g to map $x < \mathsf{f}_\mathsf{X}$ to f_Y. There would be nothing wrong with this and it would just exhibit ACS as embedded in a larger category with the same objects but more arrows.

Lemma 4.5.4 just repackages Definition 4.2.5 for objects in ACS:

Lemma 4.5.4. *Suppose* $\mathsf{X}, \mathsf{Y} \in \mathsf{ACS}$ *and* $x \in \mathsf{X}$ *and* $g : \mathsf{X} \to \mathsf{Y} \in \mathsf{ACS}$. *Then:*

— *If* $x \neq \mathsf{f}$ *then there exists some finite (possibly empty, possibly non-unique) list of atomic elements* $x_1, \ldots, x_n \in \mathit{atomic}(\mathsf{X})$ *such that* $x = x_1 \bullet \ldots \bullet x_n$.

[32] It will take more work to *prove* that this is so. See the functors F and G which we define in Sections 5 and 6, and Theorem 7.3.4.

— *If* $x \neq \mathsf{f}$ *(and in particular by Definition 4.1.4(1) if* x *is atomic) then there exists some finite (possibly empty, possibly non-unique) list of atomic elements* $y_1, \ldots, y_n \in atomic(\mathsf{Y})$ *such that* $g(x) = y_1 \bullet \ldots \bullet y_n$.

Proof. Immediate since by Definition 4.5.1 X is atomic (Definition 4.2.5). □

Proposition 4.5.5.

(1) *An arrow* $g : \mathsf{X} \to \mathsf{Y} \in \mathsf{ACS}$ *(Definition 4.5.2(2)) is uniquely determined by its action on* $atomic(\mathsf{X})$.
(2) *As a corollary, if* $g, g' : \mathsf{X} \to \mathsf{Y}$ *are two arrows, then to check the equality of arrows* $g = g'$ *it suffices to check that* $g(x) = g'(x)$ *for every* $x \in atomic(\mathsf{X})$.

Proof. (1) Consider some $x \in \mathsf{X}$. If $x \in \{\mathsf{f}, \mathsf{e}\}$ then the action of f is determined by Definition 4.5.2(2a).
Otherwise, using Lemma 4.5.4 write $x = x_1 \bullet \cdots \bullet x_n$ for atomic $x_1, \ldots, x_n \in atomic(\mathsf{X})$. We then have from Definition 4.5.2(2c) that

$$f(x_1) \bullet \ldots \bullet g(x_n) = g(x).$$

Thus $g(x)$ is determined by the values of $g(x_1), \ldots, g(x_n)$.
(2) By a routine argument from Definition 4.2.5 (since X is atomic) and Definition 4.5.2(2c).
□

4.6. Examples of abstract chunk systems

We created abstract chunk systems in (Definition 4.5.1) to abstract away the internal structure of IEUTxO models (Definition 3.1.2).

This is a standard move in mathematics: axiomatise, then generalise. As we argued in Subsections 1.1 and 4.4.2, *even if* the reader only cares about practical hands-on implementation, for which internal structure is available because we have the code, it is still useful — and arguably *indispensable* — to take a moment to get a good higher-view of what it is that is implemented, since axioms provide a language for the higher-level enterprises of comparison, description, specification, correctness, testing, and exposition.

Now we take a moment to go back and consider what the space of concrete models of the ACS axioms looks like. We give a list of examples, which is not intended to be exhaustive but which we hope may illustrate the scope, character, and structure of our definition:

(1) Consider the set of all finite sets of atoms $A = \{a_1, \ldots, a_n\} \subseteq_{fin} \mathbb{A}$ and $\mathbb{A}$ itself:

$$\mathsf{X} = fin(\mathbb{A}) \cup \{\mathbb{A}\}.$$

X forms an ACS such that
— $A \leq B$ when $A \subseteq B$.
— $A \bullet B = A \uplus B$ (disjoint union) if A and B are disjoint, and $A \bullet B = \mathbb{A}$ otherwise.
Unpacking definitions, we can check that:
— Atomic elements are singletons $\{a\}$.
— A factorisation function $factor$ (Definition 4.2.5(1b)) is obtained by ordering atoms and listing the finite set in order.
— $posi(A) = up(A) = A$ and $left(A) = right(A) = \varnothing$.

(2) Consider some datatype Terms of term syntax over variable symbols $a, b, c, \ldots$ (terms of first-order logic $s, t ::= a \mid f(t, \ldots, t)$ for some term-formers f, or terms of the untyped λ-calculus $s, t ::= a \mid ss \mid \lambda a.s$ would suffice).
Let **finite substitutions** be finite partial maps from variable symbols to terms generated by **atomic substitutions**

$$\sigma = [a{:=}t].$$

Composition is defined in a standard way by acting on terms as $s\sigma$.

If $a \notin dom(\sigma)$ then $a\sigma = a$ (so the substitution acts as the identity on variable symbols not in its domain).
Now, add a *fail* top element f, such that $s\mathsf{f} = s$ always (so f is a formal element that acts as the identity). If the reader likes, we could take f to be $[a{:=}a \mid \text{ all } a]$.
Then substitutions with f form an ACS, where
— $\sigma \leq \sigma'$ when $dom(\sigma) \subseteq dom(\sigma')$, and $\sigma \leq \mathsf{f}$ always.
— If $dom(\sigma) \cap dom(\sigma') = \varnothing$ then $\sigma \cdot \sigma'$ is the finite substitution that maps a to $a\sigma\sigma'$ for each $a \in dom(\sigma) \cup dom(\sigma')$, and
— if $dom(\sigma) \cap dom(\sigma') \neq \varnothing$ then $\sigma \cdot \sigma' = \mathsf{f}$
The 'disjoint domains or fail' condition ensures that composition is monotone in $\leq$. Unpacking definitions, we check that:
— Atomic elements are the generators $[a{:=}s]$.
— A factorisation function $factor$ (Definition 4.2.5(1b)) is obtained by ordering atoms and listing a substitution as a list of atomic substitutions in order of the atom on the left.[33]
— $posi(\sigma) = left(\sigma) = dom(\sigma)$ and $right(\sigma) = up(\sigma) = \varnothing$.

5. THE FUNCTOR $F : \mathsf{IEUTXO} \to \mathsf{ACS}$

5.1. Action on objects

Definition 5.1.1. Suppose $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator}) \in \mathsf{IEUTxO}$ is a model (Definition 3.1.9) and recall $\mathsf{Chunk}_{\mathbb{T}}$ from Notation 3.4.3. Then we define an abstract chunk system

$$F(\mathbb{T}) = (\mathsf{Chunk}_{\mathbb{T}} \cup \{fail\}, [], fail, \leq, \cdot) \in \mathsf{ACS}$$

as follows:

— The underlying set is $\mathsf{Chunk}_{\mathbb{T}} \cup \{fail\}$ — so if we write "$x \in F(\mathbb{T})$" this means "either $x = fail$ or $x \in \mathsf{Chunk}_{\mathbb{T}}$".
— $\mathsf{e} = []$ and $\mathsf{f} = fail$.
— $\leq$ is sublist inclusion (Notation 2.2.3(5)) on chunks that are lists of transactions, with f as a top element, so $ch \leq \mathsf{f}$ always.
— $x \cdot \mathsf{f} = \mathsf{f} = \mathsf{f} \cdot x$, and if x and y are both chunks then $\cdot$ is validated concatentation of chunks, defaulting to the failure element f if validation fails.

Remark 5.1.2. So if $x, y \in \mathsf{Chunk}_{\mathbb{T}}$ but $x \cdot y \in [\mathsf{Transaction}_{\mathbb{T}}] \setminus \mathsf{Chunk}_{\mathbb{T}}$ in $\mathbb{T}$ — so the two chunks cannot be validly combined in $\mathbb{T}$ — then $x \cdot y = fail$ in $F(\mathbb{T})$. We will always be clear whether $\cdot$ means 'concatenate as lists' or 'compose in $F(\mathbb{T})$'.

$F(\mathbb{T})$ resembles $\mathbb{T}$, except with an explicit failure element $fail$ added. This makes monoid composition total: not every pair of chunks composes to form a list of transactions that is a chunk — as per the criteria in Definition 3.4.1 — thus any combination of chunks that is not a chunk, can be set to $fail$.

We still need to prove that $F(\mathbb{T})$ is an abstract chunk system (Definition 4.5.1) in the category ACS (Definition 4.5.2(1)) thus:

— $F(\mathbb{T})$ is a monoid of chunks (Definition 4.2.1),
— $F(\mathbb{T})$ is atomic (Definition 4.2.5) — in fact it is perfectly atomic (Definition 4.2.5(2)) — and
— $F(\mathbb{T})$ is oriented (Definition 4.4.1).

We do this next, culminating with Theorem 5.3.4.

Note that this gives us a dual view of a chunk:

— as a chunk in the IEUTxO universe, and
— as an abstract element in the ACS universe.

[33] This can be a little subtle: if b is ordered before a then $[a{:=}b] \cdot [b{:=}c]$ would need to be written as $[b{:=}c] \cdot [a{:=}c]$.

To what extent do these two views correspond, and how can we make this formal? We prove Proposition 5.2.2, which verifies that the positions in a chunk as an element in the IEUTxO universe, coincide with its positions as an element in the ACS universe. This is refined in further results; notably Lemma 5.3.1 and its sharper corollary Proposition 5.5.4.

PROPOSITION 5.1.3.

(1) *$F(\mathbb{T})$ from Definition 5.1.1 is a perfectly atomic (Definition 4.2.5(2)) monoid of chunks (Definition 4.2.1).*
Unpacking Definition 4.2.5(2), any $x \in F(\mathbb{T}) \setminus \{\mathsf{f}\}$ can be uniquely decomposed as a (possibly empty) finite list of atomic elements
$$x = x_1 \bullet \ldots \bullet x_n,$$
and if $x \leq y < \mathsf{f}_{F(\mathbb{T})}$ then the factorisation of x is a sublist of the factorisation of y.
(2) *Furthermore, the atomic elements $x_i \in atomic(F(\mathbb{T}))$ are singleton lists of the form $[tx]$ for $tx \in \mathsf{Transaction}_{\mathbb{T}}$.*

Proof. Most of the properties in Definition 4.2.1 are facts of sublist inclusion and concatenation. Condition 6 of Definition 4.2.1 is Lemma 3.5.1.

