UTXO_{ma}: UTXO with Multi-Asset Support

Manuel M.T. Chakravarty[1], James Chapman[1], Kenneth MacKenzie[1], Orestis Melkonian[1,2], Jann Müller[1], Michael Peyton Jones[1], Polina Vinogradova[1], and Philip Wadler[1,2]

[1] IOHK, firstname.lastname@iohk.io
[2] University of Edinburgh, orestis.melkonian@ed.ac.uk, wadler@inf.ed.ac.uk

Abstract. The most prominent use case of Ethereum which is not supported by Bitcoin is the creation of a wide range of *user-defined tokens* or *assets* by way of smart contracts. User-defined assets are *non-native* on Ethereum; i.e., they are not directly supported by the ledger, but require repetitive custom code. This makes them unnecessarily inefficient, expensive, and complex. It also makes them insecure as numerous incidents on Ethereum have demonstrated. In this paper, we explore an alternative design based on Bitcoin-style UTXO ledgers. Instead of introducing general scripting capabilities together with the associated security risks, we propose an extension of the UTXO model, where we replace the accounting structure of a single cryptocurrency with a new structure that manages an unbounded number of user-defined, native tokens, which we call *token bundles.* Token creation is controlled by *forging policy scripts* that, just like Bitcoin validator scripts, use a small domain-specific language with bounded computational expressiveness, thus favouring Bitcoin's security and computational austerity. The resulting approach is lightweight, i.e., custom asset creation and transfer is cheap, and it avoids use of any global state in the form of an asset registry or similar.

Keywords: blockchain · UTXO · native tokens · functional programming.

1 Introduction

Distributed ledgers began by tracking just a single asset: money. The goal was to compete with existing currencies, and so they naturally started by focusing on their own currencies: Bitcoin and its eponymous currency, Ethereum and Ether, and so on. This focus was so clear that the systems tended to be identified with their primary currency.

More recently, it has become clear that it is possible and very useful to track other kinds of asset on distributed ledger systems. Ethereum has led the innovation in this space, with ERC-20 [12] implementing new currencies and ERC-721 [6] implementing unique non-fungible tokens.

These have been wildly popular: variants of ERC-20 are the most used smart contracts on Ethereum by some margin. However, they have major shortcomings. Notably, custom tokens on Ethereum are not native: the tokens do not live in a user's account, and in order to send another user ERC-20 tokens, the sender must interact with the governing smart contract for the currency. That is, despite the main purpose of the system Ethereum has built being to track ownership of assets and perform transactions, users have been forced to build their own *internal ledger* inside a smart contract.

Other systems have learned from this and have made custom tokens native, such as Stellar, Waves, Zilliqa, and more. However, these typically rely on some kind of global state, such as a global currency registry, or special global accounts that must be created. This is slow, and restricts creative use of custom tokens because of the high overhead in terms of time and money.

We can do better than this through a combination of two ideas. Firstly, we generalise the value type that the ledger works with to include *token bundles* that freely and uniformly mix tokens from different custom assets, both fungible and non-fungible. Secondly, we avoid any global state by "eternally" linking a currency to a governing forging policy via a hash. Between them this creates a multi-asset ledger system (which we call $\text{UTXO}_{\mathsf{ma}}$) with native, lightweight custom tokens.

Specifically, this paper makes the following contributions:

- We introduce token bundles, represented as finitely-supported functions, as a uniform mechanism to generalise the existing UTXO accounting rules to custom asserts including fungible, non-fungible, and mixed tokens.
- We avoid the need for global state in the form of a currency registry by linking custom forging policy scripts by way of their script hash to the name of asset groups.
- We support a wide range of standard applications for custom assets without the need for general-purpose smart contracts by defining a simple domain-specific language for forging policy scripts.
- We provide a formal definition of the $\text{UTXO}_{\mathsf{ma}}$ ledger rules as a basis to formally reason about the resulting system.

Creating and transferring new assets in the resulting system is lightweight and cheap. It is lightweight as we avoid special setup transactions or registration procedures, and it is cheap as only standard transaction fees are required — this is unlike the substantial costs that custom tokens can incur on Ethereum due to contract gas costs.

The proposed multi-asset system is not merely a pen and paper exercise. It forms the basis of the multi-asset support for the Cardano blockchain.

2 Multi-Asset Support

In Bitcoin's ledger model [8,2,15], transactions spend as yet *unspent transaction outputs* (*UTXOs*), while supplying new unspent outputs to be consumed by subsequent transactions. Each individual UTXO locks a specific *quantity* of cryptocurrency by imposing specific conditions that need to be met to spend that quantity, such as for example signing the spending transaction with a specific secret cryptographic key, or passing some more sophisticated conditions enforced by a *validator script*. Quantities of cryptocurrency in a transaction output are represented as an integral number of the smallest unit of that particular cryptocurrency — in Bitcoin, these are Satoshis. To natively support multiple currencies in transaction outputs, we generalise those integral quantities to natively support the dynamic creation of new user-defined *assets* or *tokens*. Moreover, we require a means to forge tokens in a manner controlled by an asset's *forging policy*.

We achieve all this by the following three extensions to the basic UTXO ledger model that are further detailed in the remainder of this section.

1. Transaction outputs lock a *heterogeneous token bundle* instead of only an integral value of one cryptocurrency.
2. We extend transactions with a *forge* field. This is a token bundle of tokens that are created (minted) or destroyed (burned) by that transaction.
3. We introduce *forging policy scripts* (*FPS*) that govern the creation and destruction of assets in forge fields. These scripts are not unlike the validators locking outputs in UTXO.

