Babel Fees via Limited Liabilities

Manuel M. T. Chakravarty
manuel.chakravarty@iohk.io
IOHK

Nikos Karayannidis
nikos.Karagiannidis@iohk.io
IOHK

Aggelos Kiayias
akiayias@inf.ed.ac.uk
University of Edinburgh
IOHK

Michael Peyton Jones
michael.peyton-jones@iohk.io
IOHK

Polina Vinogradova
polina.vinogradova@iohk.io
IOHK

ABSTRACT

Custom currencies (ERC-20) on Ethereum are wildly popular, but they are second class to the primary currency Ether. Custom currencies are more complex and more expensive to handle than the primary currency as their accounting is not natively performed by the underlying ledger, but instead in user-defined contract code. Furthermore, and quite importantly, transaction fees can only be paid in Ether.

In this paper, we focus on being able to pay transaction fees in custom currencies. We achieve this by way of a mechanism permitting *short term liabilities* to pay transaction fees in conjunction with offers of custom currencies to compensate for those liabilities. This enables block producers to accept custom currencies in exchange for settling liabilities of transactions that they process.

We present formal ledger rules to handle liabilities together with the concept of *babel fees* to pay transaction fees in custom currencies. We also discuss how clients can determine what fees they have to pay, and we present a solution to the knapsack problem variant that block producers have to solve in the presence of babel fees to optimise their profits.

1 INTRODUCTION

Custom currencies, usually following the ERC-20 standard, are one of the most popular smart contracts deployed on the Ethereum blockchain. These currencies are however second class to the primary currency Ether. Custom tokens are not natively traded and accounted for by the Ethereum ledger; instead, part of the logic of an ERC-20 contract replicates this transfer and accounting functionality. The second class nature of custom tokens goes further, though: transaction processing and smart contract execution fees can only be paid in Ether — even by users who have got custom tokens worth thousands of dollars in their wallets.

The above two limitations and the disadvantages they introduce seem hard to circumvent. After all, it seems unavoidable that custom tokens must be issued by a smart contract and interacting with a smart contract requires fees in the primary currency. Still, recent work addressing the first limitation, showed that it can be tackled: by introducing *native custom tokens* (see e.g., [5]) it is possible to allow custom tokens to reuse the transfer and accounting logic that is already part of the underlying ledger. This is achieved without the need for a global registry or similar global structure via the concept of *token bundles* in combination with *token policy scripts* that control minting and burning of custom tokens. Nevertheless, even with native custom tokens, transaction fees still need to be paid in the primary currency of the underlying ledger.

To the best of our knowledge the only known technique to tackle the second limitation is in the context of Ethereum: the *Ethereum Gas Station Network (GSN)*[1]. The GSN attempts to work around this inability to pay fees with custom tokens by way of a layer-2 solution, where a network of relay servers accepts fee-less *meta-transactions* off-chain and submits them, with payment, to the Ethereum network. In return for this service, the GSN may accept payment in other denominations, such as custom tokens. Meta-transactions have the downside that in order to remove trust from intermediaries, custom infrastructure in *every* smart contract that wants to accept transactions via the GSN is needed. This has the serious downside that GSN users are only able to engage with the subset of the ledger state that explicitly acknowledges the GSN network. Beyond reducing the scope of GSN transactions, this introduces additional complexity on smart contract development including the fact that participating smart contracts must be preloaded with funds to pay the GSN intermediaries for their services.

Motivated by the above, we describe a solution that lifts this second limitation of custom tokens entirely and without requiring any modification to smart contract design. More specifically, we introduce the concept of *babel fees,* where fee payment is possible in any denomination that another party values sufficiently to pay the actual transaction fee in the primary currency. Our requirements for babel fees go beyond what GSN offers and are summarized as follows: (1) participants that create a babel fee transaction should be able to create a normal transaction, which will be included in the ledger exactly as is (i.e., no need for meta-transactions or specially crafted smart contract infrastructure) and (2) the protocol should be non-interactive in the sense that a single message from the creator of a transaction to the participant paying the fee in the primary currency should suffice. In other words, we want transaction creation and submission to be structurally the same for transactions with babel fees as for regular transactions.

Our implementation of babel fees is based on a novel ledger mechanism, which we call *limited liabilities.* These are negative token amounts (debt if you like) of strictly limited lifetime. Due to the limited lifetime of liabilities, we prevent any form of inflation (of the primary currency and of custom tokens).

Transactions paid for with babel fees simply pay their fees with primary currency obtained by way of a liability. This liability is combined with custom tokens offered to any party that is willing to cover the liability in exchange for receiving the custom tokens. In first instance, this allows block producers to process transactions with babel fees by combining them with a second fee paying

[1]https://docs.opengsn.org/

transaction that covers the liability and collects the offered custom tokens. More generally, more elaborate matching markets can be set up.

We describe native custom tokens and liabilities in the context of the UTXO ledger model. However, our contribution is more general and we will also sketch how it can be adapted for an account-based ledger.

In summary, this paper makes the following contributions:

- We introduce the concept of *limited liabilities* as a combination of negative values in multi-asset token bundles with batched transaction processing (Section 2).
- We introduce the concept of *babel fees* on the basis of limited liabilities as a means to pay transaction fees in tokens other than a ledger's primary currency (Section 2).
- We present formal ledger rules for an UTXO multi-asset ledger with limited liabilities (Section 3).
- We present a concrete spot market scheme for block producers to match babel fees (Section 4).
- We present a solution to the knapsack problem that block producers have to solve to maximise their profit in the presence of babel fees (Section 5).

We discuss related work in Section 6.

2 LIMITED LIABILITIES IN A MULTI-ASSET LEDGER

To realise babel fees by way of liabilities, we require a ledger that supports multiple *native* assets — that is, a number of tokens that are accounted for by the ledger's builtin accounting rules. Moreover, we assume that one of these native tokens is the *primary currency* of the ledger. The primary currency is used to pay transaction fees and may have other administrative functions, such as staking in a proof-of-stake system.

2.1 Native custom assets

To illustrate limited liabilities and Babel fees by way of a concrete ledger model, we base the technical part of the paper on the UTXO_{ma} ledger model [5] — an extension of Bitcoin's *unspent transaction output (UTXO)* model to natively support multiple assets.[2] For reference, we list the definitions of that ledger model in Appendix A, with the exception of the ledger rules that we cover in the following section. To set the stage for the ledger rules, we summarise the main points of the ledger model definitions in the following.

We consider a ledger l to be a list of transactions $[t_1, \ldots, t_n]$. Each of these transaction consists of a set of inputs is, a list of outputs os, a validity interval vi, a forge field $value_{forge}$, a set of asset policy scripts ps, and a set of signatures $sigs$. Overall, we have

$$
\begin{aligned}
t = (&inputs : is, outputs : os, validityInterval : vi,\\
&forge : value_{forge}, scripts : ps, sigs : sigs)
\end{aligned}
$$

The inputs refer to outputs of transactions that occur earlier on the ledger — we say that the inputs *spend* those outputs. The outputs, in turn, are pairs of addresses and values: $(addr : a, value : v)$, where *addr* is the hash of the public key of the key pair looking that output and *value* is the *token bundle* encoding the multi-asset value carried by the output. We don't discuss script-locked outputs in this paper, but they can be added exactly as described in [4].

Token bundles are, in essence, finite maps that map an asset ID to a quantity — i.e., to how many tokens of that asset are present in the bundle in question. The asset ID itself is a pair of a hash of the policy script defining the asset's monetary policy and a token name, but that level of detail has no relevance to the discussion at hand. Hence, for all examples, we will simply use a finite map of assets or tokens to quantities. For example, the token bundle

$$\{wBTC \mapsto 0.5, MyCoin \mapsto 5, nft \mapsto 1\}$$

contains 0.5 wrapped Bitcoin, five *MyCoin* and one *nft*.

The *forge* field in a transaction specifies a token bundle of minted (positive) and burned (negative) tokens. Each asset occurring in the forge field needs to have its associated policy script included in the set of policy scripts *ps*. Moreover, the *sigs* fields contains all signatures signing the transaction. These signatures need to be sufficient to unlock all outputs spent by the transaction's inputs *is*. Finally, the validity interval specifies a time frame (in an abstract unit of ticks that is dependent on the length of the ledger) in which the transaction may be admitted to the ledger.

We call the set of all outputs that (1) occur in a transaction in ledger l and (2) are not spent by any input of any transaction in l the ledger's UTXO set — it constitutes the ledger's state.

2.2 Limited liabilities

In a UTXO, the value for a specific token in a token bundle is always positive. In other words, the value component of a UTXO is always a composition of assets. It cannot include a debt or liability. We propose to *locally* change that.

2.2.1 Liabilities. We call a token in a token bundle that has a negative value a *liability.* In other words, for a token bundle *value* and asset a, if $value(a) < 0$, the bundle *value* includes an a-liability.