Recall that being a chunk is down-closed by Corollary 3.5.2, so in a perfectly atomic monoid of chunks, every chunk is above its atomic chunks. It is just a fact of lists, sublist inclusion, and the construction in Definition 5.1.1 that $F(\mathbb{T})$ is perfectly atomic with atomic elements singleton chunks of the form $[tx]$ for $tx \in \mathsf{Transaction}_{\mathbb{T}}$. □

5.2. Relation between the partial monoid $\mathsf{Chunk}_{\mathbb{T}}$ and the monoid of chunks $F(\mathbb{T})$

LEMMA 5.2.1. *Suppose that:*

— $T \in \mathsf{IEUTxO}$ and $ch \in \mathsf{Chunk}_{\mathbb{T}}$ and $a \in pos(ch)$.
— $\pi \in Perm$ is a permutation and $\pi(a) = a$, and write $ch' = \pi \cdot ch$ (Definition 2.3.3).

It is a structural fact that $ch \bullet ch', ch' \bullet ch \in [\mathsf{Transaction}_{\mathbb{T}}]$ since a concatentation of two lists is a list. However:
$$ch' \bullet ch \notin \mathsf{Chunk}_{\mathbb{T}} \quad \textit{and} \quad ch \bullet ch' \notin \mathsf{Chunk}_{\mathbb{T}}.$$

Proof. By assumption $a \in pos(ch)$, so using Lemmas 3.5.10(3) and 3.5.14 (since $\pi(a) = a$) precisely one of
$$\begin{array}{rl} & a \in utxi(ch) \cap utxi(ch'), \\ & a \in utxo(ch) \cap utxo(ch'), \\ \text{or} & a \in stx(ch) \cap stx(ch') \end{array}$$
must hold. In each of these three cases the result follows from Lemma 3.5.10(2). □

PROPOSITION 5.2.2. *Suppose $\mathbb{T}$ is an IEUTxO model and $x \in \mathsf{Chunk}_{\mathbb{T}}$, and*

— recall $pos(x)$ from Definition 3.2.3, and
— noting from Proposition 5.1.3 that x can also be viewed as an element $x \in F(\mathbb{T})$, recall also $posi(x)$ from Definition 4.3.6.

Then
$$pos(x) = posi(x).$$

Proof. By a routine calculation from Definitions 3.4.1 and 4.3.6 using Lemma 5.2.1. □

We continue Remark 4.3.8:

REMARK 5.2.3. *pos* from Definition 3.2.3 and *posi* from Definition 4.3.6 have different constructions yet give equal results, in a sense made formal in Proposition 5.2.2. Aside from being a useful equality, what does this tell us?

pos is *intensional* — it has and requires full access to the internal structure of its argument — whereas $posi$ is *extensional* — it treats its argument as a black box in which it can only permute atoms and observe compositional behaviour. See also a similar observation in Remark 4.3.13.

A significance of Proposition 5.2.2 is as a (non-trivial) correctness assertion about the overall algebraic framework in which these definitions have been embedded: that the abstract interface of x viewed extensionally, matches the concrete interface of x when viewed intensionally.

Put another way, Proposition 5.2.2 has the flavour of being a weak but indicative soundness and completeness result relating a class of concrete models with a class of abstract ones.

Lemma 5.3.1 refines this, and we will continue to build on these ideas, culminating with Theorem 7.3.4.

5.3. $F(\mathbb{T})$ is oriented, so $F(\mathbb{T}) \in \mathsf{ACS}$

LEMMA 5.3.1. *Suppose* $\mathbb{T} \in \mathsf{IEUTxO}$. *By Proposition 5.1.3* $F(\mathbb{T}) = \mathsf{Chunk}_{\mathbb{T}} \cup \{fail\}$ *is a monoid of chunks, so it has notions of left, right, and up from Definition 4.3.9.*

Then for $x \in \mathsf{Chunk}_{\mathbb{T}}$ *we have:*[34]

$$\begin{aligned} left(x) &\subseteq utxi(x) \\ right(x) &\subseteq utxo(x) \\ stx(x) &\subseteq up(x) \end{aligned}$$

Proof. We consider each line in turn:

(1) If $a \in left(x)$ then by Lemma 4.3.12 $a \in posi(x)$ so by Proposition 5.2.2 also $a \in pos(x)$. By Definition 4.3.9 there exists $y \in \mathsf{Chunk}_{\mathbb{T}}$ such that $a \in posi(y)$, so by Proposition 5.2.2 also $a \in pos(y)$, and $y \bullet x \in \mathsf{Chunk}_{\mathbb{T}}$. It follows from Lemma 3.5.10(2) that ($a \in utxo(y)$ and) $a \in utxi(x)$ as required.
(2) If $a \in right(x)$ then $a \in pos(x)$ and by Definition 4.3.9 there exists $y \in \mathsf{Chunk}_{\mathbb{T}}$ such that $a \in pos(y)$ and $x \bullet y \in \mathsf{Chunk}_{\mathbb{T}}$, so by Lemma 3.5.10(2) ($a \in utxi(y)$ and) $a \in utxo(x)$ as required.
(3) The reasoning to prove $stx(x) \subseteq up(x)$ is no harder.

□

REMARK 5.3.2. Lemma 5.3.1 is interesting as much for what it is *not*, namely it is not the equality $left = utxi$ and $right = utxi$ and $up = stx$ that one might initially expect. Why?

(1) Consider a chunk $ch \in \mathsf{Chunk}_{\mathbb{T}}$ with an IEUTxO output located at a but with an empty validator (one which validates no inputs). Then $a \in utxo(ch)$ in $\mathbb{T}$, but $a \in up(ch)$ in $F(\mathbb{T})$.
(2) Similarly consider a chunk ch with an input located at a but such that no validator will validate it — just because an input exists, does not mean a validator must exist to accept it. Then $a \in utxi(ch)$ in $\mathbb{T}$, but $a \in up(ch)$ in $F(\mathbb{T})$.

We return to this with the notion of a *blocked channel*, in Subsection 5.5.

PROPOSITION 5.3.3. $F(\mathbb{T})$ *from Definition 5.1.1 is oriented (Definition 4.4.1).*

Proof. We check each condition of Definition 4.4.1 in turn:

(1) *We check that* $posi(x) \subseteq_{fin} \mathbb{A}$.
It is a structural fact of Definition 3.2.3 that $pos(x) \subseteq_{fin} \mathbb{A}$. We use Proposition 5.2.2.
(2) *We check that* $posi(x) = \varnothing$ *implies* $x = \mathsf{e}_{F(\mathbb{T})}$ *or* $x = \mathsf{f}_{F(\mathbb{T})}$.
An element $x \in F(\mathbb{T})$ is either a chunk or the failure element:
— If x is a chunk and $posi(x) = \varnothing$ then by Proposition 5.2.2 $posi(x) = \varnothing$ and by Lemma 3.2.5 $x = []$, which by Definition 5.1.1 is $\mathsf{e}_{F(\mathbb{T})}$.

[34] . . . so we are looking at a 'real chunk' here, for which $utxi$, $utxo$, and stx are defined, and excluding our extra failure element, for which they are not defined . . .

— If $x = \mathit{fail}$ then there is nothing to prove, since $\mathit{fail} = \mathsf{f}_{F(\mathbb{T})}$.

(3) *We check that* $\mathit{left}(x) \cap \mathit{right}(y) \neq \varnothing$ *implies* $x \cdot y = \mathsf{f}$.
If x or y are fail then $x \cdot y = \mathsf{f}$.
So suppose that x and y are chunks, that is, suppose $x, y \in \mathsf{Chunk}_{\mathbb{T}}$. By Lemma 5.3.1 $\mathit{utxi}(x) \cap \mathit{utxo}(y) \neq \varnothing$. We use Lemma 3.5.10(2).

(4) *We check that if* $\mathit{posi}(x) \cap \mathit{posi}(y) = \varnothing$ *then* x *and* y *commute (Definition 4.3.16(2)).*
From Proposition 5.2.2 and Lemma 3.5.9(2).

(5) *We check that if* $\mathit{posi}(x) \cap \mathit{posi}(y) = \varnothing$ *and* $\mathsf{f} \notin \{x, y\}$ *then* $x \cdot y < \mathsf{f}_{F(\mathbb{T})}$.
Using Proposition 5.2.2, this just rephrases Lemma 3.5.6 in the language of a monoid of chunks.

□

THEOREM 5.3.4. *$F(\mathbb{T})$ (Definition 5.1.1) is an abstract chunk system in* ACS *(Definition 4.5.1). In symbols:*

$$F(\mathbb{T}) \in \mathsf{ACS}.$$

Proof. From Propositions 5.1.3 and 5.3.3. □

5.4. Action of F on arrows

DEFINITION 5.4.1. Suppose $f : \mathbb{S} \to \mathbb{T} \in \mathsf{IEUTxO}$ is an arrow (Definition 3.7.1(2)). We define an arrow

$$F(f) : F(\mathbb{S}) \to F(\mathbb{T}) \in \mathsf{ACS}$$

by

$$\begin{array}{ll} F(f)([tx_1, \ldots, tx_n]) & = f(tx_1) \cdot \ldots \cdot f(tx_n) \\ F(f)(\mathit{fail}_{\mathbb{S}}) & = \mathit{fail}_{\mathbb{T}}. \end{array} \tag{3}$$

LEMMA 5.4.2. *$F(f)$ from Definition 5.4.1 does indeed map from $F(\mathbb{S})$ to $F(\mathbb{T})$.*

Proof. We need to check that validity is preserved, meaning that if $[tx_1, \ldots, tx_n]$ is a chunk then so is $f(tx_1) \cdot \ldots \cdot f(tx_n)$. This is Lemma 3.7.2. □

PROPOSITION 5.4.3. *Continuing Definition 5.4.1, we have that*

$$f : \mathbb{S} \to \mathbb{T} \in \mathsf{IEUTxO} \quad \textit{implies} \quad F(f) : F(\mathbb{S}) \to F(\mathbb{T}) \in \mathsf{IEUTxO}.$$

Furthermore, $F(f'\, f) = F(f')\, F(f)$ and $F(id_{\mathbb{S}}) = id_{F(\mathbb{S})}$.

Proof. We check the properties in Definition 4.5.2(2) in turn:

(1) *We check that* $F(f)(\mathsf{e}_{F(\mathbb{S})}) = \mathsf{e}_{F(\mathbb{T})}$ *and* $F(f)(\mathsf{f}_{F(\mathbb{S})}) = \mathsf{f}_{F(\mathbb{T})}$.
This is just the fact that $F(f)([]) = []$ and $F(f)(\mathit{fail}_{\mathbb{S}}) = \mathit{fail}_{\mathbb{T}}$.