2.1 Token bundles

We can regard transaction outputs in an UTXO ledger as pairs $(value, \nu)$ consisting of a locked value *value* and a validator script ν that encodes the spending condition. The latter may be proof of ownership by way of signing the spending transaction with a specific secret cryptography key or a temporal condition that allows an output to be spent only when the blockchain has reached a certain height (i.e. a certain number of blocks have been produced).

To conveniently use multiple currencies in transaction outputs, we want each output to be able to lock varying quantities of multiple different currencies at once in its *value* field. This suggests using finite maps from some kind of *asset identifier* to an integral quantity as a concrete representation. However, looking at the standard UTXO ledger rules [15], it becomes apparent that cryptocurrency quantities need to be monoids. It is a little tricky to make finite maps into a monoid, but the solution is to think of them as *finitely supported functions*. These can, in the end, be implemented with maps, and we will discuss them formally in Section 3.

It might be tempting to stop here and represent token bundles simply by way of a finitely-supported function from an asset identifier to an integral quantity of tokens of that asset. And if all we wanted to define was fungible cryptocurrencies, we would be done.

However, if we also want to handle groups of related, but *non-fungible* tokens, we need to go a step further. Let us assume we want to create tokens to represent specific physical assets, such as ownership of a house. Then, for the houses to not be interchangeable, an output locking the tokens representing ownership over three different houses must be a set of tokens with different names, so as not to confuse them with other houses.

Unfortunately, if want to create an output mixing fungible and non-fungible tokens, we now must mix the finitely-supported functions tracking fungible tokens with sets containing non-fungible tokens. To achieve a uniform representation for both, it is useful to remember that a set is isomorphic to a finite map with a unary co-domain. In other words, we can use finitely-supported functions for both, but for a non-fungible token, its quantity is just one.

In order to not lose the grouping of related non-fungible tokens (all house tokens issued by a specific entity, for example) though, we need to move to a two-level structure — i.e., finitely-supported functions of finitely-supported functions. Hence, to represent five tokens of a cryptocurrency Coin, we have $\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 5\}\}$ and, for three specific house, we have $\{\mathsf{Houses} \mapsto \{\mathsf{House1} \mapsto 1, \mathsf{House2} \mapsto 1, \mathsf{House3} \mapsto 1\}\}$.

To lock them all in one output, we simply use $\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 5\}, \mathsf{Houses} \mapsto \{\mathsf{House1} \mapsto 1, \mathsf{House2} \mapsto 1, \mathsf{House3} \mapsto 1\}\}$. This structure suffices to do what we set out to, but is its generality really warranted?

Let's consider another example. Trading of rare in-game items is popular in modern, multi-player computer games. How about representing ownership of such items and trading of that ownership on our multi-asset UTXO ledger? We might need tokens for "hats" and "swords", which form two non-fungible assets with possibly multiple tokens of each asset: a hat is interchangeable with any other hat, but not with a sword.

Here our two-level structure pays off in its full generality. We can represent currency to purchase items together with sets of items, where some can be multiples; for example, as in $\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 3\}, \mathsf{Game} \mapsto \{\mathsf{Hat} \mapsto 1, \mathsf{Sword} \mapsto 5, \mathsf{Owl} \mapsto 1\}\}$.

Values represented in this form can be added naturally; e.g.,

$$\begin{aligned} &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 2\}, \mathsf{Game} \mapsto \{\mathsf{Hat} \mapsto 1, \mathsf{Sword} \mapsto 4\}\} \\ +\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 1\}, \mathsf{Game} \mapsto \{\mathsf{Sword} \mapsto 1, \mathsf{Owl} \mapsto 1\}\} \\ =\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 3\}, \mathsf{Game} \mapsto \{\mathsf{Hat} \mapsto 1, \mathsf{Sword} \mapsto 5, \mathsf{Owl} \mapsto 1\}\}\ . \end{aligned}$$

2.2 Forge fields

If new tokens are frequently generated (such as issuing new hats whenever an in-game achievement has been reached) and destroyed (a player may lose a hat forever if the wind picks up), these operations need to be lightweight and cheap. We achieve this by adding a forge field to every transaction. It is a token bundle (just like the *value* in an output), but admits positive quantities (for minting new tokens) and negative quantities (for burning existing tokens). Of course, minting and burning needs to be strictly controlled.

2.3 Forging policy scripts

Whenever a token is minted or burned in a forge field of a transaction tx, that transaction must include the forging policy script applying to that token. For tx to be valid, all included forging policy scripts must approve of the transaction. This mechanism is similar to the validator scripts that already exist in the UTXO ledger model: in order to for a transaction to spend an output $(value, \nu)$, the validator script ν needs to be executed and approve of the spending transaction.

Validator scripts in Bitcoin offer only very limited functionality, to keep script execution cheap and predictable. In this spirit, we also keep the functionality available to forging policy scripts strictly limited — the details will follow Section 4, but we support forging policies for common use cases, such as single issue tokens, timed issue tokens, tokens whose forging is controlled by a single authority, and the ever popular non-fungible tokens, as well as many more.

There remains one question to be answered: how do we know which forging policy script is responsible for which tokens? As an example, given a transaction tx with a forge field containing $\{\mathsf{Game} \mapsto \{\mathsf{Owl} \mapsto 1\}\}$, how do we know the rules for forging Owl?

The standard approach is to maintain a global registry or account structure, where an identifier like Game, or rather a cryptographic hash representing Game, is associated with some smart contract or script code. Such a global structure, then, leads to questions. How are entries registered? Can they be destroyed again? What are the incentives to destroy them when unused, so that they do not bloat the blockchain state forever?