2.2.2 Transaction batches. In order to prevent liabilities appearing on the ledger proper, we do not allow the state of a fully valid ledger to contain UTXOs whose value includes a liability. We do, however, permit the addition of multiple transactions *at once* to a valid ledger, as long as the resulting ledger is again fully valid; i.e., it's UTXO set is again free of liabilities. We call a sequence of multiple transactions *ts*, which are being added to a ledger at once, a *transaction batch.* A transaction batch may include transaction outputs with liabilities as long as those liabilities are resolved by subsequent transactions in the same batch.

Consider the following batch of two transactions:

$$
\begin{aligned}
t_1 = (&inputs : is,\\
&outputs : [(addr : \emptyset, value : \{T_1 \mapsto -5, T_2 \mapsto 10\}),\\
&\qquad\qquad (addr : \kappa_1, value : \{T_1 \mapsto 5\})],\\
&validityInterval : vi, forge : 0, scripts : \{\}, sigs : sigs)\\
t_2 = (&inputs : \{(outputRef : (t_1, 0), key : \emptyset), i_{T_1}\},\\
&outputs : [(addr : \kappa_2, value : \{T_2 \mapsto 10\})],\\
&validityInterval : vi, forge : 0, scripts : \{\}, sigs : sigs')
\end{aligned}
$$

The first output of the first transaction, t_1, may be spend by anybody ($addr = \emptyset$). It contains both a liability of $-5T_1$ as well as an

[2]The multi-asset functionality of the Cardano blockchain is a concrete and in-production implementation of this model.

asset of $10T_2$. The second transaction t_2 spends that single output of t_1. The second transaction also has a second input i_{T_1}, which we may assume consumes an output containing $5T_1$, which is sufficient to cover the liability.

Overall, we are left with $5T_1$ exposed in t_1's second output and locked by κ_1 as well as $10T_2$, which t_2 exposes in its single output, locked with the key κ_2. Both transactions together take a fully valid ledger to a fully valid ledger as the liability is resolved within the transaction batch.

We have these two facts: (a) we have one transaction resolving the liability of another and (b) liabilities are not being permitted in the state of a fully valid ledger. Consequently, transaction batches with internal liabilities are either added to a ledger as a whole or all transactions in the batch are rejected together. This in turn implies that, in a concrete implementation of liabilities in a ledger on a blockchain, the transactions included in one batch always need to go into the same block. A single block, however, may contain several complete batches.

2.2.3 Pair production. The existence of liabilities in batches enables us to create transactions that temporarily (i.e., within the batch) inflate the supply of a currency. For example, consider a transaction t with two outputs o_1 and o_2, where o_1 contains $5000T$ and o_2 contains $-5000T$. While value is being preserved, we suddenly do have a huge amount of T at our disposal in o_1. In lose association with the somehow related phenomenon in quantum physics, we call this *pair production* — the creation of balancing positive and negative quantities out of nothing.

As all liabilities are confined to one batch of transactions only, this does not create any risk of inflation on the ledger. However, in some situations is can still be problematic as it may violate invariants that an asset's policy script tries to enforce. For example, imagine that T is a *role token* [4] — i.e., a non-fungible, unique token that we use to represent the capability to engage with a contract. In that case, we surely do not want to support the creation of additional instances of the role token, not even temporarily.

In other words, whether to permit pair production or not depends on the asset policy of the produced token. Hence, we will require in the formal ledger rules, discussed in Section 3, that transactions producing a token T always engage T's asset policy to validate the legitimacy of the pair production.

2.3 Babel fees

Now, we are finally in a position to explain the concrete mechanism underlying babel fees. The basic idea is simple. Let us assume a transaction t that attracts a fee of x C (where C is the ledger's primary currency), which we would like to pay in custom currency T. We add an additional *babel fee output* o_{babel} with a liability to t:

$$o_{babel} = \{C \mapsto -x, T \mapsto y\}$$

This output indicates that we are willing to pay y T to anybody who pays the x C in return. In other words, anybody who consumes o_{babel} will receive the y T, but will at the same time have to compensate the liability of $-x$ C. The two are indivisibly connected through the token bundle. In other words, we may view a token bundle that combines a liability with an asset as a representation of an atomic swap.

The transaction t can, due to the liability, never be included in the ledger all by itself. The liability $-x$ C does, however, make a surplus of x C available inside t to cover t's transaction fees.

To include t in the ledger, a counterparty to whom y T is worth at least x C needs to be found. Then the counterparty batches t with a fee paying transaction t_{fee} that consumes o_{babel}. In addition, t_{fee} will have to have another input from which it derives the x C together with its own transaction fee, all out of the counterparty's assets. The transaction t_{fee} puts the y T, by itself, into an unencumbered output for subsequent use by the counterparty. Finally, the counterparty combines t and t_{fee} into a transaction batch for inclusion into the ledger.

In Section 4, we will outline a concrete scheme based on Babel fees and fee paying transactions. In that scheme, block producing nodes act as fee paying counterparties for transactions that offer Babel fees in the form of custom tokens that are valuable to those block producer. They do so, on the fly, in the process of block production.

2.4 Other applications of limited liabilities

While our primary motivation for proposing liabilities limited by transaction batches are babel fees, the mechanism of limited liabilities is more broadly applicable.

2.4.1 Swaps. As discussed, we use liabilities in babel fees to form transaction outputs that represent atomic swaps — we call those *swap outputs*. We do this by including a liability (negative token value) together with an asset (positive token value). Whoever consumes such an output effectively swaps the tokens described by the liability for those constituting the asset.

UTXO ledgers do already support atomic swaps natively by way of transactions consuming two inputs carrying two different assets and swapping the keys under which these two assets are locked. Such a *swap transaction* does require both parties to sign that transaction (to authorise the consumption of each parties share in the swap). Moreover, both parties to the swap must be known at swap transaction creation time.

Liabilities enable us to break this monolithic, cooperative process into a non-interactive two-stage process. The first party creates a liability transaction consuming only it's own share of the swap and bundling it with a liability of the expected return in a swap output. The second party combines the liability transaction with a second transaction that resolves the liability and completes the swap. The second party can do that solely on the basis of the first transaction without any additional need to interact with the first party.

Our batch example from Section 2.2.2 demonstrates this pattern. The first output of t_1, namely

$$(addr : \emptyset, value : \{T_1 \mapsto -5, T_2 \mapsto 10\}),$$

is the swap output, offering to swap $10T_2$ for $5T_1$.

2.4.2 Service payments. Extending the concept of swaps from exchanging assets to exchanging assets for information. In the Extended UTXO model [4], which facilitates complex smart contracts on a UTXO ledger, transaction outputs also include a data component. This can, for example, be used to communicate information from an off-chain oracle. Liabilities included with such an output can serve as payment for consuming such an output with the data.

2.4.3 Indivisiblity. Transaction batches are different to signed transaction groups proposed for some ledgers, such as, for example, Algorand [2]. To create a signed transaction group, all component transactions need to be known and the group signed as a whole. If multiple component transactions are created by different parties, these parties need to cooperate to create the group transaction. The benefit of such a signed group is that it is indivisible.

The transaction batches that we propose are different. They are not inherently indivisible. For example, a batch comprising two transactions t_1 and t_2, where the latter consumes an output of the former, may be included in the ledger as a whole, but unless there is a liability involved in the output of t_1 consumed by t_2, we could also split the batch and simply submit t_1 on its own.

And even if t_1's output in includes a liability, while this prevents t_1 to go onto the ledger by itself, it still leaves the possibility of replacing t_2 by another transaction that resolves t_1's liability. In other words, while a liability may prevent a prefix of a batch to be fully valid on its own, it may not prevent swapping out a suffix of transactions.

Nevertheless, we can use spending conditions on outputs with liabilities to exert control over liability-resolving transactions in a batch. With pay-to-pubkey outputs, we can control who may create these transactions and within the Extended UTXO model [4], we can use script code to exert fine-grained control over these transactions.

2.5 Account-based ledgers

In this paper, we explain limited liabilities in the context of a UTXO ledger model, building on the native custom token extension for UTXO, called UTXO_{ma} introduced in [5]. We like to emphasize, though, that both the native custom token extension of UTXO_{ma} and the concept of limited liabilities from the present paper can equally well be applied to an account-based ledger — this might not come as a surprise, given that one accounting model can be translated into the other [18].

2.5.1 Native custom tokens. The core ideas of native custom tokens in UTXO_{ma} are (a) the generalisation of integral values in UTXOs to token bundles and (b) the use of policy-controlled forge fields. In the context of an account-based ledger, Point (a) translates to accounts that hold an entire bundle of tokens instead of just coins of a single currency. Just like in the UTXO_{ma} model, these token bundles can be represented as finitely-supported functions using the same group structure as UTXO_{ma} for value calculations. In a similar manner, transactions transferring value from one account to another now transfer entire token bundles, which get deducted from the source account and added to the target account, with the constraint that none of the bundle components of the source account may become negative as a result.

To realise Point (b), we extent transfer transactions with the same sort of forge field as in UTXO_{ma}. This includes the same conditions on the use of asset policies for all minted and burned assets.

2.5.2 Limited liabilities. We do require that transfer transactions based on token bundles cannot lead to negative token quantities in accounts. However, to represent liabilities, we do need a notion of account with a liability, albeit one of limited lifetime.