(2) *We check that* $x \leq y < \mathit{fail}_{F(\mathbb{S})}$ *implies* $F(f)(x) \leq F(f)(y) < \mathsf{f}_{F(\mathbb{T})}$.
It is a fact of the construction in Definition 5.4.1 that $x \leq y$ (x is a sublist of y) implies $F(f)(x) \leq F(f)(y)$.

(3) *We check that* $F(f)(x) \cdot F(f)(y) = F(f)(x \cdot y)$.
A fact of the first clause of equation (3) in Definition 5.4.1.

We can check $F(f'\, f) = F(f')\, F(f)$ and $F(id_{\mathbb{S}}) = id_{F(\mathbb{S})}$ by routine calculations which we elide. □

THEOREM 5.4.4. *The map F, with the action on IEUTxO models from Definition 5.1.1, and with the action on arrows from Definition 5.4.1, is a functor*

$$F : \mathsf{IEUTxO} \to \mathsf{ACS}.$$

Proof. This is Theorem 5.3.4 and Proposition 5.4.3. □

5.5. Blocked channels

Recall from Remark 3.4.4 that we can think of positions as communication channels in the π-calculus sense. We conclude this Section by taking a little time to refine the subset inclusions from Lemma 5.3.1. For this, we need to consider the possibility of a channel which is (intuitively) *blocked*, in the sense that no successful validation can occur across it:

DEFINITION 5.5.1. Suppose that $\mathbb{T}$ is an IEUTxO model and $ch \in \mathsf{Chunk}_{\mathbb{T}}$ and $a \in \mathbb{A}$.

(1) Suppose that
— $a \in utxi(ch)$ and
— for every $ch' \in \mathsf{Chunk}_{\mathbb{T}}$ with $a \in utxo(ch')$, $ch' \bullet ch$ is not a chunk.
Then call a a **blocked utxi** in ch. Write $blockedUtxi(ch)$ for the blocked utxis of ch.
(2) Similarly define $blockedUtxo(ch)$ the **blocked utxos** of ch to be those $a \in \mathbb{A}$ such that
— $a \in utxo(ch)$ and
— for every $ch' \in \mathsf{Chunk}_{\mathbb{T}}$ with $a \in utxi(ch')$, $ch' \bullet ch$ is not a chunk.

REMARK 5.5.2. So a blocked UTxI or UTxO in a chunk is an input or output that exists, but which fails if you try to interact with it. This could happen for an output whose validator is the empty set (it fails on any input), or for an input such that no validator in $\mathbb{T}$ exists to validate it (see Remark 5.3.2).

LEMMA 5.5.3. *Suppose $\mathbb{T}$ is an IEUTxO model and $x, y \in \mathsf{Chunk}_{\mathbb{T}}$ and $x \bullet y \in \mathsf{Chunk}_{\mathbb{T}}$. Then*

$$utxo(x) \cap utxi(y) \subseteq right(x) \cap left(y).$$

Proof. Suppose $a \in utxo(x) \cap utxi(y)$. In particular then by Lemma 3.5.10(3) $a \in pos(x) \cap pos(y)$ so by Proposition 5.2.2 also $a \in posi(x) \cap posi(y)$.

$F(\mathbb{T})$ is a monoid of chunks by Proposition 5.1.3, and since $x \bullet y \in \mathsf{Chunk}_{\mathbb{T}}$ it follows that $x \bullet y < \mathsf{f}_{F(\mathbb{T})}$. It follows from Definition 4.3.1 that $y \in rightB(x)$ and $x \in leftB(y)$.

The result now follows by Definition 4.3.9. □

PROPOSITION 5.5.4. *Suppose $\mathbb{T} \in \mathsf{IEUTxO}$ and $x \in F(\mathbb{T}) \backslash \{\mathsf{f}\}$ (that is, $x \in \mathsf{Chunk}_{\mathbb{T}}$). Then:*

$$\begin{aligned} left(x) &= utxi(x) \setminus blockedUtxi(x) \\ right(x) &= utxo(x) \setminus blockedUtxo(x) \\ up(x) &= stx(x) \cup blockedUtxi(x) \cup blockedUtxo(x) \end{aligned}$$

Proof. We know by Lemma 5.3.1 that $left(x) \subseteq utxi(x)$ and $right(x) \subseteq utxo(x)$. Now suppose $a \in utxi(x)$ and $a \notin blockedUtxi(x)$; unpacking Definition 5.5.1 this means that there exists a $y \in \mathsf{Chunk}_{\mathbb{T}}$ such that $a \in utxo(y)$ and $y \bullet x \in \mathsf{Chunk}_{\mathbb{T}}$. By Lemma 5.5.3 it follows that ($a \in right(y)$ and) $a \in left(x)$.

The case of $right(x)$ is similar, and the case of $up(x)$ follows from the previous two cases and Lemma 5.3.1. □

6. THE FUNCTOR $G : \mathsf{ACS} \to \mathsf{IEUTXO}$

6.1. A brief discussion: why represent?

In Subsection 4.4.2 we observed a hierarchy of models, from concrete EUTxO structures to IEUTxO models to abstract chunk systems.

The mapping from IEUxO to ACS is the functor $F : \mathsf{IEUTxO} \to \mathsf{ACS}$ from Section 5. We will now exhibit a functor $G : \mathsf{ACS} \to \mathsf{IEUTxO}$ *back down* from the abstract to the concrete structures.[35]

This is interesting for two reasons: one specific, and one general. We consider each in turn.

G is interesting because:

[35] Note that G consists of an action on objects, and an action on arrows, and we can usefully have the former without the latter. See Remark 6.4.4.

(1) It gives a sense in which the abstraction reasonably represents the concrete models. That is, there is nothing the abstract model could do that is so crazy that it cannot be engineered back down to a concrete structure. This may involve some ugly concrete fiddling, emulation, and choices — as one might expect going from an abstract to a concrete object — but it can be done, and seeing how, can be helpful for understanding both worlds.
(2) Sometimes, theorems are better proved in the concrete world than the abstract world. This can be particularly useful to prove negative properties, that something *cannot* happen in the abstract world, because it would correspond to something that would be impossible in the concrete world. A well-known example is that every Boolean Algebra can be represented as a powerset, and thus every finite Boolean Algebra has cardinality a power of two. Thus, to prove that some abstract structure does *not* admit any Boolean Algebra structure, it suffices if its carrier set is finite and has cardinality that is *not* a power of two.[36]

Now to understand the relevance of G specifically for this paper, consider the following question:

In what sense is Definition 4.5.1 a good abstraction of Definition 3.1.2?

Design decisions are embedded in the conditions of Definition 4.5.1, and some of these were not trivial and had more than one plausible outcome. Why did we choose as we did? How do these choices interact? In what sense were they appropriate?

To answer these questions, F is not necessarily the greatest help on its own. To illustrate why, consider that we can obtain a general 'theory of blockchains' merely by insisting that an 'abstract chunk system' is a set. We impose no further structure: *et voilà*: instant generality![37] But this tells us little; e.g. F would just be the forgetful functor, mapping an IEUTxO model to its underlying set. We could map just to monoids, if we add the failure element, and again an F would exist, but this would be only slightly less uninformative.

So where is the sweet spot, and why? As we observed, merely exhibiting an F-style functor does not help: we need to get an algebraic measure of what it is about Figure 1 that gives it its essential nature.

We get a formally meaningful measure of an appropriate level of abstraction by locating one at which we can build a sensible functor G *going back*, and seeing how conditions in Definition 4.5.1 interact with its construction — and, we can observe how tweaking them can affect, or even break, these constructions. A discussion of such tweaks, and their effects, is in Remarks 6.4.4 and 7.1.5, and Proposition 7.3.6.

Remark 6.1.1 (Comment on design). What counts as a 'sensible' G is a design decision in itself. We consider several options in this paper (listed here in increasing order of size of the category of denotations ACS):

(1) a categorical equivalence (Proposition 7.3.6); or
(2) a categorical embedding (Theorem 7.3.4 and Remark 7.3.5); or
(3) just an injection on objects (Remark 6.4.4).

All these possibilities are justifiable.[38] So to be clear: G and the choices we make in building it are not intended as direct value judgements; they are a way to measure and explore the structure of a large, abstract, and interesting design space.

[36] We do not exhibit any such application of our result in this paper; we are just making the general observation. Still, it is possible that in future work our constructions might be put to such use.
[37] This really happened. An author lifted an algebra from one of my papers, deleted crucial structure, and claimed superior generality. When the paper went to me to referee, I observed that deleting this structure also deleted all the interesting theorems. This was not necessarily fatal; but what other theorems or properties were there to replace them? No reply was forthcoming.
[38] A comparison: when giving a denotation to $\mathbb{N}$, the domain of denotations could be ω (equivalence), ordinals (embedding), or just an arbitrary infinite set (injection on objects). All three possibilities are reasonable, depending on the context.

6.2. Action on objects

Recall from Definition 3.1.2 the notion of an IEUTxO model, and the accompanying discussion in Remark 3.1.8 about the status of the injection $\nu : \mathsf{Validator} \hookrightarrow pow(\beta \times \mathsf{Transaction}_!)$.

Continuing that Remark, in Definition 6.2.1 we must be explicit about ν:

DEFINITION 6.2.1. Suppose $(\mathsf{X}, \mathsf{e}, \mathsf{f}, \leq, \bullet) \in \mathsf{ACS}$. We define an IEUTxO model $G(\mathsf{X})$

$$G(\mathsf{X}) = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator}, \nu : \mathsf{Validator} \hookrightarrow pow(\beta \times \mathsf{Transaction}_!))$$

as follows:

(1) We take $\alpha = \beta = atomic(\mathsf{X})$ (Definition 4.1.4).
(2) We take:

$$\mathsf{Validator} = \{*\}$$

where $\{*\}$ is a unit type.
(3) For each atomic $x \in atomic(\mathsf{X})$ we admit a transaction

$$\mathsf{tx}(x) \in \mathsf{Transaction}$$

such that:

$$\begin{aligned} input(\mathsf{tx}(x)) &= \{(a, x) \mid a \in left(x)\} \\ output(\mathsf{tx}(x)) &= \{(b, x, *) \mid b \in right(x) \cup up(x)\}. \end{aligned}$$

Thus:

$$\mathsf{Transaction} = \{\mathsf{tx}(X) \mid x \in atomic(\mathsf{X})\}$$

for tx defined as above.
(4) We define $\nu : \mathsf{Validator} \hookrightarrow pow(\beta \times \mathsf{Transaction}_!)$ to map $* \in \mathsf{Validator}$ as follows ($@i$ from Notation 3.1.3):

$$\nu(*) = \{(x, tx@(p, y)) \mid (p, y) \in input(tx),\ x \bullet y < \mathsf{f}\}.$$

Thus, $\nu(*)$ is the function that inputs x and a pointed transaction $tx@(p, y)$, extracts the data y from the input, and then checks that $x \bullet y < \mathsf{f}$ in X.[39]

REMARK 6.2.2. Continuing Remark 3.8.1, we see that the validator used by G in Definition 6.2.1 is UTxO-style; it only examines the (pointed) input of the transaction to be validated. So in fact, G maps not just to IEUTxO models but to the *IUTxO* models noted in Subsection 3.8. We will use this observation in Theorem 7.3.4(3).