We propose a much simpler solution. Instead of using an arbitrary string, like Game, for a group of assets, we use the cryptographic hash of the forging policy script for that group of assets as the key of the finitely-supported function. In other words, if a forge field contain $\{\mathsf{HASHVALUE} \mapsto \{\mathsf{Owl} \mapsto 1\}\}$, the transaction is only valid if it is accompanied by a forging policy script whose cryptographic hash is $\mathsf{HASHVALUE}$ and if execution of that script succeeds. (The mapping from $\mathsf{HASHVALUE}$ to the fact that it controls the tokens of the game is a question of metadata, like the mapping from account addresses to their owners in Ethereum.)

3 Formal ledger rules

Our formal ledger model follows the style of the UTXO-with-scripts model from [15] adopting the notation from [3] with basic types defined as in Figure 1.

$\mathbb{B}$	the type of Booleans
$\mathbb{N}$	the type of natural numbers
$\mathbb{Z}$	the type of integers
$\mathbb{H}$	the type of bytestrings: $\bigcup_{n=0}^{\infty}\{0,1\}^{8n}$
$(\phi_1 : T_1, \ldots, \phi_n : T_n)$	a record type with fields $\phi_1, \ldots, \phi_n$ of types $T_1, \ldots, T_n$
$t.\phi$	the value of ϕ for t, where t has type T and ϕ is a field of T
$\mathsf{Set}[T]$	the type of (finite) sets over T
$\mathsf{List}[T]$	the type of lists over T, with $\_[\_]$ as indexing and $\lvert\_\rvert$ as length
$h :: t$	the list with head h and tail t
$x \mapsto f(x)$	an anonymous function
$c^{\#}$	a cryptographic collision-resistant hash of c
$\mathsf{Interval}[A]$	the type of intervals over a totally-ordered set A
$\mathsf{FinSup}[K, M]$	the type of finitely supported functions from a type K to a monoid M

Fig. 1: Basic types and notation

Finitely-supported functions. We model token bundles as finitely-supported functions. If K is any type and M is a monoid with identity element 0, then a function $f : K \to M$ is *finitely supported* if $f(k) \neq 0$ for only finitely many $k \in K$. More precisely, for $f : K \to M$ we define the *support* of f to be $\mathsf{supp}(f) = \{k \in K : f(k) \neq 0\}$ and $\mathsf{FinSup}[K, M] = \{f : K \to M : |\mathsf{supp}(f)| < \infty\}$.

If $(M, +, 0)$ is a monoid then $\mathsf{FinSup}[K, M]$ also becomes a monoid if we define addition pointwise ($(f + g)(k) = f(k) + g(k)$), with the identity element being the zero map. Furthermore, if M is an abelian group then $\mathsf{FinSup}[K, M]$ is also an abelian group under this construction, with $(-f)(k) = -f(k)$. Similarly, if M is partially ordered, then so is $\mathsf{FinSup}[K, M]$ with comparison defined pointwise: $f \leq g$ if and only if $f(k) \leq g(k)$ for all $k \in K$.

It follows that if M is a (partially ordered) monoid or abelian group then so is $\mathsf{FinSup}[K, \mathsf{FinSup}[L, M]]$ for any two sets of keys K and L. We will make use of this fact in the validation rules presented later in the paper (see Figure 4).

Finitely-supported functions are easily implemented as finite maps, with a failed map lookup corresponding to returning 0.

3.1 Ledger types

Figure 2 defines the ledger primitives and types that we need to define the UTXO$_{\mathsf{ma}}$ model. All outputs use a pay-to-script-hash scheme, where an output is locked with the hash of a script. We use a single scripting language for forging policies and to define output locking scripts. Just as in Bitcoin, this is a restricted domain-specific language (and not a general-purpose language) — the details follow in Section 4. We assume that each transaction has a unique identifier derived from its value by a hash function. This is the basis of the lookupTx function to look up a transaction, given its unique identifier.

Token bundles. We generalise per-output transferred quantities from a plain Quantity to a bundle of Quantities. A Quantities represents a token bundle: it is a mapping from a policy and a *asset*,

Ledger primitives

$\mathsf{Quantity}$	an amount of currency, forming an abelian group (typically $\mathbb{Z}$)
Asset	a type consisting of identifiers for individual asset classes
Tick	a tick
$\mathsf{Address}$	an "address" in the blockchain
TxId	the identifier of a transaction
$\mathsf{txId} : \mathsf{Tx} \to \mathsf{TxId}$	a function computing the identifier of a transaction
$\mathsf{lookupTx} : \mathsf{Ledger} \times \mathsf{TxId} \to \mathsf{Tx}$	retrieve the unique transaction with a given identifier
$\mathsf{verify} : \mathsf{PubKey} \times \mathbb{H} \times \mathbb{H} \to \mathbb{B}$	signature verification
$\mathsf{FPScript}$	forging policy scripts
$\mathsf{scriptAddr} : \mathsf{FPScript} \to \mathsf{Address}$	the address of a script