To this end, we introduce the concept of a *temporary anonymous accounts.* A temporary anonymous account is being identified with a hash derived from the transaction hash. Moreover, such an account is only available within one batch of transactions. In other words, just like a liability, it never gets added to the ledger state. As a logical consequence, a temporary anonymous account must have a balance of zero at the end of the batch in which it is active. This ensures that temporary anonymous accounts are never included in the ledger state.

In contrast to regular accounts in the ledger state, a temporary anonymous account may hold a token bundle, where one or more assets occur in a negative quantity. Those accounts, thus, serve the same purpose as transaction outputs with liabilities in UTXO_{ll}. Hence, they can form the basis for implementing babel fees for an account-based ledger.

Unlike the multi-input and multi-output UTXO transactions, account transactions typically have only one source and one destination. However, we can use temporary anonymous accounts as a sink as well as a source for the currency redistributed in a batch.

3 FORMAL LEDGER RULES FOR LIMITED LIABILITIES

In this section, we formalise the concept of limited liabilities by building on the UTXO_{ma} ledger; i.e., the UTXO ledger with custom native tokens as introduced in existing work [5]. To add support for limited liabilities, we modify the ledger rules in three ways:

(1) The original UTXO_{ma} rules are defining ledger validity by adding transactions to the ledger one by one. We extend this by including the ability to add transactions in *batches*; i.e., multiple transactions at once.
(2) We drop the unconditional per-transaction ban on negative values in transaction outputs and replace it by the weaker requirement that there remain no negative values at the fringe of a batch of transactions. In other words, liabilities are confined to occur inside a batch and are forced to be resolved internally in the batch where they are created.
(3) We amend the rules about the use of policy scripts such that the script of a token T is guaranteed to be run in every transaction that increases the *supply* of T.

In this context, the supply of a token T in a given transaction t is the amount of T that is available to be locked by outputs of t. If that supply is larger than the amount of T that is consumed by all inputs of t taken together, then we regard t as increasing the supply. This may be due to forging T or due to pair production (as discussed in Section 2.2.3).

3.1 Validity

In the original UTXO_{ma} ledger rules, we extend a ledger l with one transaction t at a time. In the UTXO_{ll} ledger rules (UTXO_{ma} with limited liabilities), we change that to add transactions in a two stage process that supports the addition of batches of transactions ts with internal liabilities:

$\mathsf{unspentTxOutputs} : \mathsf{Tx} \rightarrow \mathsf{Set}[\mathsf{OutputRef}]$			**– output references provided by a transaction**
$\mathsf{unspentTxOutputs}(t)$	$=$	$\{(\mathsf{txId}(t), 1), \ldots, (\mathsf{txId}(id), \lvert t.outputs \rvert)\}$	
$\mathsf{unspentOutputs} : \mathsf{Ledger} \rightarrow \mathsf{Set}[\mathsf{OutputRef}]$			**– a ledger's UTXO set**
$\mathsf{unspentOutputs}([\,])$	$=$	$\{\}$	
$\mathsf{unspentOutputs}(t :: l)$	$=$	$(\mathsf{unspentOutputs}(l) \setminus t.inputs) \cup \mathsf{unspentTxOutputs}(t)$	
$\mathsf{getSpentOutput} : \mathsf{Input} \times \mathsf{Ledger} \rightarrow \mathsf{Output}$			**– the outputs spent by the given set of transaction inputs**
$\mathsf{getSpentOutput}(i, l)$	$=$	$\mathsf{lookupTx}(l, i.outputRef.id).outputs[i.outputRef.index]$	
$\mathsf{policiesWithChange} : \mathit{Quantities} \times \mathit{Quantities} \rightarrow \mathsf{Set}[\mathsf{PolicyID}]$			**– policy IDs of assets whose amount varies**
$\mathsf{policiesWithChange}(val_1, val_2)$	$=$	$\{a.pid \mid a \in \mathsf{supp}(val_1 - val_2)\}$	
$\mathsf{changedSupply} : \mathsf{Tx} \times \mathsf{Ledger} \rightarrow \mathsf{Set}[\mathsf{PolicyID}]$			**– policy IDs whose supply changed in the transaction**
$\mathsf{changedSupply}(t, l)$	$=$	$\mathsf{policiesWithChange}(\sum_{o \in \mathsf{getSpentOutput}(t.inputs)} o.value^{+}, \sum_{o \in t.outputs} o.value^{+}) \cup$ $\mathsf{policiesWithChange}(\sum_{o \in \mathsf{getSpentOutput}(t.inputs)} o.value^{-}, \sum_{o \in t.outputs} o.value^{-})$ **where** $value^{+}(a) = \textbf{if } value(a) > 0 \textbf{ then } value(a) \textbf{ else } 0$; $value^{-}(a) = \textbf{if } value(a) < 0 \textbf{ then } value(a) \textbf{ else } 0$	

Figure 1: Auxiliary validation functions

(1) **The current tick is within the validity interval**

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

(2) ~~**All outputs have non-negative values**~~

~~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 locally double spent**

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

(6) **All inputs validate**

For all $i \in t.inputs,$ there exists $sig \in t.sigs, \mathsf{verify}(i.key, sig, \mathsf{txId}(t))$

(7) **Validator scripts match output addresses**

For all $i \in t.inputs,$ $\mathsf{keyAddr}(i.key) = \mathsf{getSpentOutput}(i, l).addr$

(8) **Forging**

♦ A transaction which changes the supply —i.e., $\mathsf{changedSupply}(t, l) \neq \{\}$— is only valid if either:

(a) the ledger l is empty (that is, if it is the initial transaction).

(b) ♦ for every policy ID $h \in \mathsf{changedSupply}(t, l)$, there exists $s \in t.scripts$ with $h = \mathsf{scriptAddr}(s)$.

(9) **All scripts validate**

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

Figure 2: Conditional validity of a transaction t in a ledger l permitting liabilities

(1) We modify the definition of the *validity* of a transaction t in a ledger l from $\mathrm{UTXO}_{\mathrm{ma}}$, such that it gives us *conditional validity* of t in l for $\mathrm{UTXO}_{\mathrm{ll}}$ as defined in Figure 2.
(2) We define validity of a batch of one or more transactions ts by way of the conditional validity of the individual $t \in ts$ together with the *batch validity* of ts in ledger l.

We describe the details of these two stages in the following.

3.2 Stage 1: conditional validity

Conditional validity in $\mathrm{UTXO}_{\mathrm{ll}}$ is defined very much like full validity in $\mathrm{UTXO}_{\mathrm{ma}}$. Figure 2 defines the conditions for transactions and ledgers to be conditionally valid, which are mutually dependent.

Definition 3.1 (Conditional validity of transactions and ledgers). A transaction $t \in \mathsf{Tx}$ is *conditionally valid* for a conditionally valid ledger $l \in \mathsf{Ledger}$ during tick $\mathsf{currentTick}$ if t abides by the conditional validity rules of Figure 2, using the auxiliary functions summarised in Figure 1.

A ledger $l \in \mathsf{Ledger}$, in turn, is conditionally valid if either l is empty or l is of the form $t :: l'$ with l' being a conditionally valid ledger and t being conditionally valid for l'.

Figure 2 highlights the two changes that we are making to the $\mathrm{UTXO}_{\mathrm{ma}}$ rules: firstly, we struck out Rule (2), and secondly, we changed Rule (8) in two places marked with ♦. The removal of Rule (2) is what permits liabilities in the first place. Outputs may now contain negative values and, if they do, the associated transaction is merely conditionally valid. Full validity is, then, conditional on the resolution of all liabilities by other transactions added to the ledger in the same batch.

Moreover, the change to Rule (8) ensures that transactions that change the supply of a token under a policy s with script address h do run the policy script s, regardless of whether the change in supply is due to a non-empty forge field $t.forge$ or due to pair production. In either case, the script is guaranteed an opportunity to validate that the increase in supply abides by the rules enforced by the token policy. In other words, transactions that contain supply changes that violate the associated token policy are guaranteed to be rejected.

3.2.1 *Changed supply.*

The change in supply is computed with the help of the function $\mathsf{changedSupply}(t, l)$ from Figure 1 that, for a given ledger l, determines all policy script hashes h that control an asset whose supply is changed by the transaction t. Such a change may be due to the minting or burning of assets in the transactions forge field $t.forge$ or it may be due to pair production, as discussed in Section 2.2.3. The function $\mathsf{changedSupply}$ spots supply changes by comparing the quantity of assets and asset liabilities in the inputs and outputs of a transaction. It uses the helper functions $value^{+}$ and $value^{-}$ to filter all positive (assets) and negative (liabilities), respectively, out of a token bundle.

3.2.2 *Script validation.*

Rule (8) uses the set of hashes of policy scripts computed by $\mathsf{changedSupply}$ to check that all the corresponding scripts are included in the $t.scripts$ field. The scripts in $t.scripts$ are exactly the scripts that Rule (9) executes.