An easy sanity check:

LEMMA 6.2.3. *Suppose* $\mathsf{X} \in \mathsf{ACS}$ *and* $x, y \in \mathsf{X}$. *Then* $\mathsf{tx}(x) \bullet \mathsf{tx}(y)$ *is a chunk if and only if* $x \bullet y < \mathsf{f}$.

Proof. By construction, unravelling Definition 6.2.1. □

REMARK 6.2.4 (Comment on design). In Definition 6.2.1(3) we set

— $input(\mathsf{tx}(x)) = \{(a, x) \mid a \in left(x)\}$ and
— $output(\mathsf{tx}(x)) = \{(b, x, *) \mid b \in right(x) \cup up(x)\}$.

So up-atoms in x map to $output$-atoms in $\mathsf{tx}(x)$. Why? For two reasons:

— *The short reason* is that it makes Lemma 6.3.2 work: all atoms in $posi(x)$ get recorded in an input or output (even the ones in $up(x)$, which cannot participate in a non-failing interaction), along with a copy of x (to get injectivity).

[39] In fact, the only pointed transactions possible in this system are $\mathsf{tx}(y)@(p, y)$, so we could also extract y from tx.

— *The longer reason* is as follows:
An ACS element $x \in \mathsf{X}$ has no internal structure and thus no explicit structural notion of inputs or outputs. Our only interaction with x is by combining it with other elements and observing partiality (cf. Remark 4.3.5).
But suppose our ACS X was obtained concretely from an IEUTxO model using F from Definition 5.1.1, so that x is 'secretly' a singleton chunk, presented as an atomic element in an ACS. Then $p \in up(x)$ could occur for two reasons:
— either p is the position of an input which no output will accept (perhaps it is labelled with some data that all validators disapprove of);
— or p is the position of an output that will not validate any available input (e.g. it has the empty validator).
When we come to map x back to a transaction $\mathsf{tx}(x)$, the simplest way to record p is to attach it to an IEUTxO output located at p, with a validator that always fails.
The other option would be to attach p to an *input* in $\mathsf{tx}(x)$, to tag the data carried by that input with some special 'fail-me' tag, and remember to create only validators that recognise this tag and reject the input. But this is clearly a more complicated way of doing things, and the design adopted in Definition 6.2.1(3) seems the natural and simple approach.

Several things about Definition 6.2.1 need checked. We start with Lemma 6.2.5:

Lemma 6.2.5.

(1) *If* $\mathsf{X} \in \mathsf{ACS}$ *and* $x \in atomic(\mathsf{X})$ *then* $\mathsf{tx}(x)$ *has the right type to be a transaction as per Figure 1.*
(2) *If* $\mathsf{X} \in \mathsf{ACS}$ *and* $x \in atomic(\mathsf{X})$ *then* $[\mathsf{tx}(x)]$ *is a chunk.*
(3) *As a corollary, atomic elements in* $F(\mathsf{X})$ *are precisely the singleton chunks of* $\mathsf{tx}(x)$*, where* x *ranges over atomic elements of* X *— or more concisely in symbols:*
$$atomic(F(\mathsf{X})) = \{[\mathsf{tx}(x)] \mid x \in atomic(\mathsf{X})\}.$$

Proof.

(1) From Definitions 4.4.1(1) and 6.2.1(3) and Lemma 4.3.12, $\mathsf{tx}(x)$ has finitely many inputs and outputs; so as per Figure 1 it is indeed a pair of a *finite* set of inputs and a *finite* set of outputs.
(2) By Lemma 3.4.6, to show $[\mathsf{tx}]$ is a valid chunk it would suffice to show that $input(\mathsf{tx}(x)) \cap output(\mathsf{tx}(x)) = \varnothing$. This follows from Lemma 4.3.12 and Corollary 4.4.5.
(3) By construction and Lemma 4.1.6, noting that in lists ordered by subset inclusion, atomic elements are singleton lists.

□

Remark 6.2.6 (Comment on design). We mention an alternative definition of G from Definition 6.2.1, just to illustrate that more than one encoding is possible:

(1) We take $\alpha = \beta = \mathsf{Validator} = atomic(\mathsf{X})$ (Definition 4.1.4).
(2) For each atomic $x \in atomic(\mathsf{X})$ we admit a transaction $\mathsf{tx}(x) \in \mathsf{Transaction}$ such that:
$$input(\mathsf{tx}(x)) = \{(a, x) \mid a \in left(x)\}$$
$$output(\mathsf{tx}(x)) = \{(b, x, x) \mid b \in right(x) \cup up(x)\}$$
(3) We define $\nu : \mathsf{Validator} \hookrightarrow pow(\beta \times \mathsf{Transaction}_!)$ to map $x \in \mathsf{Validator}$ as follows:
$$\nu(x) = \{(x, \mathsf{tx}(y)@i) \mid x \bullet y < \mathsf{f},\ i \in input(\mathsf{tx}(y))\}.$$

Remark 6.2.7. It remains to prove that ν is well-defined and (as required by Definition 3.1.2(5)) is injective, and that $G(\mathsf{X})$ is indeed an IEUTxO model. See Corollaries 6.3.3 and 6.3.4.

6.3. ν is injective

Lemma 6.3.1. *Suppose* $\mathsf{X} \in \mathsf{ACS}$ *(Definition 4.5.2). Then*
$$x \in atomic(\mathsf{X}) \quad \textit{implies} \quad posi(x) \neq \varnothing.$$

Proof. Suppose x is atomic. By Definition 4.1.4(1) $x \notin \{\mathsf{e}, \mathsf{f}\}$. We use Lemma 4.4.3. □

LEMMA 6.3.2. *Suppose* $\mathsf{X} \in \mathsf{ACS}$. *Then:*

(1) *The assignment*

$$x \in \mathit{atomic}(\mathsf{X}) \longmapsto \mathsf{tx}(x) \in \mathsf{Transaction}_{G(\mathsf{X})}$$

from Definition 6.2.1(3) is injective.
(2) $\mathit{posi}(x) = \mathit{pos}(\mathsf{tx}(x))$.

Proof. (1) By Lemma 6.3.1 (since x is atomic) $\mathit{posi}(x) \neq \varnothing$, so by Lemma 4.3.12 at least one of $\mathit{left}(x)$ or $\mathit{right}(x)$ or $\mathit{up}(x)$ must be nonempty.
If $\mathit{left}(x)$ is nonempty then $\mathsf{tx}(x)$ has an input and we can read x off the data in that input.
Otherwise $\mathsf{tx}(x)$ has an output and we can read x off the data in that output.
(2) It follows from Definition 6.2.1(3) and Figure 4 that $\mathit{pos}(\mathsf{tx}(x)) = \mathit{left}(x) \cup (\mathit{right}(x) \cup \mathit{up}(x))$.
Also, by Lemma 4.3.12 $\mathit{pos}(x) = \mathit{left}(x) \cup \mathit{right}(x) \cup \mathit{up}(x)$.
□

As promised in Remark 6.2.7, we prove:

COROLLARY 6.3.3. *The map*

$$\nu : \mathsf{Validator} \hookrightarrow \mathit{pow}(\beta \times \mathsf{Transaction}_!)$$

from Definition 6.2.1(4) is well-defined and injective.

Proof. By Lemma 6.3.2(1) we can deduce y from $\mathsf{tx}(y)$. The result follows. □

COROLLARY 6.3.4. *If* X *is an abstract chunk system (Definition 4.5.1) then* $G(\mathsf{X})$ *is an IEUTxO model (Definition 3.1.2).*

Proof. We just need to check the conditions of Definition 3.1.2; the only nontrivial part is that ν is an injection, and that is Corollary 6.3.3. □

6.4. Action on arrows

DEFINITION 6.4.1. Suppose $g : \mathsf{X} \to \mathsf{Y} \in \mathsf{ACS}$ and recall from Definition 3.7.1(2) that an arrow

$$G(g) = G(\mathsf{X}) \to G(\mathsf{Y}) \in \mathsf{IEUTxO}$$

should be a mapping from $\mathsf{Transaction}_{G(\mathsf{X})}$ to $\mathsf{Chunk}_{G(\mathsf{Y})}$. Recall *factor* from Definition 4.2.5(1b), which factorises non-failure elements into atomic constituents, and recall from Definition 4.5.2(2b) that g maps non-failure elements to non-failure elements.

Then define $G(g)$ by

$$G(g) : \mathsf{tx}(x) \longmapsto \mathsf{tx}(y_1) \bullet \ldots \bullet \mathsf{tx}(y_n) \in \mathsf{Chunk}_{G(\mathsf{Y})}$$
$$\text{where} \quad \mathit{factor}(g(x)) = [y_1, \ldots, y_n] \in [\mathit{atomic}(\mathsf{Y})].$$

LEMMA 6.4.2. $G(g)$ *from Definition 6.4.1 is well-defined.*

Proof. We must check that $\mathsf{tx}(y_1) \bullet \ldots \bullet \mathsf{tx}(y_n)$ is a chunk; this follows by Lemma 3.7.2. □

PROPOSITION 6.4.3. *The map* G, *with the action on abstract chunk systems from Definition 6.2.1, and with the action on arrows from Definition 6.4.1, is a functor*

$$G : \mathsf{ACS} \to \mathsf{IEUTxO}.$$

Proof. Given the results above, the only remaining thing to check is that if $g : \mathsf{X} \to \mathsf{Y}$ and $g' : \mathsf{Y} \to \mathsf{Z}$ then $G(g'\, g) = G(g')G(g)$. This follows by a routine argument from the definitions, using Definition 4.2.5(1(b)ii). □

REMARK 6.4.4 (Comment on design). The significance of condition 1b of Definition 4.2.5 is not that elements can be factored into atomic elements — this follows already from condition 1a — but that a factorisation can be *selected*, as a monoid homomorphism.