Ledger types

$$
\begin{aligned}
\mathsf{PolicyID} &= \mathsf{Address} && \text{(an identifier for a custom currency)}\\
\mathsf{Signature} &= \mathbb{H}\\
\\
\mathsf{Quantities} &= \mathsf{FinSup}[\mathsf{PolicyID}, \mathsf{FinSup}[\mathsf{Asset}, \mathsf{Quantity}]]\\
\\
\mathsf{Output} &= (\mathit{addr} : \mathsf{Address}, \mathit{value} : \mathsf{Quantities})\\
\\
\mathsf{OutputRef} &= (\mathit{id} : \mathsf{TxId}, \mathit{index} : \mathsf{Int})\\
\\
\mathsf{Input} &= (\mathit{outputRef} : \mathsf{OutputRef}\\
&\qquad \mathit{validator} : \mathsf{FPScript})\\
\\
\mathsf{Tx} &= (\mathit{inputs} : \mathsf{Set}[\mathsf{Input}],\\
&\qquad \mathit{outputs} : \mathsf{List}[\mathsf{Output}],\\
&\qquad \mathit{validityInterval} : \mathsf{Interval}[\mathsf{Tick}],\\
&\qquad \mathit{forge} : \mathsf{Quantities}\\
&\qquad \mathit{scripts} : \mathsf{Set}[\mathsf{FPScript}],\\
&\qquad \mathit{sigs} : \mathsf{Set}[\mathsf{Signature}])\\
\\
\mathsf{Ledger} &= \mathsf{List}[\mathsf{Tx}]
\end{aligned}
$$

Fig. 2: Ledger primitives and basic types

which defines the asset class, to a $\mathsf{Quantity}$ of that asset.[3] Since a $\mathsf{Quantities}$ is indexed in this way, it can represent any combination of tokens from any assets (hence why we call it a token *bundle*).

Asset groups and forging policy scripts. A key concept is the *asset group*. An asset group is identified by the hash of special script that controls the creation and destruction of asset tokens of that asset group. We call this script the *forging policy script*.

[3] We have chosen to represent $\mathsf{Quantities}$ as a finitely-supported function whose values are themselves finitely-supported functions (in an implementation, this would be a nested map). We did this to make the definition of the rules simpler (in particular Rule 8). However, it could equally well be defined as a finitely-supported function from tuples of $\mathsf{PolicyID}$s and Assets to $\mathsf{Quantity}$s.

Forging. Each transaction gets a *forge* field, which simply modifies the required balance of the transaction by the Quantities inside it: thus a positive *forge* field indicates the creation of new tokens. In contrast to outputs, Quantities in forge fields can also be negative, which effectively burns existing tokens.[4]

Additionally, transactions get a *scripts* field holding a set of forging policy scripts Set[FPScript]. This provides the forging policy scripts that are required as part of validation when tokens are minted or destroyed (see Rule 8). A forging policy script is executed in a context that provides access to the main components of the forging transaction, the UTXOs it spends, and the policy ID. The passing of the context provides a crucial piece of the puzzle regarding self-identification: it includes the script's own PolicyID, which avoids the problem of trying to include the hash of a script inside itself.

Validity intervals. A transaction's *validity interval* field contains an interval of ticks (monotonically increasing units of "time", from [3]). The validity interval states that the transaction must only be validated if the current tick is within the interval. The validity interval, rather than the actual current chain tick value, must be used for script validation. In an otherwise valid transaction, passing the current tick to the evaluator could result in different script validation outcomes at different ticks, which would be problematic.

Language clauses. In our choice of the set of predicates `p1, ..., pn` to include in the scripting language definition, we adhere to the following heuristic: we only admit predicates with quantification over finite structures passed to the evaluator in the transaction-specific data, i.e. sets, maps, and lists. The computations we allow in the predicates themselves are well-known computable functions, such as hashing, signature checking, arithmetic operations, comparisons, etc. These boolean predicates, which make up the basis of the scripting language, are expressed directly in the underlying language used to specify the functionality of our model as a whole. Therefore, they do not need to be interpreted.

The gamut of policies expressible in the model we propose here is fully determined by the collection of predicates, assembled into a single script by logical connectives `&&`, `||`, and `Not`. As a result, proving of any statements about any variation of the language is trivial. Due to the expressiveness of the underlying language, the resulting policies can themselves be quite expressive, as we will demonstrate in the upcoming applications section. This holds true despite the constraints we place on admissible clauses, and without the need for specifying the semantics of a new language.

When specifying forging predicates, we use `tx._` notation to access the fields of a transaction.

3.2 Transaction validity

Figure 4 defines what it means for a transaction t to be valid for a valid ledger l during the tick currentTick, using some auxiliary functions from Figure 3. A ledger l is *valid* if either l is empty or l is of the form $t :: l'$ with l' valid and t valid for l'.

The rules follow the usual structure for an UTXO ledger, with a number of modifications and additions. The new **Forging** rule (Rule 8) implements the support for forging policies by requiring that the currency's forging policy is included in the transaction — along with Rule 9 which ensures that they are actually run! The arguments that a script is applied to are the ones discussed earlier.

[4] The restriction on outputs is enforced by Rule 2 — we simply do not impose such a restriction on the *forge* field. This lets us define rules in a simpler way, with cleaner notation.

$$
\begin{aligned}
&\mathsf{unspentTxOutputs} : \mathsf{Tx} \to \mathsf{Set}[\mathsf{OutputRef}] \\
&\mathsf{unspentTxOutputs}(t) \;= \{(\mathsf{txId}(t), 1), \ldots, (\mathsf{txId}(id), |t.outputs|)\} \\
\\
&\mathsf{unspentOutputs} : \mathsf{Ledger} \to \mathsf{Set}[\mathsf{OutputRef}] \\
&\mathsf{unspentOutputs}([]) \quad = \{\} \\
&\mathsf{unspentOutputs}(t :: l) = (\mathsf{unspentOutputs}(l) \setminus t.inputs) \cup \mathsf{unspentTxOutputs}(t) \\
\\
&\mathsf{getSpentOutput} : \mathsf{Input} \times \mathsf{Ledger} \to \mathsf{Output} \\
&\mathsf{getSpentOutput}(i, l) \;= \mathsf{lookupTx}(l, i.outputRef.id).outputs[i.outputRef.index]
\end{aligned}
$$

Fig. 3: Auxiliary validation functions

When forging policy scripts are run, they are provided with the appropriate transaction data, which allows them to enforce conditions on it. In particular, they can inspect the *forge* field on the transaction, and so a forging policy script can identify how much of its own currency was forged, which is typically a key consideration in whether to allow the transaction.

