The Extended UTXO Ledger Model

18th April 2020

1 Introduction: The Extended UTXO Model

The Cardano blockchain [Car, 2015-2019, Corduan et al., 2019] uses a variant of the *Unspent Transaction Output* (UTXO) model used by Bitcoin. Transactions consume *unspent outputs* (UTXOs) from previous transactions and produce new outputs which can be used as inputs to later transactions. Unspent outputs are the liquid funds on the blockchain. Users do not have individual accounts, but rather have a software *wallet* on a smartphone or PC which manages UTXOs on the blockchain and can initiate transactions involving UTXOs owned by the user. Every core node on the blockchain maintains a record of all of the currently unspent outputs, the *UTXO set*; when outputs are spent, they are removed from the UTXO set.

This document contains a description of some extensions of the UTXO model: the main aim of these extensions is to facilitate the implementation of *smart contracts*, programs which perform automated and irrevocable transfer of funds on the blockchain, subject to certain conditions being met. A smart contract may involve multiple transactions, and our aim is to define a transaction model which enables the implementation of highly expressive contracts.

An important feature of our UTXO models is *scripts*, programs which run on the blockchain to check the validity of transactions. In Cardano, scripts will be programs in the Plutus Core language [IOHK, 2019]. The Extended UTXO models are largely agnostic as to the scripting language.

1.1 Structure of the document

The papers [Zahnentferner, 2018a] and [Zahnentferner, 2018b] give a formal specification of a basic UTXO model. See Note 2 for some background on this model.

This document proposes two extensions of the basic UTXO model (EUTXO stands for *Extended UTXO*):

- **EUTXO-1** (Section 3): this extends the basic UTXO model with enhanced scripting features, allowing the implementation of complex smart contracts.

- **EUTXO-2** (Section 4): this adds multicurrency features to EUTXO-1, allowing users to define *custom currencies* and *non-fungible tokens*.

The rationale for providing two separate extensions is that (1) introducing the extensions separately clarifies the structure of the models and makes it easier to explain the relevant design decisions, and (2) it is possible that a particular blockchain might not need the full power of EUTXO-2 and so could use the simpler EUTXO-1 model, perhaps with less computational overhead.

For ease of reference we have kept exposition to a minimum in the main text. Some aspects of the models are explained in more detail in Section A, with cross-references in the main text. Further explanation and many examples are contained in the book [Brünjes and Vinogradova, 2019].

2 Notation

This section defines some basic notation. We generally follow the notation established by [Zahnentferner, 2018b], except that we make use of finitely-supported functions in most places that [Zahnentferner, 2018b] use maps.

2.1 Basic types and operations

This section describes some types, notation, and conventions used in the remainder of the document.

- Types are typeset in $\mathsf{sans\ serif}$.
- $\mathbb{B}$ denotes the type of booleans, $\{\mathsf{false}, \mathsf{true}\}$.
- $\mathbb{N}$ denotes the type of natural numbers, $\{0, 1, 2, \ldots\}$.
- $\mathbb{Z}$ denotes the type of integers, $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- We regard $\mathbb{N}$ as a subtype of $\mathbb{Z}$ and convert freely between natural numbers and non-negative integers.
- $\mathbb{H}$ denotes the type of bytestrings, $\bigcup_{n=0}^{\infty}\{0,1\}^{8n}$.

  $\emptyset$ denotes the empty bytestring.

  A bytestring is a sequence of 8-bit bytes: the symbol $\mathbb{H}$ is used because bytestrings are often presented as sequences of hexadecimal digits.
- If a type M is a monoid, we use $+$ for the monoidal operation and 0 for the unit of the monoid.

  If M is a commutative monoid, we use $\sum$ for the extension of $+$ to a finite set of elements of type M.

  If M is a group, we use $-$ for the group inverse operation.

  This should never be ambiguous.
- A record type with fields $\phi_1, \ldots, \phi_n$ of types $T_1, \ldots, T_n$ is denoted by $(\phi_1 : T_1, \ldots, \phi_n : T_n)$.

  If t is a value of a record type T and ϕ is the name of a field of T then $t.\phi$ denotes the value of ϕ for t.
- If T is a type then $\mathsf{FinSet}[T]$ is the type of finite sets with elements of type T.
- A list l of type $\mathsf{List}[T]$ is either the empty list $[]$ or a list $e :: l$ with *head* e of type T and *tail* l of type $\mathsf{List}[T]$. A list has only a finite number of elements. We denote the ith element of a list l by $l[i]$ and the length of l by $|l|$.
- $x \mapsto f(x)$ denotes an anonymous function.