We use this condition in just one place: to define the action of G on arrows in Definition 6.4.1 and prove it functorial in Proposition 6.4.3.

It would be legitimate to remove condition 1b of Definition 4.2.5. This would exhibit our category ACS as a subcategory of a larger category which would include more objects — 'even more abstract' abstract chunk systems — at a cost of no longer being able to functorially map this larger space back down to IEUTxO.

Intuitively, this larger space behaves more like a space of all possible denotations rather than a space of IEUTxO-representable ones, which might be worthwhile if it admits other interesting examples; whether or not this will be the case cannot be predicted at time of writing.

Note that the action on *objects* from Definition 6.2.1 would still be well-defined even without Definition 4.2.5(1b), so that we can still represent our 'even more abstract' abstract chunk systems concretely in IEUTxO models: this just would not correspond to a functor.[40]

7. AN ADJUNCTION BETWEEN $F : \mathsf{IEUTXO} \to \mathsf{ACS}$ AND $G : \mathsf{ACS} \to \mathsf{IEUTXO}$

7.1. The counit map $\epsilon_{\mathsf{X}} : FG(\mathsf{X}) \to \mathsf{X}$ exists and is a surjection

REMARK 7.1.1. Suppose $\mathsf{X} \in \mathsf{ACS}$; we wish to define an arrow $\epsilon_{\mathsf{X}} : FG(\mathsf{X}) \to \mathsf{X} \in \mathsf{ACS}$. Unpacking Definitions 6.2.1 and 5.1.1, we see that an $x \in FG(\mathsf{X})$ has one of the following forms:

— $x = \mathit{fail}_{FG(\mathsf{X})}$ for $\mathit{fail}_{FG(\mathsf{X})}$ the *failure* element added by F to $\mathsf{Chunk}_{G(\mathsf{X})}$ in Definition 5.1.1.
— $x = [\mathsf{tx}(x_1), \ldots, \mathsf{tx}(x_n)]$ for some unique $[x_1, \ldots, x_n] \in [\mathit{atomic}(\mathsf{X})]$.

We also know from Lemma 6.3.2 that $\mathsf{tx} : \mathit{atomic}(\mathsf{X}) \to \mathsf{Transaction}(G(\mathsf{X}))$ is injective, and it follows that we can recover each x_i from the unique corresponding $[\mathsf{tx}(x_i)]$ above.

DEFINITION 7.1.2. Let $\epsilon_{\mathsf{X}} : FG(\mathsf{X}) \to \mathsf{X}$ be determined by:

(1) $\epsilon_{\mathsf{X}}([]) = \mathsf{e}_{\mathsf{X}}$ (this would be a special case of the next clause, for $n = 0$)
(2) $\epsilon_{\mathsf{X}}([\mathsf{tx}(x_1), \ldots, \mathsf{tx}(x_n)]) = x_1 \cdot \ldots \cdot x_n$ for $n \geq 1$ and $x_1, \ldots, x_n \in \mathit{atomic}(\mathsf{X})$
(3) $\epsilon_{\mathsf{X}}(\mathsf{f}_{FG(\mathsf{X})}) = \mathsf{f}_{\mathsf{X}}$

LEMMA 7.1.3. *Definition 7.1.2 is well-defined and determines an arrow in* ACS.

Proof. Well-definedness follows as per Remark 7.1.1 from Lemma 6.3.2, since we can recover each x_i from its $[\mathsf{tx}(x_i)]$. It remains to check the arrow conditions from Definition 4.5.2(2):

(1) *We check that* $\epsilon_{\mathsf{X}}(\mathsf{e}_{FG(\mathsf{X})}) = \mathsf{e}_{\mathsf{X}}$ *and* $\epsilon(\mathsf{f}_{FG(\mathsf{X})}) = \mathsf{f}_{\mathsf{X}}$.
A fact of Definition 7.1.2.
(2) *We check that* $x \leq y < \mathsf{f}_{FG(\mathsf{X})}$ *implies* $\epsilon_{\mathsf{X}}(x) \leq \epsilon_{\mathsf{X}}(y) < \mathsf{f}_{\mathsf{X}}$.
By Proposition 5.1.3 $FG(\mathsf{X})$ is perfectly atomic, and if $x \leq y < \mathsf{f}_{FG(\mathsf{X})}$ then x and y factorise uniquely into a composition of lists of singleton chunks, which by construction in Definition 6.2.1(3) have the form $\mathsf{tx}(x_i)$ and $\mathsf{tx}(y_j)$, such that the factorisation of x is a sublist of the factorisation of y.
The result follows by a routine calculation from Definition 7.1.2.
(3) $\epsilon_{\mathsf{X}}(x) \cdot \epsilon_{\mathsf{X}}(y) = \epsilon_{\mathsf{X}}(x \cdot y)$
A fact of Definition 7.1.2, again using the fact that by Proposition 5.1.3 $FG(\mathsf{X})$ is perfectly atomic.

□

[40] Thus, the G in the loop of embeddings in Remark 7.3.5 would weaken a mapping on objects.

PROPOSITION 7.1.4. *The counit map* $\epsilon_{\mathsf{X}} : FG(\mathsf{X}) \to \mathsf{X} \in \mathsf{ACS}$ *is a surjection on underlying sets.*

Proof. Suppose we are given $x \in \mathsf{X}$; we want to exhibit an element in $FG(\mathsf{X})$ that maps to it under ϵ_{X}.

If $x = \mathsf{f}_{\mathsf{X}}$ then by construction in Definition 5.1.1 $fail_{FG(\mathsf{X})} = \mathsf{f}_{FG(\mathsf{X})}$ and also by construction $\epsilon_{\mathsf{X}}(fail_{FG(\mathsf{X})}) = \mathsf{f}_{\mathsf{X}}$ so we are done.

Otherwise by Proposition 5.1.3 (or just by Definition 4.2.5) we can write

$$x = x_1 \bullet \ldots \bullet x_n$$

for some atomic $x_1, \ldots, x_n \in atomic(\mathsf{X})$, and looking at Definition 7.1.2 we immediately have that

$$\epsilon_{\mathsf{X}}([\mathsf{tx}(x_1), \ldots, \mathsf{tx}(x_n)]) = x_1 \bullet \ldots \bullet x_n = x.$$

□

REMARK 7.1.5 (Comment on design). By Proposition 7.1.4 ϵ_{X} surjects $FG(\mathsf{X})$ onto X as sets, but is it not necessarily surjective on the $\leq$ structure, meaning that $\epsilon_{\mathsf{X}}(ch) \leq_{\mathsf{X}} \epsilon_{\mathsf{X}}(ch')$ does not imply $ch \leq_{FG(\mathsf{X})} ch'$.

We can have this if we add condition 2b of Definition 4.2.5, that: if $x \leq y < \mathsf{f}$ then $factor(x) \leq factor(y)$ (the right-hand $\leq$ denotes sublist inclusion; the left-hand $\leq$ is the partial order on X). More on this in Proposition 7.3.6. Further tweaks to the design of abstract chunk systems are also mentioned in Remark 6.4.4.

7.2. The unit map $\eta_{\mathbb{T}} : \mathbb{T} \to GF(\mathbb{T})$ exists and is an isomorphism

REMARK 7.2.1. Suppose $\mathbb{T} = (\alpha, \beta, \mathsf{Transaction}, \mathsf{Validator}) \in \mathsf{IEUTxO}$. We wish to define an arrow

$$\eta_{\mathbb{T}} : \mathbb{T} \to GF(\mathbb{T}).$$

We can make some observations:

— By construction in Definition 5.1.1, $F(\mathbb{T})$ is isomorphic as a partial ordering to $\mathbb{T}$, with the addition of the *fail* top element.
  As noted in Proposition 5.1.3 it follows that the atomic elements of $F(\mathbb{T})$ correspond precisely with the singleton chunks in $\mathsf{Chunk}_{\mathbb{T}}$, and thus with transactions in $\mathsf{Transaction}_{\mathbb{T}}$.
  The chunks in $F(\mathbb{T})$ are then determined by combining the singleton chunks, subject to the well-formedness conditions of Definition 3.4.1 and the locality properties noted in Lemma 3.5.1.
— By construction in Definition 6.2.1 the atomic elements of $GF(\mathbb{T})$ are isomorphic to $atomic(F(\mathbb{T}))$, and by Lemma 6.2.5(3) also to $\mathsf{Transaction}_{\mathbb{T}}$.

DEFINITION 7.2.2. Let $\eta_{\mathbb{T}} : \mathbb{T} \to GF(\mathbb{T})$ be determined by mapping $tx \in \mathsf{Transaction}_{\mathbb{T}}$ to $[\mathsf{tx}(tx)] \in \mathsf{Chunk}_{GF(\mathbb{T})}$. Thus using Lemma 3.7.2 we have:

$$\eta_{\mathbb{T}}([tx_1, \ldots, tx_n]) = [\mathsf{tx}(tx_1), \ldots, \mathsf{tx}(tx_n)].$$

LEMMA 7.2.3. *If*

— $x = [tx_1, \ldots, tx_n] \in \mathsf{Chunk}_{\mathbb{T}}$, *then*
— $\eta_{\mathbb{T}}(x) = [\mathsf{tx}(tx_1), \ldots, \mathsf{tx}(tx_n)] \in \mathsf{Chunk}_{GF(\mathbb{T})}$.

As a corollary, Definition 7.2.2 does indeed map chunks to chunks.

Proof. By a routine check on Definition 6.2.1(3), using Proposition 5.5.4. □

LEMMA 7.2.4. *Recall the notions of utxi, utxo, and stx from Definition 3.4.8 and the notion of blockedUtxi from Definition 5.5.1. Recall η from Definition 7.2.2 and suppose $x \in \mathsf{X} \in \mathsf{ACS}$. Then:*

(1) $utxi(\eta_{\mathbb{T}}(x)) = utxi(x) \setminus blockedUtxi(x)$
(2) $utxo(\eta_{\mathbb{T}}(x)) = utxo(x) \cup blockedUtxi(x)$
(3) $stx(\eta_{\mathbb{T}}(x)) = stx(x)$

Proof. By routine calculations using Proposition 5.5.4. □

PROPOSITION 7.2.5. *The unit map* $\eta_{\mathbb{T}} : \mathsf{Chunk}_{\mathbb{T}} \to \mathsf{Chunk}_{GF(\mathbb{T})} \in \mathsf{IEUTxO}$ *is a bijection on underlying sets.*

Proof. Using Lemma 6.3.2 and the fact that from Definitions 5.1.1 and 6.2.1, any element of $GF(\mathbb{T})$ has the form $[\mathsf{tx}(tx_1), \ldots, \mathsf{tx}(tx_n)]$ for some transactions $tx_1, \ldots, tx_n \in \mathsf{Transaction}_{\mathbb{T}}$. □

7.3. F is left adjoint to G

REMARK 7.3.1. Naturality of ϵ and η is Propositions 7.3.2 and 7.3.3. The proofs are by diagram-chasing. We do need to be a little careful because of the choice made in the *factor* function (Definition 4.2.5(1b)), which propagates to the action of G on arrows (Definition 6.4.1). In the event, the diagram-chasing is all standard and it works fine.[41]

PROPOSITION 7.3.2. *The counit map* ϵ *is a natural transformation from* FG *to* 1_{ACS}.