We also need to be careful to ensure that transactions in our new system preserve value correctly. There are two aspects to consider:

1. We generalise the type of value to Quantities. However, since Quantities is a monoid, Rule 4 is (almost) identical to the one in the original UTXO model, simply with a different monoid. Concretely, this amounts to preserving the quantities of each of the individual token classes in the transaction.
2. We allow forging of new tokens by including the forge field into the balance in Rule 4.

4 A stateless forging policy language

The domain-specific language for forging policies strikes a balance between expressiveness and simplicity. It particular, it is stateless and of bounded computational complexity. Nevertheless, it is sufficient to support the applications described in Section 5.

Semantically meaningful token names. The policy ID is associated with a policy script (it is the hash of it), so it has a semantic meaning that is identified with that of the script. In the clauses of our language, we give semantic meaning to the names of the tokens as well. This allows us to make some judgements about them in a programmatic way, beyond confirming that the preservation of value holds, or which ones are fungible with each other. For example, the `FreshTokens` constructor gives us a way to programmatically generate token names which, by construction, mean that these tokens are unique, without ever checking the global ledger state.

Forging policy scripts as output-locking scripts. As with currency in the non-digital world, it is a harder problem to control the transfer of assets once they have come into circulation (see also Section 6.1). We can, however, specify directly in the forging policy that the assets being forged must be locked by an output script of our choosing. Moreover, we can use the asset policy ID and the address interchangeably. The `AssetToAddress` clause is used for this purpose.

1. **The current tick is within the validity interval**

$$\mathsf{currentTick} \in t.validityInterval$$

2. **All outputs have non-negative values**

$$\text{For all } o \in t.outputs,\ o.value \geq 0$$

3. **All inputs refer to unspent outputs**

$$\{i.outputRef : i \in t.inputs\} \subseteq \mathsf{unspentOutputs}(l).$$

4. **Value is preserved**

$$t.forge + \sum_{i \in t.inputs} \mathsf{getSpentOutput}(i, l) = \sum_{o \in t.outputs} o.value$$

5. **No output is double spent**

$$\text{If } i_1, i \in t.inputs \text{ and } i_1.outputRef = i.outputRef \text{ then } i_1 = i.$$

6. **All inputs validate**

$$\text{For all } i = t.inputs[n],\ [\![i.validator]\!](t, n) = \mathsf{true}$$

7. **Validator scripts match output addresses**

$$\text{For all } i \in t.inputs,\ \mathsf{scriptAddr}(i.validator) = \mathsf{getSpentOutput}(i, l).addr$$

8. **Forging**
   A transaction with a non-zero *forge* field is only valid if either:
   (a) the ledger l is empty (that is, if it is the initial transaction).
   (b) for every key $h \in \mathsf{supp}(t.forge)$, there exists $s \in t.scripts$ with $h = \mathsf{scriptAddr}(s)$.
9. **All forging policies validate**

$$\text{For all } s \in t.scripts,\ [\![s]\!](\mathsf{scriptAddr}(s), t, \mathsf{getSpentOutput}(t.inputs, l)) = \mathsf{true}$$

Fig. 4: Validity of a transaction t in a ledger l

Language clauses. The various clauses of the validator and forging policy language are as described below, with their formal semantics as in Figure 5. In this figure, we use the notation `x` $\mapsto$ `y` to represent a single key-value pair of a finite map.

- `JustMSig(s)` verifies that the m-of-n signatures required by s are in the set of signatures provided by the transaction. We do not give the multi-signature script evaluator details as this is a common concept, and use the $[\![\_]\!]$ notation for it as well.
- `SpendsOutput(o)` checks that the transaction spends the output referenced by `o` in the UTXO.
- `TickAfter(tick1)` checks that the validity interval of the current transaction starts after time `tick1`.
- `Forges(tkns)` checks that the transaction forges exactly `tkns` of the asset with the policy ID that is being validated.
- `Burns(tkns)` checks that the transaction burns exactly `tkns` of the asset with the policy ID that is being validated.

```
⟦JustMSig(s)⟧(h, tx, utxo) = ⟦s⟧

⟦SpendsOutput(o)⟧(h, tx, utxo) = o ∈ tx.inputs

⟦TickAfter(tick1)⟧(h, tx, utxo) = tick1 ≤ min.validityInterval.tx

⟦Forges(tkns)⟧(h, tx, utxo) = h ↦ tkns ⊆ (tx.forge) && (h ↦ tkns) ≥ 0

⟦Burns(tkns)⟧(h, tx, utxo) = h ↦ tkns ⊆ (tx.forge) && (h ↦ tkns) ≤ 0

⟦FreshTokens⟧(h, tx, utxo) =
   ∀ pid ↦ tkns ∈ (tx.forge), pid == h ⇒
     ∀ t ↦ q ∈ tkns,
       t == hash (indexof(t, tkns), tx.inputs) && q == 1

⟦AssetToAddress(pid, addr)⟧(h, tx, utxo) =
   ∀ pid ↦ tkns ∈ utxo.balance, pid == h ⇒
      addr == _ ⇒ (h, pid ↦ tkns) ∈ utxo
    ∧ addr ≠ _ ⇒ (addr, pid ↦ tkns) ∈ utxo

⟦DoForge⟧(h, tx, utxo) = h ∈ dom(tx.forge)

⟦SignedByPIDToken⟧(h, tx, utxo) =
   ∀ pid ↦ tkns ∈ utxo.balance, pid == h ⇒
     ∀ s ∈ tx.sigs, ∃ t ∈ dom(tkns),
       isSignatureOnTxBy(s, tx, t)

⟦SpendsCur(pid)⟧(h, tx, utxo) =
     pid == _ ⇒ h ∈ domain(utxo.balance)
   ∧ pid ≠ _ ⇒ pid ∈ domain(utxo.balance)
```