Note that the primary currency of the ledger may require a special case in this rule. The total supply of the primary currency may be constant as part of the ledger implementation, and therefore its minting policy will always fail to validate, even in the case of producing and consuming transient debt. This may be addressed in (among others) one of the following ways: either modify the policy to specifically allow pair production of the primary currency, or modify this rule to not check the primary currency policy at all.

3.3 Stage 2: batch validity

For a ledger to be valid, we require that it is conditionally valid and that its state (i.e., the set of unspent outputs) does not contain any negative quantities.

Definition 3.2 (Ledger validity). A ledger l : Ledger is *(fully) valid* if l is conditionally valid and also,

$$\text{for all, } o \in \mathsf{unspentOutputs}(l), o.value \geq 0.$$

On that basis, we define the validity of a batch of transactions ts for a valid ledger l.

Definition 3.3 (Validity of a batch of transactions). A batch of transactions $ts : \mathsf{List}[\mathsf{Tx}]$ is *(fully) valid* for a valid ledger $l : \mathsf{Ledger}$ if $ts +\!\!+\, l$ is a fully valid ledger.

4 IMPLEMENTING BABEL FEES

In this section, we describe a concrete spot market, where users can exchange custom tokens via the babel fees mechanism described in Section 2.3. This spot market comprises a set of *sellers* $\mathbb{S} = \{s_1, s_2, ..., s_n\}$ and a set of *buyers*[3] $\mathbb{B} = \{b_1, b_2, ..., b_m\}$. Sellers sell bundles of custom tokens to buyers, who in return provide primary tokens to cover the fees incurred by the transactions submitted by the sellers to the network.

4.1 Babel offers

In this context, a transaction with a babel fee output (as per Section 2.3) essentially constitutes an offer — specifically, the offer to obtain a specified amount of custom tokens by paying the liability in primary tokens included in the babel fee output. We define such offers as follows.

Definition 4.1. We define a *babel offer* to be a tuple of the form:

$$BabelOffer \overset{\text{def}}{=} (Tx_{id}, TName, TAmount, Liability)$$

where Tx_{id} is a unique identifier of the transaction containing the babel fee output, $TName$ is a string corresponding to the name of a custom token, $TAmount$ is a positive integer $\in \mathbb{Z}^{+}$ corresponding to the amount of tokens offered and $Liability$ is a negative integer $\in \mathbb{Z}^{-}$ corresponding to the amount in primary tokens that has to be paid for obtaining the tokens.

Sellers produce such babel offers, which are then published to the network and are visible to all buyers.

4.2 Exchange rates

In our model, we assume that the spot market of babel offers operates in distinct rounds[4]. In every round, a buyer is selected from

[3]Buyers in this market are the *block issuers* of the blockchain.
[4]In practice this can be the block-issuing rounds.

the set $\mathbb{B}$ at random. The selected buyer has the opportunity to accept some of the outstanding offers by paying the corresponding liabilities. The rational buyer chooses the offers that maximise her utility function, which we elaborate in Section 5.

In order to help sellers to make attractive offers, we assume that every buyer i, $i = 1, 2, ..., m$ publishes a list $L_i[(T_j, XR_j)]$ of exchange rates XR_j for every exchangeable custom token T_j, $j = 1, ..., k$. The list of exchange rates from all buyers $BL[i] = L_i$, $i = 1, ..., m$ is available to all sellers $s \in \mathbb{S}$. Note that the buyer can set $XR_j = +\infty$ if they don't accept the token.

Given a specific babel offer $g = (t_g, (\mathtt{token_A}, \mathtt{amount_A}, \mathtt{liability_A}))$ offering an amount of a custom token $\mathtt{token_A}$, and assuming that there is only a single buyer b with a published exchange rate for $\mathtt{token_A}$ equal to XR_A, an attractive offer should adhere the inequality

$$\mathtt{amount_A} \geq |\mathtt{liability_A}| XR_A \qquad (1)$$

Naturally, an offer gets more attractive to the degree that excess tokens are offered over the minimum needed to meet the exchange rate for the liability, where the exchange rate XR_A expresses the $\frac{\mathtt{token_A}}{\mathtt{primary\ token}}$ rate.

4.3 Coverage

To generalise to the case where m possible buyers express an interest in $\mathtt{token_A}$, we need to consider the following question: how many $\mathtt{token_A}$ does a seller need to offer to ensure that $P\%$ buyers consider the offer attractive?

The seller has to choose the cheapest P_{th} percentile from the available exchange rates listed for $\mathtt{token_A}$, which by definition is satisfied by an effective exchange rate that is greater than $P\%$ of the published exchange rates. In other words, for the offer g from above to be attractive to $P\%$ of buyers, the seller needs to choose the amount for $\mathtt{token_A}$ as follows:

$$\mathtt{amount_A} \geq |\mathtt{liability_A}| \mathit{percentile}(P, \mathtt{token_A}, BL) \qquad (2)$$

where $\mathit{percentile}(P, \mathtt{token_A}, BL)$ is the lowest exchange rate for $\mathtt{token_A}$, thus that it is still greater than $P\%$ of the exchange rates listed for that token in the exchange rate table BL. In this case, we say that the offer g has *P% coverage.*

For example, assume a liability of 0.16 primary tokens and a set of 10 buyers with the following published exchange rates for $\mathtt{token_A}$, $BL_{\mathtt{token_A}} = \{1.63, 1.38, 3.00, 1.78, 2.00, 1.81\}$. If a seller wants to ensure that more than 70% of the buyers will consider her offer, she computes the 70th percentile of the exchange rates, which is 2.00. Thus, the seller knows that she has to offer at least $0.16 \times 2.00 = 0.32$ of $\mathtt{token_A}$.

4.4 Liveness

Consider a babel offer that is published to the network and assume that there is *at least one party* b_i (buyer) that is attracted by this offer. The interested party will then create a transaction batch t_{xb} (see Section 2.2.2) that covers the liability and will publish it to the network with the expectation that this will (eventually) be included into a block and be published in the ledger implemented by the blockchain. Therefore, it is crucial to ensure *censorship resilience* for our Babel offers and show that our spot market for Babel offers enjoys the property of *liveness* [9].

If b_i is selected as a block issuer, then she will include the transaction batch in the block she will create and thus liveness is preserved. However, if b_i is never selected as a block issuer (or is selected with a very low probability), then we must ensure the accepted offer will eventually be included into the blockchain. In the following analysis, we distinguish between two cases: a) The case where all buyers are acting rationally (but not maliciously) and b) the case where a percent of the buyers are controlled by a malicious adversary party.

4.4.1 Liveness in the presence of rational players. In this case, the aim of the players is to increase their income by collecting transaction fees and accepting Babel offers. Returning to our running example, lets assume that some other player b_j is selected to produce the next block. Then there are only two options: a) b_j is not attracted by the offer, or b) b_j is attracted by the offer.

In the former case, b_j will ignore the Babel offer, but she will not ignore the transaction batch t_{xb}. Since players are acting rationally and t_{xb} includes the appropriate transaction fees, then b_j or some other block issuer following b_j, will eventually select t_{xb} to be included in a block. The higher the fees the faster this will take place. So liveness is preserved in this case.

In the latter case, where b_j is indeed attracted by the Babel offer, then she can *front-run* and substitute t_{xb} with her own transaction batch t'_{xb} that will accept the specific Babel offer. This is a case where front-running transactions is a feature: it makes it feasible for block issuers to be paid in the tokens they prefer for their transaction processing services. However, liveness is also preserved in this case since the Babel offer will be accepted (not by b_i but by b_j) and be published to the blockchain.

4.4.2 Liveness in the presence of adversary players. Given that the aim of Babel fees is to facilitate transaction processing on a blockchain, we need to consider the adversarial case, where some of the block producers may conspire against the use of a particular token T. In our scheme for the implementation of Babel fees, block producers are also the buyers in the spot market for Babel offers. Hence, they may, in addition to ignoring offers with token T, also advertise unrealistically low exchange rates for T in BL (the table of buyers-accepted rates) to trick sellers into creating transactions that stand no chance of being processed. On the other hand, we do assume that the processing of transactions with Babel fees, just like directly payed transaction, is generally in the interest of the network. Hence, honest block producers will advertise a rate at which they will in fact process transactions with Babel fees if they are offered any.

In our analysis, we require the ratio $\frac{t}{m-t} \leq 1 - \delta$, where t is the number of parties controlled by the adversary and m is the total number of parties and $\delta \in (0, 1)$. This is an *honest majority assumption* if, without loss of generality, one assumes that all parties command the same amount of power, cf. [9]. Assuming that this ratio also holds for the entries in BL corresponding to published rates for a token T, then as long as the seller includes a sufficient amount of T tokens, such that the offer has 50% coverage, the seller's transaction will attract *at least one* honest party. This honest party will then issue a transaction batch t_{xb} accepting the offer.