Proof. Consider some arrow $g : \mathsf{X} \to \mathsf{Y} \in \mathsf{ACS}$. We must check a commuting square in ACS that

$$\epsilon_{\mathsf{Y}}\, FG(g) = g\, \epsilon_{FG(\mathsf{X})}.$$

Using Proposition 4.5.5(2) it suffices to check for each $x' \in \mathit{atomic}(FG(\mathsf{X}))$ that

$$\epsilon_{\mathsf{Y}}(FG(g)\,(x')) = g(\epsilon_{FG(\mathsf{X})}\, x').$$

From Lemma 6.2.5(3), $x' = [\mathsf{tx}(x)]$ for $x \in \mathit{atomic}(\mathsf{X})$.

Write $g(x) = y_1 \bullet \ldots \bullet y_n$ where $\mathit{factor}(g(x)) = [y_1, \ldots, y_n] \in [\mathit{atomic}(\mathsf{Y})]$ (Definition 4.2.5(1b)), so that

$$(FG(g))([\mathsf{tx}(x)]) = [\mathsf{tx}(y_1), \ldots, \mathsf{tx}(y_n)] \in FG(\mathsf{Y}).$$

Then

$$\epsilon_{\mathsf{Y}}(FG(g)([\mathsf{tx}(x)])) = \epsilon_{\mathsf{Y}}([\mathsf{tx}(y_1), \ldots, \mathsf{tx}(y_n)]) = y_1 \bullet \ldots \bullet y_n = y$$

and

$$g(\epsilon_{\mathsf{X}}([\mathsf{tx}(x)])) = g(x) = y$$

as required. □

Consider $f : \mathbb{S} \to \mathbb{T}$ and $F(f) : F(\mathbb{S}) \to F(\mathbb{T})$ and some $[tx] \in \mathsf{Chunk}_{\mathbb{S}}$ and $f(tx) = [tx_1, \ldots, tx_n] \in \mathsf{Chunk}_{\mathbb{T}}$. Then $\eta_{\mathbb{S}}([tx_1, \ldots, tx_n]) = [\mathsf{tx}(tx_1), \ldots, \mathsf{tx}(tx_n)]$ and

$$F(f)([tx])\,([tx_1, \ldots, tx_n])$$

PROPOSITION 7.3.3. *The unit map* η *is a natural transformation from* 1_{IEUTxO} *to* GF.

Proof. Consider some arrow $f : \mathbb{S} \to \mathbb{T} \in \mathsf{IEUTxO}$. We must check a commuting square in IEUTxO that

$$\eta_{\mathbb{T}}\, f = GF(f)\, \eta_{\mathbb{S}}.$$

Using Lemma 3.7.2 it suffices to check for each $tx \in \mathsf{Transaction}_{\mathbb{S}}$ that

$$\eta_{\mathbb{T}}(f([tx])) = GF(f)(\eta_{\mathbb{S}}([tx])).$$

[41] Let's pause on this 'it works fine'. This paper is populated by structures whose definitions can be quite different, and yet which mesh together: consider Proposition 5.2.2 and Remark 5.2.3, and the adjunction here.

If aspects of the proof of the equivalence follow without fuss, then this tells us that the different elements mesh correctly and with minimal friction. It might even have taken much thought by a certain author, and patient finessing of widely-separated definitions and proofs, for us to enjoy this happy state of affairs. Thus: the property of this argument that it is *fairly straightforward*, may in and of itself have some *mathematical* significance.

Suppose $f(tx) = [tx_1, \ldots, tx_n] \in \mathsf{Chunk}_{\mathbb{T}}$. Then

$$\eta_{\mathbb{T}}(f([tx])) = \eta_{\mathbb{T}}([tx_1, \ldots, tx_n]) = [\mathsf{tx}(tx_1), \ldots, \mathsf{tx}(tx_n)]$$

and

$$GF(f)(\eta_{\mathbb{S}}([tx])) = GF(f)([\mathsf{tx}(tx)]) = [\mathsf{tx}(tx_1), \ldots, \mathsf{tx}(tx_n)]$$

as required. □

Theorem 7.3.4.

(*1*) *The functors*

$$F : \mathsf{IEUTxO} \to \mathsf{ACS} \quad \textit{and} \quad G : \mathsf{ACS} \to \mathsf{IEUTxO}$$

from Definitions 5.1.1 and 5.4.1 (for F) and from Definitions 6.2.1 and 6.4.1 (for G) form an adjoint pair. In symbols:

$$F \dashv G : \mathsf{IEUTxO} \to \mathsf{ACS}$$

(*2*) *F is full and faithful (a bijection on homsets), and G is full (a surjection on homsets). Thus F is a* full embedding *of* IEUTxO *into* ACS.[42]

(*3*) *The image $G(\mathsf{X})$ of $\mathsf{X} \in \mathsf{ACS}$ is in fact a pure IUTxO model — meaning that its validators examine only the input-point of the transaction passed to them, not the entire transaction — and* G *maps more specifically to the full subcategory* IUTxO *of* IEUTxO *(Subsection 3.8). Thus have the following short chain of maps*

$$\mathsf{IUTxO} \overset{e \dashv GF}{\longrightarrow} \mathsf{IEUTxO} \overset{F \dashv G}{\longrightarrow} \mathsf{ACS} \tag{4}$$

where on the left e denotes the trivial inclusion/embedding map (Remark 3.8.1) with right adjoint GF.[43]

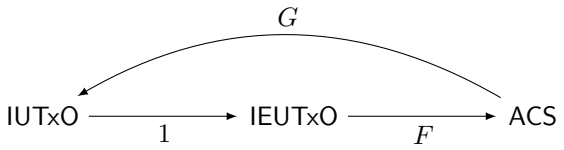
(*4*) *The image $GF(\mathbb{T})$ of $\mathbb{T} \in \mathsf{IEUTxO}$ is a pure IUTxO model and as a corollary, every IEUTxO model $\mathbb{T}$ is isomorphic via $\eta_{\mathbb{T}} : \mathbb{T} \to GF(\mathbb{T})$ to an IUTxO model $GF(\mathbb{T})$,*

Proof.

(1) The natural transformations are $\epsilon : FG \to 1$ and $\eta : 1 \to GF$ from Definitions 7.1.2 and 7.2.2. They are natural by Propositions 7.3.2 and 7.3.3.
(2) By Proposition 7.2.5 the unit η is a bijection and it follows that f is full and faithful. By Proposition 7.1.4 the counit ϵ is a surjection and it follows that G is full (this also follows from the fact that η is a bijection and so an injection).
(3) This is a structural fact of Definition 6.2.1, as observed in Remark 6.2.2.
(4) From Proposition 7.2.5 and part 3 of this result.

□

Remark 7.3.5. We can present the diagram in (4) in Theorem 7.3.4(3) (recall that 1 and IUTxO are from Subsection 3.8) quite nicely as a loop of embeddings:



[42] There seem to be various definitions in the literature of what an 'embedding of categories' should mean. By *full embedding* here, we mean a functor that is injective on objects, and bijective on arrows.
[43] Does G map to IEUTxO, or to IUTxO? Both, because we embedded the latter in the former: IUTxO $\subseteq$ IEUTxO. See Remark 3.8.3.

Intuitively, ACS denotations seem to be the largest denotational class which conveniently maps back down into IEUTxO models as above (this is an intuitive observation, not a theorem). But if we wish to optimise differently and make our denotation more specific, then we can be rewarded with a tighter result:

PROPOSITION 7.3.6. *If we strengthen Definition 4.5.1 so that an abstract chunk system is a* perfectly atomic *oriented monoid of chunks (Definition 4.2.5(2)) instead of just being an atomic one, then*

— ϵ_{X} in Proposition 7.1.4 becomes a bijection, and
— the embedding $F \dashv G$ in Theorem 7.3.4 becomes an equivalence of categories, and
— the loop illustrated in Remark 7.3.5 becomes a loop of equivalences.

Thus, the full subcategory in ACS *of* perfectly atomic *abstract chunk systems, and all arrows between them, is the 'properly IEUTxO-like' abstract chunk systems.*[44]

Proof. The bijection is just a structural fact: if by Definition 4.2.5(2a) an element $x \in \mathsf{X} \in \mathsf{ACS}$ factorises uniquely into a list of atomic elements $x_1 \bullet \cdots \bullet x_n$, then x can be *identified* with that list and G just maps it to a list of transactions $[\mathsf{tx}(x_1), \ldots, \mathsf{tx}(x_n)]$ — which, by design in Definition 6.2.1, has the same composition behaviour in $G(\mathsf{X})$ as x does in X.

Then Definition 4.2.5(2b) is exactly what is required for $FG(\mathsf{X})$ to 'remember' all the $\leq$-structure of X. □

8. CONCLUSIONS

We have presented the EUTxO blockchain model in a novel and compact form and derived from it an algebra-style theory of blockchains as partially-ordered partial monoids with channel name communication. This builds on previous work [BG20].

We hope this paper will make two contributions:

(1) its specific definitions and theorems, but also,
(2) an *idea* that blockchain structures can be subjected to this kind of analysis.

Or to put it another way: we illustrate that an algebraic theory of blockchains is possible, and what it might look like.

The reader can apply these ideas to their favourite blockchain architecture, and if this were widespread practice then this might help make the field accessible to an even broader audience, ease technical comparisons between systems, add clarity to a fast-changing field — and as we have argued, it might suggest structures and tests for practical programs, as is already reflected in a recent work [BG20; Gab20b].

We now reflect on the design decisions made along the way and suggest possibilities for future work.

8.1. Observational equivalence

We touched on notions of observational equivalence in Subsections 3.5.2 and 4.3.3.