Fig. 5: Forging Policy Language

- `FreshTokens` checks that all tokens of the asset being forged are non-fungible.
  This script must check that the names of the tokens in this token bundle are generated by hashing some unique data. This data must be unique to both the transaction itself and the token within the asset being forged. In particular, we can hash a pair of
  1. some *output* in the UTXO that the transaction consumes, and
  2. the *index* of the token name in (the list representation of) the map of tokens being forged (under the specific policy, by this transaction). We denote the function that gets the index of a key in a key-value map by `indexof`.
- `AssetToAddress(addr)` checks that all the tokens associated with the policy ID that is equal to the hash of the script being run are output to an UTXO with the address `addr`. In the case that no `addr` value is provided (represented by _), we use the `addr` value passed to the evaluator as the hash of the policy of the asset being forged.

- `DoForge` checks that this transaction forges tokens in the bundle controlled by the policy ID that is passed to the FPS script evaluator (here, again, we make use of the separate passing of the FPS script and the policy ID).
- `SignedByPIDToken(pid)` verifies the hash of every key that has signed the transaction.
- `SpendsCur(pid)` verifies that the transaction is spending assets in the token bundle with policy ID `pid` (which is specified as part of the *constructor*, and may be different than the policy ID passed to the evaluator).

5 Applications

UTXO$_{ma}$ is able to support a large number of standard use cases for multi-asset ledgers, as well as some novel ones. In this section we give a selection of examples. There are some common themes: (1) Tokens as resources can be used to reify many non-obvious things, which makes them first-class tradeable items; (2) cheap tokens allow us to solve many small problems with *more tokens*; and (3) the power of the scripting language affects what examples can be implemented.

5.1 Simple single token issuance

To create a simple currency SimpleCoin with a fixed supply of `s = 1000 SimpleCoins` tokens, we might try to use the simple policy script `Forges (s)` with a single forging transaction. Unfortunately, this is not sufficient as somebody else could submit another transaction forging another `1000 SimpleCoins`.

In other words, we need to ensure that there can only ever be a single transaction on the ledger that successfully forges SimpleCoin. We can achieve that by requiring that the forging transaction consumes a specific UTXO. As UTXOs are guaranteed to be (1) unique and (2) only be spent once, we are being guaranteed that the forging policy can only be used once to forge tokens. We can use the script:

```
simple_policy(o, v) = SpendsOutput(o) && Forges(v)
```

where `o` is an output that we create specifically for this purpose in a preceding setup transaction, and `v = s`.

5.2 Reflections of off-ledger assets

Many tokens are used to represent off-ledger assets on the ledger. An important example of this is *stablecoins*, which are a kind of token designed to be backed by some off-ledger asset. Other noteworthy examples of such assets include video game tokens, as well as service tokens (which represent service provider obligations).

A typical design for such a system is that a trusted party (the "issuer") is responsible for creation and destruction of the asset tokens on the ledger. The issuer is trusted to hold one of the backing off-ledger assets for every token that exists on the ledger, so the only role that the on-chain policy can play is to verify that the forging of the token is signed by the trusted issuer. To implement a forging policy that allows the issuer full control over the forging, as required by the above examples, we need only a clause that uses an m-out-of-n multi-signature scheme:

```
trusted_issuer(msig) = JustMSig(msig)
```

5.3 Vesting

A common desire is to release a supply of some asset on some schedule. Examples include vesting schemes for shares, and staged releases of newly minted tokens.

This seems tricky in our simple model: how is the forging policy supposed to know which tranches have already been released without some kind of global state which tracks them? However, this is a problem that we can solve with more tokens. We start building this policy by following the single issuer scheme, but we need to express more.

Given a specific output `o`, and two tranches of tokens `tranche1` and `tranche2` which should be released after `tick1` and `tick2`, we can write a forging policy such as:

```
vesting = SpendsOutput(o) && Forges({"tranche1" ↦ 1, "tranche2" ↦ 1})
       || TickAfter(tick1) && Forges(tranche1bundle) && Burns({"tranche1" ↦ 1})
       || TickAfter(tick2) && Forges(tranche2bundle) && Burns({"tranche2" ↦ 1})
```

This disjunction has three clauses:

- Once only, you may forge two unique tokens `tranche1` and `tranche2`.
- If you spend and burn `tranche1` and it is after `tick1`, then you may forge all the tokens in `tranche1bundle`.
- If you spend and burn `tranche2` and it is after `tick2`, then you may forge all the tokens in `tranche2bundle`.

By reifying the tranches as tokens, we ensure that they are unique and can be used precisely once. As a bonus, the tranche tokens are themselves tradeable.