In the following, we show that t_{xb} will eventually be published on the ledger and thus liveness is preserved. Assume that t_{xb} is issued at round r. By the *chain growth* property of the chain [9],we know that the chain adopted by any honest party will keep growing. After s rounds there will be a growth of at least τs blocks, where τ is the chain velocity parameter of the chain growth property. Intuitively, τ equals the probability that an honest party is selected as a block issuer, which in our case translates to $\tau = \frac{m-t}{m}$. So after s rounds (starting from r) an honest party's chain will have $\frac{m-t}{t}s$ new blocks. By the *chain quality* property of the chain [9], we know that the ratio of *honest blocks* (i.e., blocks produced by honest parties) in this chain will be at least[5] $\frac{m-t}{t}s\mu$, where $\mu \in (0, 1)$ is the honest block proportion parameter of the chain quality property. We know that if an honest block exists, then t_{xb} will be included, or a new t'_{xb} will be included instead accepting the Babel offer by the honest block issuer. So we require that $\frac{m-t}{t}s\mu \geq 1$, which means:

$$s \geq \frac{1}{\mu}\frac{t}{m-t} \tag{3}$$

Due to the chain growth property of the chain, this honest block that will include t_{xb} (or t'_{xb}) will be buried under k blocks, where k is the number of blocks for the *common prefix* property of the blockchain [9]. Then it will be reported in the ledger. Therefore, even in the presence of an adversary controlling t/m of the total hashing power, if honest majority holds and a Babel offer attracts at least one honest party, the accepted offer will be (eventually) published in the blockchain and thus liveness is preserved.

Just like transaction fees in Bitcoin, or exchange rates in fiat currencies, the published exchange rates BL are driven by market forces. We assume that in each round the BL array is updated with values that reflect the desirability of each custom token by each buyer in the current moment. Therefore, the seller (i.e., user running a wallet) does not need to know anything apart from the published exchange rates from the buyers nor do they need to perform advanced calculations — the simple percentile calculation described above is sufficient to create an attractive offer.

On the other hand, buyers need to solve a more difficult problem. In each round where a block producer is selected, they need to assemble a block of transactions (conventional ones, or babel offers) that maximises their utility. This problem is tackled in the next section.

5 TRANSACTION SELECTION FOR BLOCK ISSUERS

A block issuer constructs a block of transactions by choosing from a set of available transactions called the *mempool*. A rational block issuer tries to maximize her utility. In our case, we assume that this utility is a value, corresponding to the amount of primary currency earned by this block. These earnings come from the transaction fees paid either in primary currency or custom tokens. Hence we assume the existence of a utility function of the form: *utility* :: $CandidateBlock \longrightarrow Value$, where *CandidateBlock* is a list of transactions $CandidateBlock \overset{\text{def}}{=} List[CandidateTransaction]$ and *Value* is an amount $\in \mathbb{Z}^+$ of primary currency at the lowest denomination.

[5]Note that the number of honest blocks can be less than $\frac{m-t}{t}s$, because the adversary can employ attacks such as *selfish mining* that can eliminate honest blocks [9]

5.1 The value of Babel offers

A candidate transaction residing in the mempool and waiting to be included in a block can be either a (single) transaction or a transaction batch (see Section 2.2.2). In the following, we define the concept of a *candidate transaction*:

Definition 5.1. A candidate transaction residing in the mempool is defined as quadruple:

$$CandidateTransaction \overset{\text{def}}{=} (Tx_{id}, Value, Liability, Size)$$

where Tx_{id} is a unique identifier of a transaction (or a transaction batch) in the mempool, *Value* for the case of transactions corresponds to the transaction fees expressed in the primary currency, while for the case of transaction batches, it corresponds to the total value of the obtained custom tokens expressed as an amount in the primary currency. In the case of transaction batches, $Liability \in \mathbb{Z}^-$ is the amount expressed in the primary currency that has to be paid for covering this liability. In the case of transactions, it equals zero. Finally, *Size* is the total size of the transaction, or the transaction batch as a whole, expressed in bytes.

We assume the existence of a function that can transform a Babel offer (Definition 4.1) into a candidate transaction batch:

$$batchVal :: BabelOffer \longrightarrow CandidateTransaction$$

We need this function in order to be able to express the value of the obtained custom tokens in primary currency, so that Babel offers are comparable to the transaction fees of conventional transactions. Any such conversion function might be chosen by the block issuer based on her business logic of how to evaluate a specific offer. Indicatively, we propose the following formula for calculating the primary currency *Value* (of Definition 5.1) of a babel offer:

$$Value = \sum_{\forall token \in BabelOffer} TAmount \frac{nominalVal}{|Liability\ per\ token|} nominalVal = \sum_{\forall token \in BabelOffer} \frac{(TAmount \times nominalVal)^2}{|Liability|}$$

The nominal value of the token (*nominalVal*) is essentially the current $\frac{primary\ currency}{custom\ token}$ rate; i.e., it expresses what amount of primary currency one custom token is worth. Therefore, if the exchange rate between a custom token T and the primary currency A is $3:1$, then $\texttt{nominalVal} = 0.33A$. Of course, this rate is dynamic and it is determined by market forces just like with fiat currencies and Bitcoin fees. We assume that this information is available to the block producer, when they need to select candidate transactions from the mempool to include in a new block. In fact, block issuers can publish exchange rates for specific tokens they consider acceptable (as discussed in see Section 4). Intuitively, the higher the nominal value, the more valuable the token is to the block issuer.

Hence, whenever a block issuer tries to assemble a block they face the following optimization problem:

Definition 5.2. The *transaction selection problem*

$$TxSelection(n, S_B, M, R)$$

is the problem of filling a candidate block of size S_B, with a subset $B_n \subseteq M$ of n available candidate transactions $M = \{tx_1, tx_2, ..., tx_n\}$, where we use $B_n \subseteq \{1, 2, .., n\}$, without spending more than a reserve R of available primary currency on liabilities, in such a way that $utility(B_n) \geq utility(B'_n)$ $\forall$ block $B'_n \subseteq M$. Every candidate transaction $tx_i = (i, v_i, l_i, s_i)$, for $i = 1, ..., n$ is defined according to Definition 5.1 and has a fixed liability l_i and size s_i in bytes. We assume that the value of a candidate transaction that corresponds to a Babel offer is not fixed; instead, it decreases (just as its desirability) as we select candidate transactions offering the same custom token for the block. Thus, the value v_i of a candidate transaction is expressed as a function of what has already been selected for the block, $v_i(B_{i-1}) : CandidateBlock \longrightarrow Value$, where $B_{i-1} \subseteq \{1, 2, ..., i-1\}$ and $v_i(\emptyset) = v_{io}$ is the initial value of the offer and $0 \leq v_i(B_{i-1}) \leq v_{oi}$. Finally, the `utility` function that we want to maximize is defined as $utility = \sum_{i \in B_n} v_i(B_{i-1})$, where B_{i-1} is the solution to the $TxSelection(i-1, S_B - \sum_{j=i}^{n} s_j, M - \{i, ..., n\}, R - \sum_{j=i}^{n} l_j)$ problem.

5.2 Dynamic programming

Algorithm 1 presents an optimal solution to the transaction selection problem. It is a variation of the dynamic programming solution to the 0-1 knapsack problem [8]. It is important to note that we want conventional transactions and transaction batch offers to be comparable *only* with respect to the value offered and their size. We do not want to view liability as another constraint to the knapsack problem, because this would favor zero liability candidate transactions (i.e., conventional transactions) over Babel offers. The liability aspect of the offer has already been considered in the value calculation of the the conversion function from a `BabelOffer` to a `CandidateTransaction`, as shown in the indicative conversion formula above.

Initially, we order the candidate transactions of M in descending order of their (initial) value per size ratio $v_{io}/s_i, i = 1, 2, ...n$. We maintain an array $U[i], i = 1, 2, ...n$. Each entry $U[i]$ is a list of tuples of the form (t_s, t_v, r, b). A tuple (t_s, t_v, r, b) in the list $U[i]$ indicates that there is a block B assembled from the first i candidate transactions that uses space exactly $t_s \leq S_B$, has a total value exactly $utility(B) = t_v \leq \sum_{i=1}^{n} v_{oi}$, has a residual amount of primary currency to be spent on liabilities exactly $r \leq R$ and has a *participation bit* b indicating if transaction i is included in B, or not.

This list does not contain all possible such tuples, but instead keeps track of only the most efficient ones. To do this, we introduce the notion of one tuple *dominating* another one; a tuple (t_s, t_v, r, b) dominates another tuple (t'_s, t'_v, r', b'), if $t_s \leq t'_s$ and $t_v \geq t'_v$; that is, the solution indicated by the tuple (t_s, t_v, r, b) uses no more space than (t'_s, t'_v, r', b'), but has at least as much value. Note that domination is a transitive property; that is, if (t_s, t_v, r, b) dominates (t'_s, t'_v, r', b') and (t'_s, t'_v, r', b') dominates (t''_s, t''_v, r'', b''), then (t_s, t_v, r, b) also dominates (t''_s, t''_v, r'', b''). We will ensure that in any list, no tuple dominates another one; this means that we can assume each list $U[i]$ is of the form $[(t_{s1}, t_{v1}, r_1, b_1), ..., (t_{sk}, t_{vk}, r_k, b_k)]$ with $t_{s1} < t_{s2} < ... < t_{sk}$ and $t_{v1} < t_{v2} < ... < t_{vk}$. Since every list $U[i], i = 1, 2, ..., n$ does not include dominating tuples and also the sizes of the transactions are integers and so are their values, then we can see that the maximum length of such a list is $min(S_B+1, V_o+1)$, where $V_o = \sum_{i=1}^{n} v_{oi}$.