The theory of EUTxO observational equivalence is a little weaker than we might like, in the sense that not as many things get identified as one might first anticipate.

This is because a validator gets access to the whole transaction from which an input emanates — see the line for Validator in Figure 1. (This is specific to EUTxO; UTxO is more local, see Remark 3.8.1.)

This makes it hard to factor out internal structure. For instance, even if all but one of the inputs and outputs of a transaction have been spent, so that it has just one dangling input — then that entire transaction is still observable at the final dangling input. Recall Proposition 3.5.12, and consider $tx, tx' \in \mathsf{Transaction}$ such that $pos(tx) \cap pos(tx') = \varnothing$, so that tx and tx' are commuting. We might

[44] G and F still have non-trivial work to do, as we see e.g. from Propositions 5.5.4 and 5.2.2.

reasonably wish to identify $tx \bullet tx'$ and $tx' \bullet tx$ with a composite transaction which we might write $tx \cup tx'$, but we cannot do this because a validator could see the difference.

Thus, currently the EUTxO framework can observe the difference between

— one large transaction, and
— a composite chain (i.e. a chunk; see Definition 3.4.1) of smaller ones

— even if they are 'morally' the same. This is not good for developing compositional theories of observational equivalence.

We propose it might be helpful if the EUTxO model allowed us to limit access to the channels in a transaction, such that an input can limit validators to access only certain inputs and outputs in a transaction, e.g. by nominating positions that are 'examinable' along that input.[45] The pure UTxO model (Remark 3.8.1) would correspond to the special case when an input nominates only itself for validators to access.

Then, we would in suitable circumstances be able to switch between a single large transaction, and a composite chunk with the same inputs and outputs.

Another observable of a transaction is the names of its spent transaction channels, and this brings us to garbage-collection:

8.2. Garbage-collection

Our model has no garbage-collection of spent positions — where by *spent positions* we mean the stx-atoms from Definition 3.4.8; see also the up-atoms from Definition 4.3.9 (a connection is precisely stated in Proposition 5.5.4).

There is nothing wrong with this; the EUTxO presentation in [CCM+20] does not garbage-collect either (i.e. positions on the blockchain can be occupied at most once, and can never be un-occupied).

However, since we name our positions, it might be nice to consider garbage-collecting (i.e. locally binding) the names of spent positions, by removing or α-converting them in some way. We do this in the implementation in [Gab20b].

However, the maths indicates that this does come at a certain price. For instance, consider a very simple model of finite lists of atoms in which the first atom is an 'input' and the last atom is an 'output', and we garbage-collect by removing matching atoms, like so:

$$[a,b] \bullet [b,c] = [a,c] \qquad [a,c] \bullet [c,b] = [a,b].$$

Thus,

$$([a,b] \bullet [b,c]) \bullet [c,b] = [a,b].$$

Now if we bracket the other way, then $[b,c] \bullet [c,b]$ is ill-defined, and therefore so is $[a,b] \bullet ([b,c] \bullet [c,b])$. We cannot have $[b,b]$ because this would violate the condition that an input must point to an earlier (not a later) output (Definition 3.4.1(3); we consider relaxing this condition in item 2 of Subsection 8.6).

The partiality is no issue — chunks are already a partial monoid (Theorem 3.5.4). But, our monoid of garbage-collecting chunks would not be *associative*. It would still be *nearly* associative, meaning that $x \bullet (y \bullet z) = (x \bullet y) \bullet z$ where both sides are defined. However this is a weaker property that would be messier to work with, and for now we do not need it to make our case.

(This could also be read as a mathematical hint that our uniqueness conditions are a little too strict, and that $[b,c] \bullet [c,b]$ should be permitted, where the leftmost b does not point to the later b but instead points backwards in time to some earlier b, and the rightmost b points forwards towards some later b — in the style of an *interleaved scope* [GGP15], for example. In the presence of garbage-collection this would make perfect sense, since we would expect both bs to eventually get bound, i.e. 'spent'.)

In this paper we do not garbage-collect. We leave deciding whether we should, and if so how, for future work.

[45] So a validator would be passed the *restriction* of a transaction obtained by including only nominated inputs and outputs in that transaction, and withholding the rest.

8.3. Tests

A few more words on the equations in Figure 1 (IEUTxO type equations) vs. those in Definition 4.5.2 (its algebraic counterpart of abstract chunk systems).

We want the equations in Figure 1 to be short and sweet, so that the system looks simple and solutions are easy to build and manipulate.

The design parameters in Definition 4.5.2 are somewhat different: we do not mind if there are plenty of algebraic properties, because this means that we have captured as many interesting properties as possible. The adjunction in Theorem 7.3.4 gives a formal sense in which the two correspond.

(This leaves it for future work to see how these conditions could be relaxed, as discussed e.g. in Remarks 6.4.4 and 7.1.5 and Proposition 7.3.6.)

This has mathematical interest, but not only that.

As noted in Subsection 4.4.2, modern programming languages support efficient programming on abstract denotations, thus delaying instantiating to specific instances until truly necessary. Also, they allow us to express and test against properties — and equality properties in particular can be helpful for optimising transformations.

So an algebraic theory can be relevant to producing concrete working code, because:

(1) it can help structure code; and
(2) an axiom can be read both as a testable property and as a program transformation; so that
(3) the more axioms we have, the more transformations and tests are available, and the more scope we have to transform, structure, and test our programs.

8.4. Connections with nominal techniques

This paper borrows ideas from *nominal techniques* [GP01] and in particular it follows the ideas on Equivariant ZFA from [Gab20a] in handling the atoms which we use to name positions. We use atoms to name positions in IEUTxO models and in abstract chunk systems (ACS).

Partly, this is just using nominal-style names and permutations as a standard vocabulary. We can think of this application of nominal ideas as reaching for a familiar API, typeclass, or algebra, and there is nothing wrong with that.

The reader can find this implemented in [Gab20b], where the IEUTxO equations in Figure 1 are combined with this author's nominal datatypes package to create first a Haskell typeclass, and then a working implementation of chunks and blockchains, following the IEUTxO model of this paper.

But in parts of this paper, something deeper is also taking place. For example:

(1) The notion of abstract chunk system — which makes up the more abstract half of this paper — depends on that of an oriented monoids of chunks from Definition 4.4.1, which depends on the notion of $posi$ from Definition 4.3.6, whose definition depends on the permutation action.
$posi$ is a nameful definition, in the nominal sense, and it is not clear how would express it, and thus the notion of ACS, were it not for our nominal use of names.
(2) In [Gab20b] we go somewhat further than in this paper in developing IEUTxO models, in that we permit α-conversion to garbage-collect spent positions as discussed in Subsection 8.2. This too is a fully nominal definition, which uses the nominal model of binding in specific ways.

Remark 8.4.1 (No support assumed). Experts on nominal techniques should note that no finite support conditions are imposed; for example, there is nothing to insist that a validator $\nu(v) \subseteq \beta \times \mathsf{Transaction}_!$ should be a finitely-supported subset. This paper is in ZFA, not Fraenkel-Mostowski set theory.

We could construct a finitely-supported account of the theory in this paper in FM, just by imposing finite support conditions appropriately — but it would cost us complexity, and since we do not seem to need this, we do not do it. We hope we have struck a good balance in the mathematics between being rigorous in our treatment of names, and not drowning the reader in detail.

Note that finiteness conditions on sets of names are still important: notably in Definition 4.4.1(1); and name-management is key to some nontrivial results, notably Corollary 4.4.5. So there is a 'finite

support' flavour to the maths, just not as a direct translation of the Fraenkel-Mostowski notion of finite support.

REMARK 8.4.2 (Disjointness conditions). Continuing the previous remark, 'freshness-flavoured' set disjointness conditions like

— $input(tx) \cap output(tx) = \varnothing$ in Lemma 3.4.6, and
— $pos(ch) \cap pos(ch') = \varnothing$ in Lemma 3.5.6, and
— $posi(x) \cap posi(y) = \varnothing$ in Definition 4.4.1 and elsewhere,

are quite important in this paper.

We could have imported a nominal notation and written these $\#$, as in $ch\#ch'$ or $x\#y$. This would not be wrong, but it might mislead because, as discussed above, we do not assume nominal notions of support and freshness. Thus, $pos(ch)$ and $posi(x)$ are *not* necessarily equal to the support of ch and x respectively, and if e.g. x and y were in an abstract chunk system that happened also to be finitely-supported (a plausible scenario), it would not be guaranteed that $x\#y$ would coincide with $posi(x) \cap posi(y) = \varnothing$.[46]

We could insist that $supp(x)$ must exist and coincide with $posi(x)$, of course — but that would be an additional restriction.

8.5. Concrete formalisation

Can we implement all or parts of this paper in a theorem-prover, and use that to verify properties of a blockchain system?

This paper has a lot of moving parts, and it is not all or nothing: a user can import whichever components (IEUTxO? ACS?) they wish — though we also hope that the overall mathematical vision could provide useful guidance.

How the ideas in this paper might be brought to bear in the setting of a formal verification cannot have a clear-cut answer, because it depends on interactions between the maths, what we want to verify, the resources available,[47] and what facilities are offered by a particular theorem-proving environment.

A reasonable (but naive) implementation of the EUTxO inductive definition in [CCM+20] would suggest modelling positions by numbers which are essentially de Bruijn indices. The maths in this paper suggests against this:

— We want to talk about chunks. Indices only make sense in a blockchain which has an initial genesis block from which we start to index.
— We want to rearrange transactions and chunks, e.g. to talk about how they commute, as in (for example) Definition 4.3.16. With indices, a transaction in a different place is a *different transaction*, and potentially subject to different validation if a validator were to directly inspect its indices.
— Because we use names, we never have to worry about reindexing functions, as can be an issue with the de Bruijn indexed approach.

This is borne out by the practice: the developers of the EUTxO implementation underlying [CCM+20] have explored theorem-provers and they do not reference transactions by position (even though the mathematical description in the literature makes it look like they do, at least to the uninformed reader).

What they actually do is maintain an explicit naming context. This is ongoing research, but the interested reader can find a brief description in [MSC19].

[46] $posi(x)\#posi(y)$ in the nominal sense would still mean $posi(x) \cap posi(y) = \varnothing$, just because nominal freshness coincides with sets disjointness on finite sets of atoms.

[47] Very important: a single person working alone will make different tradeoffs than a large team, and a short-term project will make different tradeoffs than a longer-term project that can e.g. afford to invest in building basic tooling.