5.4 Inventory tracker: tokens as state

We can use tokens to carry some data for us, or to represent state. A simple example is inventory tracking, where the inventory listing can only be modified by a set of trusted parties. To track inventory on-chain, we want to have a single output containing all of the tokens of an "inventory tracking" asset. If the trusted keys are represented by the multi-signature `msig`, the inventory tracker tokens should always be kept in a UTXO entry with the following output:

```
(hash(msig) , {hash(msig) ↦ {hats ↦ 3, swords ↦ 1, owls ↦ 2}})
```

The output and forging policy script prevents anyone but the trusted parties from modifying the output, but they are free to add or remove tokens to keep the inventory listing up to date. The inventory tracker is an example of an asset that should indefinitely be controlled by a script, and we enforce this condition in the forging script itself. Instead of just `msig`, we can use a script that requires checking both the multi-signature and that the inventory tracker asset is output to this very script:

```
inventory_tracker(msig) = JustMSig(msig) && AssetToAddress(_, _)
```

In this case, `inventory_tracker(msig)` is both the forging script and the output-locking script. The blank values supplied as arguments mean that the policy ID (and also the address) are both assumed to be the hash of the `inventory_tracker(msig)` script. Note that the evaluator is always passed the hash of the script being run (in addition to the script itself). So, for running output scripts, it is passed the hash of the output script, not the forging script hash.

Defined this way, our script is run at initial forge time, any time the inventory is updated, and any time it is spent. Each time it only validates if all the inventory tracker tokens in the transaction's outputs are always locked by this exact output script.

5.5 Non-fungible tokens

A common case is to want an asset group where *all* the tokens are non-fungible. A simple way to do this is to simply have a different asset policy for each token, each of which can only be run once by requiring a specific UTXO to be spent.

However, this is clumsy, and typically we want to have a set of non-fungible tokens all controlled by the same policy. We can do this with the `FreshTokens` clause. If the policy always asserts that the token names are hashes of data unique to the transaction and token, then the tokens will always be distinct.

5.6 Revocable permission

An example where we employ this dual-purpose nature of scripts is revocable permission. We will express permissions as a *credential token*.

The list of users (as a list of hashes of their public keys) in a credential token is composed by some central accreditation authority. Users usually trust this authority to have verified some real-life data, e.g. a KYC accreditation authority has checked off-chain that those it accredits meet some standard. Suppose now that exchanges are only willing to transfer funds to those that have proved that they are KYC-accredited. [5]

In this case, the accreditation authority could issue an asset that looks like

$$\{\texttt{KYC\_accr\_authority} \mapsto \{\texttt{accr\_key\_1} \mapsto 1, \texttt{accr\_key\_2} \mapsto 1, \texttt{accr\_key\_3} \mapsto 1\}\}$$

where the token names are the public keys of the accredited users. We would like to make sure that

- only the authority has the power to ever forge or burn tokens controlled by this policy, and it can do so at any time,
- all the users with listed keys are able to spend this asset as on-chain proof that they are KYC-accredited, and
- once a user is able to prove they have the credentials, they should be allowed to receive funds from an exchange.

We achieve this with a script of the following form:

```
credential_token(msig) = JustMSig(msig) && DoForge
                       || AssetToAddress(_) && Not DoForge && SignedByPIDToken(_)
```

Here, forges (i.e. updates to credential tokens) can only be done by the `msig` authority, but every user whose key hash is included in the token names can spend from this script, provided they return the asset to the same script. To make a script that only allows spending from it if the user doing so is on the list of key hashes in the credential token made by `msig`, we write

```
must_be_on_list(msig) = SpendsCur(credential_token(msig))
```

In our definition of the credential token, we have used all the strategies we discussed above to extend the expressivity of an FPS language. We are not yet using the UTXO model to its full potential, as we are just using the UTXO to store some information that cannot be traded. However, we could consider updating our credential token use policy to associate spending it with another action, such as adding a pay-per-use clause. Such a change really relies on the UTXO model.

[5] KYC stands for "know your customer", which is the process of verifying a customer's identity before allowing the customer to use a company's service

6 Related work

Waves. Waves [13] is an account-based multi-asset system, supporting its own smart contract language. In Waves, both accounts and assets themselves can be associated with contracts. In both cases, the association is made by adding the associated script to the account state (or the state of the account containing the asset). A script associated with an asset imposes conditions on the use of this asset, including minting and burning restrictions, as well as transfer restrictions.

Stellar. Stellar [10] is an account-based native multi-asset system geared towards tracking real-world item ownership via associated blockchain tokens. Stellar is optimised to allow a token issuer to maintain a level of control over the use of their token even once it changes hands. The Stellar ledger also features a distributed exchange listing, which is used to facilitate matching (by price) exchanges between different tokens.

Zilliqa. Zilliqa is an account-based platform with an approach to smart contract implementation similar to that of Ethereum [9]. The Zilliqa fungible and non-fungible tokens are designed in a way that mimics the ERC-20 and ERC-721 tokens, respectively. While this system is designed to be statically analysable, it does not offer new solutions to the problem of dependency on the global state.

DAML. DAML [5] is a smart contract language designed to be used on the DAML ledger model. The DAML ledger is a list of pairs of a transaction and the party that submitted it. A transaction is a list of actions, such as posting a new contract, engaging with an existing contract, etc. Interacting with a contract results in the creation of contracts that are the next steps of the original contract. This is the DAML approach to representing contract state.

Only the contracts with which a transaction interacts are relevant to validating it, which is similar to our approach of validation without global context. Note that this model does not have a base currency. Although this system does not have built-in multi-asset support, the transfer of any type of asset can be represented on the ledger. All transfers of funds are expressed in the form of a contract, wherein the party transferring the funds specifies the type of asset, the amount, and any other relevant details. Due to the design of the system to operate entirely by listing contracts on the ledger, each action, including a transfer of funds contract, requires consent. This is another significant way in which this is model is different from ours.