Algorithm 1 starts out with the initialization of list $U[1]$ (line 2) and then iterates through all $n-1$ transactions (lines 3-10). In each iteration j, we initially set $U[j] \longleftarrow U[j-1]$ after turning off the participation bit in all tuples (lines 4-5). Then for each tuple $(t_s, t_v, r, b) \in U[j-1]$, we also add the tuple $(t_s+s_j, t_v+v_j(B_{j-1}), r-l_j, 1)$ to the list, if $t_s + s_j \leq S_B \land r - l_j \geq R$; that is, if by adding transaction j to the corresponding subset, we do not surpass the total available size S_B and do not deplete our reserve R for liabilities (lines 6-9). Note that the value of transaction j at this point is determined by the contents of the corresponding block B_{j-1} through the function call $v_j(B_{j-1})$. To this end, in lines 14-22 we provide a function that returns the block corresponding to a specific tuple. We finally remove from $U[j]$ all dominated tuples by sorting the list with respect to their space component, retaining the best value for each space total possible, and removing any larger space total that does not have a corresponding larger value (line 10). We return the maximum total value from the list $U[n]$ along with the corresponding block B_n (lines 11-13).

5.3 Optimality

Next we argue that Algorithm 1 returns the optimal solution.

Theorem 5.3. *Algorithm 1 correctly computes the optimal value for the transaction selection problem.*

Proof. We will prove by induction that for any feasible block of transactions $B \subseteq \{1, 2, ..., j\}$ corresponding to the tuple (t_s, t_v, r, b), $t_s \leq S_B \land r \leq R$, list $U[j]$ will always include some tuple (t'_s, t'_v, r', b'), $t'_s \leq S_B \land r' \leq R$ that dominates (t_s, t_v, r, b).

For $j = 1$, we have $U[1] = \{(0, 0, R, 0), (s_1, v_1, R - l_1, 1)\}$ and the claim for any $B \subseteq \{1\}$ trivially holds.

We assume that the claim holds for the list $U[j-1]$.

Let $B \subseteq \{1, 2, ..., j\}$ be any block and (t_s, t_v, r, b) be the corresponding tuple, $t_s \leq S_B \land r \leq R$. We have two options: a) $b == 0$ and b) $b == 1$. In other words, transaction j is not part of B, or it is part of B.

If j is not part of B, then by induction hypothesis we know that in the list $U[j-1]$ there will be some tuple (t''_s, t''_v, r'', b'') that dominates (t_s, t_v, r, b). Remember that Algorithm 1 first sets $U[j] \longleftarrow U[j-1]$ and then removes all dominating pairs. Thus there will be some tuple in $U[j]$ that dominates (t''_s, t''_v, r'', b'') and then by transitivity of domination will also dominate (t_s, t_v, r, b). Thus there will be some tuple in $U[j]$ that dominates (t_s, t_v, r, b).

If j is part of B, then we consider block $B' = B - \{j\}$. By induction hypothesis again, there will be some tuple (t''_s, t''_v, r'', b'') in list $U[j-1]$ that dominates tuple

$$(\sum_{k \in B'} s_k, \sum_{k \in B'} v_k(B'_{k-1}), r_k, b_k)$$

Which means that $t''_s \leq \sum_{k \in B'} s_k$ and $t''_v \geq \sum_{k \in B'} v_k(B'_{k-1})$. Then, Algorithm 1 will add transaction j to the tuple (t''_s, t''_v, r'', b'') and add tuple $(t''_s + s_j, t''_v + v_j(B'), r'' - l_j, 1)$ to the list $U[j]$. But then we have $t''_s + s_j \leq \sum_{k \in B'} s_k + s_j = t_s$ and $t''_v + v_j(B') \geq \sum_{k \in B'} v_k(B'_{k-1}) + v_j(B') = t_v$. Thus there will be some tuple in $U[j]$ that dominates (t_s, t_v, r, b). □

Algorithm 1: Transaction selection algorithm for a block (Optimal Solution).

```
Input: A set $M$ of candidate transactions $M = \{tx_1, tx_2, ..., tx_n\}$, where
       $tx_i = (i, v_i(B_{i-1}), l_i, s_i)$ for $i = 1, ..., n$ according to definition 5.1
Input: An amount of primary currency available for covering liabilities, called the
       reserve $R$.
Input: An available block size $S_B$
Input: A utility function $util = \sum_{i \in B_n} v_i(B_{i-1})$
Output: $(B, util(B), res)$: A candidate block $B \subseteq M$ such that
        $util(B) > util(B') \forall B' \subseteq M$, the value of this block ($util(B)$) and a residual
        amount $res$ from the reserve $R$ such that $res \geq 0$
/* Assume array U[i]: Array[List[(Size, Value, Liability, Bit)]],
   $i = 1, ...n$                                                        */
order transactions in $M$ in descending order of $v_{io}/s_i$, $i = 1, 2..., n$
$U[1] \longleftarrow [(0, 0, R, 0), (s_1, v_{1o}, R - l_1, 1)]$
for $j = 2$ to $n$ do
    $baseList \longleftarrow$ copy list $U[j-1]$ with zero participation bits for all tuples
    $U[j] \leftarrow$ baseList
    foreach $(t_s, t_v, r, b) \in$ baseList do
        if $t_s + s_j \leq S_B \wedge r - l_j \geq R$ then
            $B_{j-1} \longleftarrow getBlock(U, j-1, t_s)$
            Add tuple $(t_s + s_j, t_v + v_j(B_{j-1}), r - l_j, 1)$ to $U[j]$
    Remove dominating pairs from list $U[j]$
$(S_{final}, V_{max}, residual, b) \longleftarrow max_{(s,v,r) \in U[n]}(v)$
$B_n \longleftarrow getBlock(U, n, S_{final})$
return $(B_n, V_{max}$, residual)
// ——————————————
getBlock(U: Array[List[(Size, Value, Liability, Bit)]], n:$Tx_{id}$, $t_{sn}$: Size) return
  CandidateBlock
$B \longleftarrow []$
$t_s \longleftarrow t_{sn}$
for $i = n$ down to 1 do
    $(t_{si}, t_{vi}, r_i, b_i) \longleftarrow getTuple(U[i], t_s)$
    if $b_i == 1$ then
        $B \longleftarrow i : B$ // ":" is list construction
        $t_s \longleftarrow t_s - t_{si}$
return $B$
```

5.4 Polynomial approximation

Since we iterate through all available n transactions and in each iteration we process a list of length $min(S_B + 1, V_o + 1)$, where $V_o = \sum_{i=1}^{n} v_{oi}$, we can see that algorithm 1 takes $O(n \, min(S_B, V_o))$ time. This is not a polynomial-time algorithm, since we assume that all input numbers are encoded in binary; thus, the size of the input number S_B is essentially $log_2 S_B$, and so the running time $O(nS_B)$ is exponential in the size of the input number S_B, not polynomial. Based on the intuition that if the maximum value V_o was bounded by a polynomial in n, the running time will indeed be a polynomial in the input size, we now propose an approximation algorithm for the transaction selection problem that runs in polynomial time and is based on a well-known fully polynomial approximation scheme of the 0-1 knapsack problem [12].

The basic intuition of the approximation algorithm is that if we round the (integer) values of the candidate transactions to $v'_i(B_{i-1}) = \lfloor v_i(B_{i-1})/\mu \rfloor$, where $0 \leq v'_i(B_{i-1}) \leq \lfloor v_{io}/\mu \rfloor = v'_{io}$ and run Algorithm 1 with values v'_i instead of v_i, then by an appropriate selection of μ, we could bound the maximum value $V'_o = \sum_{i=1}^{n} v'_{io}$ by a polynomial in n and return a solution that is at least $(1 - \epsilon)$ times the value of the optimal solution (OPT). In particular, if we choose $\mu = \epsilon v_{omax}/n$, where v_{omax} is the maximum value of a transaction; that is, $v_{omax} = max_{i \in M}(v_{oi})$. Then, for the total maximum value V'_o, we have $V'_o = \sum_{i=1}^{n} v'_{io} = \sum_{i=1}^{n} \lfloor \frac{v_{io}}{\epsilon v_{omax}/n} \rfloor = O(n^2/\epsilon)$. Thus, the running time of the algorithm is $O(n \, min(S_B, V'_o)) = O(n^3/\epsilon)$ and is bounded by a polynomial in $1/\epsilon$. In Algorithm 2, we describe our approximate algorithm for the transaction selection problem. We can now prove that this algorithm returns a solution whose value is at least $(1 - \epsilon)$ times the value of the optimal solution.