The difficulty with this hands-on context-based approach is that we end up having to curate our context of names, using explicit context-weakening and context-combining operations. These get limited, extremely tedious to formulate and prove, and clutter the proofs of properties.[48]

It is standard and known that maintaining contexts of 'known names' can get painful. Indeed, this is one of the issues that nominal techniques were developed to alleviate.

In the context of this paper, we would advise looking at how the implementation in [Gab20b] manages names and binding using permutations and the `Nom` binding context (which is the nominal construct that closely corresponds to a local context of known names).

So it works in Haskell — but determining the extent to which this can be translated to a *theorem-prover* remains to be seen. We could imitate the nominal datatypes package; there is a nominal package in Isabelle [Urb08]; and this author has written some recommendations on implementing nominal techniques in theorem-proving environments [Gab20a].[49] Exploring this is future work.

8.6. Future work

We discussed future work above and in the body of the paper. We conclude with some further observations:

(1) The authors of [CCM+20] map their concrete models to *Constraint Emitting Machines*, which are a novel variant of Mealy machines. It is future work to see whether the idealised EUTxO solutions from Definition 3.1.2 admit corresponding descriptions. For the interested reader, we can note that a body of work on nominal automata does exist [BKL14; Boj18].

(2) In condition 3 of Definition 3.4.1 we restrict inputs to point to strictly earlier outputs. This restriction makes operational sense for a blockchain, and it is unavoidable and required for well-definedness in [CCM+20] because of its inductive construction.
However, there is no mathematical necessity to retain it here; it would be perfectly valid and possible to contemplate a generalisation of Definition 3.4.1 which permits loops from a transaction to itself, or even forward pointers from inputs to later outputs (i.e. 'feedback loops').
Mathematically and structurally, loops are perfectly admissible in the framework of this paper.
Loops from a transaction to itself are particularly interesting, because this would remove the need for a *genesis block* — which has no inputs, and therefore exists *sui generis* in that it is not subject to the action of any validation from other blocks. So, we could insist that all transactions have at least one input, but that input may loop to an output on the same transaction.
This might seem like a minor difference, but it is not, because *we* control the set of validators (the injection ν in Definition 3.1.2), so we can enforce checks of good behaviour — even of the first transaction, which would e.g. be forced to validate itself via a loop — by controlling the set of validators.

(3) We have considered EUTxO blockchains in this paper. It would be natural to attempt a similar analysis for an accounts-based blockchain architecture such as Ethereum. A start on this is the *Idealised Ethereum* equations in [BG20, Figure 4]; developing this further is future work.

(4) As is often the case, in practice there are desirable features of real systems that would break our model.
For instance, it can be useful to make transactions time-sensitive using *slot ranges*, also called *validity intervals*; a transaction can only be accepted into a block whose slot is inside the transaction's slot range. Clearly, this could compromise results having to do with chunks and commutations, because these are by design 'pure' notions, with no notion of time.
It is future work to see to what extent a theory of chunks might be compatible with explicit time dependence constraints. In one sense this temporal aspect should be orthogonal to the nominal techniques used so far, since atoms are just used as positions and we impose no finite support

[48] *Properties* often get called *meta-theorems* in this field. So wherever we write '(algebraic) property', a reader with a theorem-proving background can read 'meta-theorem', and they will not go too far wrong.

[49] This last paper essentially says: make sure that permuting names in properties is a pushbutton operation. Once you have that, the rest should follow; and if you do not then there may be trouble.

conditions; however in another sense it may also be possible to usefully import notions of e.g. ordered atoms from nominal automata, as presented in [BKL14; Boj18]. In any case, it is common to find a pure sublanguage with a nice theory embedded in a more expressive and larger language with less good behaviour, as e.g. pure SML is embedded in SML with global variables.

8.7. Final words

The slogan of this paper is *blockchains as algebras*, and more specifically *blockchains as nominal algebras of chunks*. Based on the maths above, which substantiates this slogan, we can note that:

(1) UTxO blockchains can be viewed as an algebraic structure, both intensionally (Definition 3.1.2) and extensionally (Definition 4.5.1), and these views are equivalent (Theorem 7.3.4).
(2) The natural and fundamental unit of blockchain algebras is *partial* blockchains — what we call *chunks* in this paper — and chunks naturally organise themselves into monoids with extra structure.
(3) It can be convenient to address inputs and outputs by *name* in a 'nominal' style — contrast with a de Bruijn-style index (addressing a location by some kind of offset from a genesis block; cf. Subsection 8.5), or a blockchain hash.
(4) We have proved some high-level properties (some readers might call these *meta-theorems*). Notably: a kind of Church-Rosser property in Theorem 3.6.1; and Theorem 7.3.4(4) that every IEUTxO system admits a presentation as an IUTxO system; and that both can be presented as Abstract Chunk Systems, in senses made formal by a loop of embeddings in Remark 7.3.5. These capture global properties of how the structures fit together, which are not immediately obvious just reading the definitions.
(5) An open question is how this maths might help the blockchain community to simplify proofs, write better programs, more quickly and safely design new systems, or accommodate existing and new extensions (cf. next point). The results in this paper, coupled with a toy (but perfectly real) blockchain implementation based on these ideas [Gab20b], suggests that these things are plausible — but time and future research will reveal more, and that is as it should be.
(6) We knowingly threw out concrete structure that real blockchains need. *Real* blockchains have tokens, monetary policies, smart contracts, and more structure that is being invented literally on a daily basis. But instantiating algebraic structures is possible as discussed in Subsection 2.1 and indeed the *raison d'être* of algebra is just this: abstract, find instances, and extend.
(7) One path to applying this paper is essentially to take Proposition 3.5.12 and Remark 3.5.13 seriously and use them to design domain-specific resource-aware logics for reasoning about algebras of chunks equipped with specific features of possibly industrial relevance.
These would be logics for verifying domain-specific elaborations of Theorem 3.6.1, and for tracing resources through the evolution of a blockchain. The nominal method of tracking resources (e.g. as we have seen used to label inputs and outputs and define Abstract Chunk Systems) lends itself naturally to such applications.[50]
Thus the semantics here, enriched with data structures of practical interest — e.g. currencies, NFTs, or other 'real' data coming from specific user needs — could lead to logics that are both powerful and useful, helping to prove commutativity and other resource-based properties, using logics for algebras of chunks in the style of this paper.

In summary: this paper introduces the idea of *blockchain algebra*, at least as applied to UTxO-style blockchains. It is reasonable to hope that this might provide a convenient target semantics for new programs and logics for building and reasoning about blockchain systems.

[50] Nominal techniques were originally designed to track resources (namely: variable symbols in inductive definitions with binding) and they have been elaborated considerably since e.g. through nominal rewriting, automata, and nominal algebra. So there is a body of theory to draw on.

ACKNOWLEDGMENTS

I thank the editors and three anonymous referees for their detailed and constructive feedback: thank you for your time and input. Thanks also to Lars Brünjes, without whom this paper might not have been written. I dedicate this paper to the memory of our colleague Martin Hofmann. May he rest in peace.

REFERENCES

[Acz88] Peter Aczel, *Non-wellfounded set theory*, CSLI lecture notes, no. 14, CSLI, 1988.

[Bar84] Henk P. Barendregt, *The lambda calculus: its syntax and semantics (revised ed.)*, North-Holland, 1984.

[BG20] Lars Brünjes and Murdoch J. Gabbay, *UTxO- vs account-based smart contract blockchain programming paradigms*, Proceedings of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2020), Springer, October 2020, See arXiv preprint https://arxiv.org/abs/2003.14271.

[BKL14] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota, *Automata theory in nominal sets*, Logical Methods in Computer Science **10** (2014).

[Boj18] Mikołaj Bojańczyk, *Slightly infinite sets (book draft)*, 2018.

[CCM+20] Manuel M.T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler, *The Extended UTXO model*, Proceedings of the 4th Workshop on Trusted Smart Contracts (WTSC'2020), LNCS, vol. 12063, Springer, 2020.

[CFW86] William D. Clinger, Dan P. Friedman, and Mitchell Wand, *A scheme for a higher-level semantic algebra*, p. 237–250, Cambridge University Press, USA, 1986.

[Gab01] Murdoch J. Gabbay, *A Theory of Inductive Definitions with alpha-Equivalence*, Ph.D. thesis, University of Cambridge, UK, March 2001.

[Gab20a] ———, *Equivariant ZFA and the foundations of nominal techniques*, Journal of Logic and Computation **30** (2020), 525–548.

[Gab20b] ———, *Implementation of idealised EUTxO in the nominal datatypes package*, https://github.com/bellissimogiorno/nominal/blob/6e9c/src/Language/Nominal/Examples/IdealisedEUTxO.hs, June 2020.

[GGP15] Murdoch J. Gabbay, Dan R. Ghica, and Daniela Petrisan, *Leaving the Nest: Nominal Techniques for Variables with Interleaving Scopes*, 24th EACSL Annual Conference on Computer Science Logic (CSL 2015) (Dagstuhl, Germany) (Stephan Kreutzer, ed.), Leibniz International Proceedings in Informatics (LIPIcs), vol. 41, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 374–389.

[GP01] Murdoch J. Gabbay and Andrew M. Pitts, *A New Approach to Abstract Syntax with Variable Binding*, Formal Aspects of Computing **13** (2001), no. 3–5, 341–363.

[Mil99] Robin Milner, *Communicating and mobile systems: the π-calculus*, Cambridge University Press, 1999.

[Mok17] Andrey Mokhov, *Algebraic graphs with class (functional pearl)*, SIGPLAN Not. **52** (2017), no. 10, 2–13.

[MSC19] Orestis Melkonian, Wouter Swierstra, and Manuel MT Chakravarty, *Formal investigation of the Extended UTxO model (Extended Abstract)*, https://omelkonian.github.io/data/publications/formal-utxo.pdf, 2019.

[Nes21] Chad Nester, *A Foundation for Ledger Structures*, 2nd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2020) (Dagstuhl, Germany) (Emmanuelle Anceaume, Christophe Bisière, Matthieu Bouvard, Quentin Bramas, and Catherine Casamatta, eds.), Open Access Series in Informatics (OASIcs), vol. 82, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, pp. 7:1–7:13.

[Rey02] John C. Reynolds, *Separation logic: A logic for shared mutable data structures*, Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002), IEEE Computer Society Press, 2002, pp. 55–74.

[Urb08] Christian Urban, *Nominal reasoning techniques in Isabelle/HOL*, Journal of Automatic Reasoning **40** (2008), no. 4, 327–356.