Bitcoin. Bitcoin popolarised UTXO ledgers, but has neither native nor non-native multi-asset support. The Bitcoin ledger model does not appear to have the accounting infrastructure or sufficiently expressive smart contracts for implementing multi-asset support in a generic way. There have been attempts to implement custom tokens using Lightning network channels [1], but these are layer-two solutions, and would not be tradeable across all accounts in a universal way.

Tezos. Tezos [7] is an account-based platform with its own smart contract language. It has been used to implement an ERC20-like fungible token standard (FA1.2), with a unified token standard in the works (see [11]). The custom tokens for both multi-asset standards are non-native, and thus have shortcomings similar to those of Ethereum token standards.

Nervos CKB. Nervos CKB [14] is a UTXO-inspired platform that operates on a broader notion of a Cell, rather than the usual output balance amount and address, as the value stored in an entry. A Cell entry can contain any type of data, including a native currency balance, or any type of code. This platform comes with a Turing-complete scripting language that can be used to define custom native tokens. There is, however, no dedicated accounting infrastructure to handle trading custom assets in a similar way as the base currency type.

6.1 General observations

Asset registries and distributed exchanges. The most obvious way to manage custom assets might be to add some kind of global *asset registry*, which associates a new asset group with its policy. Once we have an asset registry, this becomes a natural place to put other kinds of infrastructure that rely on global state associated with assets, such as decentralised exchanges.

However, our system provides us with a way to associate forging policies and the assets controlled by them *without* any global state. This is tantalising: while it might be convenient to have a structure that gives a full listing of all custom tokens in existence (e.g., to access credentials in a single location, or browse tokens offered for exchange), a stateless approach, such as ours, simplifies the ledger implementation in the concurrent and distributed environment of a blockchain. Introducing global state into our model would result in disrupted synchronisation (on which [4] relies to great effect to implement fast, optimistic settlement), as well as gating asset registration behind slow and costly state updates.

Hence, on balance we think it is better to have a stateless system, even if it relegates features like decentralised exchanges to be Layer 2 solutions.

Spending policies. Some platforms we discussed provide ways to express restrictions on the *transfer* of tokens, not just on their forging and burning. We refer to such restrictions as *spending policies*.

Unlike forging policies, spending polices are not native to our system. We have considered a number of approaches to adding them, but we have not found a solution that does not put an unsustainable burden on the *user* of such tokens.

Forging tokens requires a specific action by the user (providing and satisfying a forging policy script), but this action is always taken knowingly by a user who is specifically trying to forge those tokens. *Spending* tokens is, however, a completely generic operation that works over arbitrary bundles of tokens. Indeed, a putative virtue of our system is that custom tokens all look and behave uniformly. Spending policies break this property: a user might need to find and work out how to satisfy a potentially opaque spending policy. This makes such tokens extremely difficult to handle in a generic way, in particular smart contract systems are likely to stumble over them.

In a sense, this is the outcome that spending policies are designed to create: their purpose is to make transferring such assets harder. But this brings into doubt the point of tracking such assets on a distributed ledger of this kind. If you cannot transact with them normally, what have you gained by putting them on the ledger?

Viral scripts. One way to emulate spending policies in our system is to lock all the tokens with a particular script, and have said script ensure that when they are transferred they *remain* locked by the same script. We call such a script a "viral" script (since it spreads to any new outputs that are "infected" with the token).

This allows the conditions of the script to be enforced on every transaction that uses the tokens, but at significant costs. In particular such tokens can never be locked by a *different script*, which prevents usage in smart contracts, as well as preventing an output from containing tokens from two such viral asset groups (since both would require that *their* validator be applied to the output!).

References

1. Lightning Network multi-asset channels. https://github.com/lightningnetwork/lightning-rfc/pull/72 (2016)
2. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Meiklejohn, S., Sako, K. (eds.) Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10957, pp. 541–560. Springer (2018). https://doi.org/10.1007/978-3-662-58387-6_29, https://doi.org/10.1007/978-3-662-58387-6_29
3. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Peyton Jones, M., Wadler, P.: The Extended UTXO model. In: Proceedings of Trusted Smart Contracts 2020 (WTSC 20) (2020), to appear
4. Chakravarty, M.M., Coretti, S., Fitzi, M., Gazi, P., Kant, P., Kiayias, A., Russell, A.: Hydra: Fast isomorphic state channels. Tech. rep., Cryptology ePrint Archive, Report 2020/299 (2020), https://eprint.iacr.org/2020/299
5. DAML Team: DAML SDK documentation. https://docs.daml.com/ (2020)
6. Entriken, W., Shirley, D., Evans, J., Sachs, N.: ERC-721 non-fungible token standard. Ethereum Foundation (2018), https://eips.ethereum.org/EIPS/eip-721
7. Goodman, L.: Tezos—a self-amending crypto-ledger white paper (2014)
8. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper (October 2008)
9. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. arXiv preprint arXiv:1801.00687 (2018)
10. Stellar Development Foundation: Stellar Development Guides. https://solidity.readthedocs.io/ (2020)
11. Tezos Team: Digital Assets on Tezos. https://assets.tqtezos.com/docs/intro/ (2020)
12. Vogelsteller, F., Buterin, V.: ERC-20 token standard. Ethereum Foundation (Stiftung Ethereum), Zug, Switzerland (2015), https://eips.ethereum.org/EIPS/eip-20
13. Waves Team: Waves blockchain documentation. https://docs.wavesprotocol.org/ (2020)
14. Xie, J.: Nervos CKB: A Common Knowledge Base for Crypto-Economy. https://docs.daml.com/ (2018)
15. Zahnentferner, J.: An abstract model of UTxO-based cryptocurrencies with scripts. IACR Cryptology ePrint Archive **2018**, 469 (2018), https://eprint.iacr.org/2018/469