Theorem 5.4. *Algorithm 2 provides a solution which is at least $(1 - \epsilon)$ times the value of OPT.*

Proof. Let B be the block returned from Algorithm 2 for the problem TxSelection(n, S_B, M, R). Let B_{opt} be the optimal solution for this problem. We want to show that $utility(B) \geq (1 - \epsilon) utility(B_{opt}) = (1 - \epsilon)OPT$. Certainly $v_{omax} \leq OPT$, since one possible solution is to put the most valuable transaction in a block by itself. By the definition of v'_i we know that $\mu v'_i \leq v_i \leq (\mu + 1)v'_i$, so that $\mu v'_i \geq v_i - \mu$. These inequalities along with the fact that B is the optimal solution for the problem TxSelection(n, S_B, M', R) and thus $\sum_{i \in B} v'_i(B_{i-1}) \geq \sum_{i \in B_{opt}} v'_i(B_{i-1})$, we have the following:

$$\begin{aligned}
utility(B) &= \sum_{i \in B} v_i(B_{i-1}) \\
&\geq \mu \sum_{i \in B} v'_i(B_{i-1}) \\
&\geq \mu \sum_{i \in B_{opt}} v'_i(B_{i-1}) \\
&\geq \sum_{i \in B_{opt}} v_i(B_{i-1}) - |B_{opt}|\mu \\
&\geq \sum_{i \in B_{opt}} v_i(B_{i-1}) - n\mu \\
&= \sum_{i \in B_{opt}} v_i(B_{i-1}) - \epsilon v_{omax} \\
&= utility(B_{opt}) - \epsilon v_{omax} \\
&= OPT - \epsilon v_{omax} \\
&\geq OPT - \epsilon OPT = (1 - \epsilon)OPT
\end{aligned}$$

□

Algorithm 2: Transaction selection algorithm for a block (Approximate Solution).

```
Input: A set $M$ of candidate transactions $M = \{tx_1, tx_2, ..., tx_n\}$, where
       $tx_i = (i, v_i(B_{i-1}), l_i, s_i)$ for $i = 1, ..., n$ according to definition 5.1
Input: An amount of primary currency available for covering liabilities, called the
       reserve $R$.
Input: An available block size $S_B$
Input: A utility function $util = \sum_{i \in B_n} v_i(B_{i-1})$
Input: The acceptable error $\epsilon$ from the optimal solution, where $0 < \epsilon < 1$
Output: $(B, util(B), res)$: A candidate block $B$ such that
        $util(B) > util(B') \forall B' \subseteq M$, the value of this block ($util(B)$) and a residual
        amount $res$ from the reserve $R$ such that $res \geq 0$
$v_o max \longleftarrow max_{i \in M}(v_{oi})$
$\mu \longleftarrow \epsilon v_{omax}/n$
$v'_i(B_{i-1}) \longleftarrow \lfloor v_i(B_{i-1})/\mu \rfloor$ for $i = 1, 2, ..., n$
run algorithm 1 for the problem instance TxSelection$(n, S_B, M', R)$, where
  $M' = \{tx'_1, tx'_2, ..., tx'_n\}$, and $tx'_i = (i, v'_i(B_{i-1}), l_i, s_i)$ for $i = 1, ..., n$
```

6 RELATED WORK

Ultimately, our Babel fee mechanism is made possible by swap outputs, which in turn are based on limited-lifetime liabilities. Babel fees require swaps that once being proposed (as part of a complete transaction) can be resolved unilaterally by the second party accepting the swap as discussed in Section 2.4.1.

In existing systems, atomic swaps (which may be used to pay for fees) often go via an exchange, including for Ethereum ERC-20 tokens [14] and Waves' custom natives [17], including multi-blockchain exchanges based on atomic swaps [11, 13], which come with varying degrees of decentralisation. Alternatively, atomic swaps may swap assets across chains [10].

Our limited-lifetime liabilities are a sort of loan, but one that is resolved before it is even recorded on the ledger. There is also work on ledger-based loans [3, 16], but this leads to rather different challenges and mechanisms.

All these mechanisms, while having some capacity to simulate non-primary currency fee payments, are usually a combination of off-chain solutions and layer-2 (implemented via smart contracts on the participating blockchains), and are quite different from the single-chain, ledger-integrated proposal we give here. The following are several designs which solve problems which are more similar to our proposal.

6.1 Ethereum's Gas Station Network

As we mentioned previously in this paper, Ethereum supports fee payment in non-primary currencies via its Gas Station Network (GSN) [1]. The gas station infrastructure consists of

- a network of nodes listening for meta-transactions (transaction-like requests to cover transaction fees), which turn these requests into complete transactions, with fees covered by the relay node, and
- an interface that contracts must implement in order for the relay nodes to use this contract's funds to subsidize the transaction fees.

This infrastructure consists of many moving parts working together, including smart contracts, relays, relay hubs, and communication on a network separate from the mainchain network. Only GSN-enabled contracts can cover transaction fees. Our proposal does not require any changes to existing smart contracts, and does not require meta-transactions to be disseminated on a separate network, since they are already fully-formed and signed transactions.

In addition, unlike the GSN, there is no further action required from the user after submitting a Babel-fee transaction. The design of the GSN allows for the possibility that an incorrect transaction is submitted by a relay node in response to a sender's fee-coverage request. The onus is on the sender to monitor the chain, and request punitive measures to be taken against an offending relay. There are other verifications necessary to participate in GSN. Submitting transactions with liabilities has no potential of unexpected consequences (whether they are included in the ledger or not).

The GSN requires participating fee-covering contracts to prepay for the fee amounts they intend to cover. This approach involves additional maintenance, monitoring, and communication. The contract may specify the tokens it accepts in exchange for covering fees, but the extra step of posting and updating the contracts on-chain is less flexible and has more steps (including submitting potentially costly transactions) than our strategy. Recall that in our design, we propose to automate the process of any user getting a transaction with an exchange offer, accepting or rejecting the offer based on maximizing the value they are getting by engaging in the offer, then submitting the batch containing the swap transactions to the chain.

With Babel fees, a user may use a higher-than-minimum fee or exchange bid to increase the chances of their transaction to be accepted sooner. The exchange offers made via GSN-enable smart contracts, as well as the fee amounts the contract is willing to cover, are all fixed.

6.2 Algorand

Algorand in an account-based cryptocurrency which supports custom native tokens. It provides users with a way to perform atomic transfers (see [2]). An atomic transfer requires combining unsigned transactions into a single group transaction, which must then be signed by each of the participants of each of the transactions included. This design allows users to perform, in particular, atomic swaps, which might be used to pay fees in non-primary currencies.

As with our design, the transactions get included into the ledger in batches. Unlike the mechanism we propose, however, incomplete transactions cannot be sent off to be included in the ledger without any further involvement of the transaction author. This interactive protocol specification ensures that batches cannot be taken apart and completed using other transactions. This may be an advantage in certain cases over a batch that is combined in a loose, easily-decomposable way, but this behaviour can also be implemented in the system we have presented. Moreover, an interactive protocol for building group transaction requires additional communication, which is, in this case, reliant on off-chain communication.

6.3 Debt Representation in UTXO Blockchains

There are similarities between the debt representation proposal presented in [7] and the mechanism we propose, the main one being the idea of representing debt as special inputs on an UTXO ledger. The debt model presented in that paper allows debt to be recorded in a persistent way on the ledger. In contrast, we propose liabilities that only appear mid-batch and are guaranteed to be resolved before they are even recorded on the ledger.

As we prevent liabilities to ever enter the ledger state, we side step the main issues discussed in [7], including the need for managing permissions for issuing debt on the ledger, and therefore also for the trust users may be obligated to place in the debt issuer, and vice-versa. The possibility of unresolved debt remaining on the ledger (and therefore inflation) is a concern that needs to be taken seriously in this case.

Debt recorded on the ledger state (and outside a transaction batch) enables functionality that we cannot support with limited liabilities. Moreover, if a debt-creating transaction is complete and ready to be applied to the ledger, all nodes are able to explicitly determine the validity of this transaction. This way, these transaction can be relayed by the existing network, without any special consideration for their potential to be included in a batch, and by who.

Another key difference between the two proposals is that ours assumes an underlying multi-asset ledger, so that the debt-outputs have another major interpretation — they also serve as offers for custom token fee coverage, as well as swaps. Finally, the ledger we propose treats debt outputs and inputs in a uniform way, rather than in terms of special debt transactions and debt pools, which are needed to mitigate the hazards of special cases.

6.4 Stellar DEX

A commont blockchain solution to providing swap functionality (and therefore, custom token fee payment) is a distributed exchange (DEX). The Stellar system (see [15]), supports a native, ledger-implemented DEX.

In the Stellar DEX, offers posted by users are stored on the ledger. A transaction may attempt an exchange of any asset for any other asset, and will fail if this exchange is not offered. This approach requires submitting transactions to manage a user's on-chain offers, and also requires all exchanges to be exact — which means no overpaying is possible to get one's bid selected. A transaction may attempt to exchange assets that are not explicitly listed as offers in exchange for each other on the DEX. The DEX, in this case, is searched for a multi-step path to exchanging these assets via intermediate offers. This is not easily doable using the approach we have presented.

A DEX of this nature is susceptible to front-running. In our case, block issuers are given a permanent advantage in resolving liability transactions over non-block-issuing users. Among them, however, exactly one may issue the next block, including the liabilities they resolved.

REFERENCES

[1] Yoav Weiss and Dror Tirosh and Alex Forshtat: EIP-1613: Gas stations network. https://eips.ethereum.org/EIPS/eip-1613 (2018)

[2] Algorand Team: Algorand Developer Documentation. https://developer.algorand.org/docs/ (2021)

[3] Black, M., Liu, T., Cai, T.: Atomic loans: Cryptocurrency debt instruments (2019)

[4] Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P.: Native custom tokens in the extended UTXO model. In: Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. LNCS, vol. 12478 (2020)

[5] Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P., Zahnentferner, J.: UTXO$_{\mathrm{ma}}$: UTXO with multi-asset support. In: Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. LNCS, vol. 12478 (2020)

[6] Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Peyton Jones, M., Wadler, P.: The Extended UTXO model. In: Proceedings of Trusted Smart Contracts (WTSC). LNCS, vol. 12063. Springer (2020)

[7] Chiu, M., Kalabić, U.: Debt representation in UTXO blockchains. In: Financial Cryptography and Data Security 2021 (2021)

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)

[9] Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II. LNCS, vol. 9057, pp. 281–310. Springer (2015)

[10] Herlihy, M.: Atomic cross-chain swaps (2018)

[11] IDEX Team: IDEX documentation. https://docs.idex.io/ (2021)

[12] Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[13] Komodo Team: AtomicDEX documentation. https://developers.komodoplatform.com/basic-docs/atomicdex/ (2021)

[14] Kyber Team: Kyber: An On-Chain Liquidity Protocol. https://files.kyber.network/Kyber_Protocol_22_April_v0.1.pdf (2019)

[15] Stellar Development Foundation: Stellar Development Guides. https://developers.stellar.org/docs/ (2020)

[16] Team, M.: The maker protocol: Makerdao's multi-collateral dai (mcd) system. https://makerdao.com/en/whitepaper/ (Retrieved May 26, 2021)

[17] Waves.Exchange Team: Waves.Exchange Documentation. https://docs.waves.exchange/en/waves-exchange/ (2021)

[18] Zahnentferner, J.: Chimeric ledgers: Translating and unifying UTxO-based and account-based cryptocurrencies. IACR Cryptology ePrint Archive **2018**, 262 (2018), http://eprint.iacr.org/2018/262

A DEFINITIONS SUPPORTING THE FORMAL LEDGER RULES

The $\mathrm{UTXO}_{\mathrm{ll}}$ ledger rules presented in Section 3 are based on those of $\mathrm{UTXO}_{\mathrm{ma}}$ [5]. This appendix summarises supporting definitions for ease of reference. There is, however, one notable simplification in the system that we use here and the original $\mathrm{UTXO}_{\mathrm{ma}}$ system. The original $\mathrm{UTXO}_{\mathrm{ma}}$ system, while not supporting general-purpose scripts for smart contracts, does support a special-purpose language for defining asset policy script as well as multisig and timed UTXO validator scripts. In the present work, we keep the asset policy script, but we restrict UTXO outputs to be simply pay-to-pubkey outputs. This is to keep the presentation simpler and because the additional functionality of $\mathrm{UTXO}_{\mathrm{ma}}$ doesn't lead to additional insight in the context of the present paper.

$\mathbb{B}$	the type of Booleans
$\mathbb{N}$	the type of natural numbers
$\mathbb{Z}$	the type of integers
$\mathbb{H}$	the type of bytestrings: $\bigcup_{n=0}^{\infty}\{0,1\}^{8n}$
$(\phi_1 : T_1, \ldots, \phi_n : T_n)$	record type with fields ϕ_i of types T_i
$t.\phi$	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 |_| as length
$h :: t$	the list with head h and tail t
$h + +t$	list concatenation
$x \mapsto f(x)$	an anonymous function
$c^{\#}$	cryptographic collision-resistant hash of c
$\mathsf{Interval}[A]$	type of intervals over totally-ordered set A
$\mathsf{FinSup}[K, M]$	type of finitely supported functions from a type K to a monoid M

Figure 3: Basic types and notation

Figure 3 includes some basic types and notation. Crucial are finitely-supported functions, which provide the algebraic structure underpinning token bundles on a multi-asset ledger.

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 $f : K \longrightarrow M$ is *finitely supported* if $f(k) \neq 0$ for only finitely many $k \in K$. More precisely, for $f : K \longrightarrow M$ we define the *support* of f to be $\mathsf{supp}(f) = \{k \in K : f(k) \neq 0\}$ and $\mathsf{FinSup}[K, M] = \{f : K \longrightarrow M : |\mathsf{supp}(f)| < \infty\}$.

If $(M, +, 0)$ is a monoid then $\mathsf{FinSup}[K, M]$ also becomes a monoid if we define addition pointwise (i.e., $(f+g)(k) = f(k)+g(k)$), 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 **??**). Finitely-supported functions are easily implemented as finite maps, with a failed map lookup corresponding to returning 0.

A.1 Ledger types

Figure 4 defines the ledger primitives and types that we need to define the $\mathrm{UTXO}_{\mathrm{ll}}$ model. All outputs use a pay-to-pubkey-hash scheme, where an output is locked with the hash of key of the owner. We use a simple scripting language for forging policies, which we don't detail here any further — please see [5] for details. 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 an *asset*, which defines the asset class, to a Quantity of that asset.[6] Since a 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*.

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. Quantities in forge fields can also be negative, which effectively burns existing tokens.

Additionally, transactions get a *scripts* field holding a set of forging policy scripts: $\mathsf{Set}[\mathsf{Script}]$. This provides the forging policy scripts that are required as part of validation when tokens are minted or destroyed (see Rule 8 in Figure 2). The forging scripts of the assets being forged are executed and the transaction is only considered valid if the execution of the script returns true. 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 Policy, 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 [6]). 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.

[6]We have chosen to represent 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 PolicyIDs and AssetIDs to Quantitys.

Ledger primitives

Quantity	an amount of currency, forming an abelian group (typically $\mathbb{Z}$)
AssetID	a type consisting of identifiers for individual asset classes
Tick	a tick
Address	an "address" in the blockchain
TxId	the identifier of a transaction
$\mathsf{txId} : \mathsf{Tx} \rightarrow \mathsf{TxId}$	a function computing the identifier of a transaction
$\mathsf{lookupTx} : \mathsf{Ledger} \times \mathsf{TxId} \rightarrow \mathsf{Tx}$	retrieve the unique transaction with a given identifier
$\mathsf{verify} : \mathsf{PubKey} \times \mathbb{H} \times \mathbb{H} \rightarrow \mathbb{B}$	signature verification
$\mathsf{keyAddr} : \mathsf{PubKey} \rightarrow \mathsf{Address}$	the address of a public key
Script	forging policy scripts
$\mathsf{scriptAddr} : \mathsf{Script} \rightarrow \mathsf{Address}$	the address of a script
$[\![\_]\!] : \mathsf{Script} \rightarrow (\mathsf{Address} \times \mathsf{Tx} \times \mathsf{Set}[\mathsf{Output}]) \rightarrow \mathbb{B}$	apply script inside brackets to its arguments

Ledger types

$$
\begin{array}{rcl}
\mathsf{PolicyID} & = & \mathsf{Address} \qquad \text{(an identifier for a custom asset)} \\
\mathsf{Signature} & = & \mathbb{H} \\
\\
\mathsf{AssetID} & = & (\mathit{pid} : \mathsf{PolicyID}, \mathit{aName} : \mathsf{AssetName}) \\
\\
\mathsf{Quantities} & = & \mathsf{FinSup}[\mathsf{AssetID}, \mathsf{Quantity}] \\
\\
\mathsf{Output} & = & (\mathit{addr} : \mathsf{Address}, \mathit{value} : \mathsf{Quantities}) \\
\\
\mathsf{OutputRef} & = & (\mathit{id} : \mathsf{TxId}, \mathit{index} : \mathsf{Int}) \\
\\
\mathsf{Input} & = & (\mathit{outputRef} : \mathsf{OutputRef} \\
 & & \ \mathit{key} : \mathsf{PubKey}) \\
\\
\mathsf{Tx} & = & (\mathit{inputs} : \mathsf{Set}[\mathsf{Input}], \\
 & & \ \mathit{outputs} : \mathsf{List}[\mathsf{Output}], \\
 & & \ \mathit{validityInterval} : \mathsf{Interval}[\mathsf{Tick}], \\
 & & \ \mathit{forge} : \mathsf{Quantities} \\
 & & \ \mathit{scripts} : \mathsf{Set}[\mathsf{Script}], \\
 & & \ \mathit{sigs} : \mathsf{Set}[\mathsf{Signature}]) \\
\\
\mathsf{Ledger} & = & \mathsf{List}[\mathsf{Tx}]
\end{array}
$$

Figure 4: Ledger primitives and basic types