- A cryptographic collision-resistant hash of a value c is denoted $c^{\#}$.
- For a type A which forms a total order, $\mathsf{Interval}[A]$ is the type of intervals over that type. Intervals may be bounded or unbounded, and open or closed at either end. The type $\mathsf{Interval}[A]$ forms a lattice under inclusion.

2.2 Finitely-supported functions

Finitely-supported functions are a generalisation of maps to monoidal values. They always return an answer (which will in all but finitely many cases be zero), and can be queried for the set of non-zero points in their domain.

For two types K and V where V is a monoid, $\mathsf{FinSup}[K, V]$ denotes the type of *finitely-supported functions* from K to V. That is, there is a function $\mathsf{support} : \mathsf{FinSup}[K, V] \to \mathsf{FinSet}[K]$ such that $k \in \mathsf{support}(f) \Leftrightarrow f(k) \neq 0$.

Equality on finitely-supported functions is defined as pointwise equality. Similarly, if V has a partial order, then a partial order on finitely-supported functions is also defined pointwise.

If the type M is a monoid then we define the sum of two finitely-supported functions $f, g \in \mathsf{FinSup}[K, M]$ to be the function $f + g \in \mathsf{FinSup}[K, M]$ given by

$$(f + g)(k) = f(k) + g(k)$$

Note that the type $\mathsf{FinSup}[K, M]$ is a monoid with this operation, and the empty function as identity element.

If the type M is a group, then we can similarly define the inverse of a finitely-supported function f as the function $(-f)$ with the same support, given by

$$(-f)(k) = -f(k)$$

Again, $\mathsf{FinSup}[K, M]$ is a group with this operation.

See Note 1 for discussion of using finitely-supported functions computationally.

2.3 The Data type

We also define a type Data which can be used to pass information into scripts in a type-safe manner: see Figure 1. The definition is given here in EBNF form, but can easily be translated to a Haskell type, for instance.

```
Data =
    "I" ℤ
  | "B" ℍ
  | "Constr" ℕ (List[Data])
  | "List" List[Data]
  | "Map" List[Data × Data]
```

Figure 1: The Data type

Thus values of type Data are nested sums and products built up recursively from the base types of integers and bytestrings. This allows one to encode a large variety of first-order data structures:

for example, we could encode values of Haskell's `Maybe Integer` type using `Constr 0 []` to represent `Nothing` and `Constr 1 [I 41]` to encode `Just 41`.

The `List` and `Map` constructors are strictly redundant, but are included for convenience to allow straightforward encoding of lists and records.

We assume that the scripting language has the ability to parse values of type Data, converting them into a suitable internal representation.

3 EUTXO-1: Enhanced scripting

The EUTXO-1 model adds the following new features to the model proposed in [Zahnentferner, 2018b]:

- Every transaction has a *validity interval*, of type Interval[SlotNumber]. A core node will only process the transaction if the current slot number lies within the transaction's validity interval.
- The redeemer script of Zahnentferner [2018b] has been replaced with a *redeemer object* (*redeemer* for short) of type Data.
- Each unspent output now has an object of type Data associated with it: we call this the output's *datum* (or occasionally *datum object*) (see Note 4). Only the hash *datumHash* of the datum is stored in the output: the full value must be provided when the output is spent, much like the validator.
- Validator scripts make use of information about the pending transaction (ie, the transaction which is just about to take place, assuming that validation succeeds). This information is contained in a structure which we call Context (see Section 3.1.2 for its definition). We may refer to this information as the *validation context* in cases where ambiguity may arise.
- Validation of an output is performed by running the validator with three inputs:
  1. the datum,
  2. the redeemer,
  3. the Context information, encoded as Data.

3.1 A Formal Description of the EUTXO-1 Model

In this section we give a formal description of the EUTXO-1 model. The description is given in a straightforward set-theoretic form, which (a) admits an almost direct translation into Haskell, and (b) should easily be amenable to mechanical formalisation. This will potentially allow us to argue formally about smart contracts and to develop tools for automatic contract analysis.

The definitions in this section are essentially the definitions of UTXO-based cryptocurrencies with scripts from [Zahnentferner, 2018b], except that we have added the new features mentioned above (the validity interval, the datum and the Context structure), changed the type of the redeemer from Script to Data, and used finitely-supported functions in place of maps.

Figure 2 lists the types and operations used in the the basic EUTXO model. Some of these are defined, the others must be provided by the ledger.

Ledger primitives

$\mathsf{Quantity}$	an amount of currency
$\mathsf{SlotNumber}$	a slot number
$\mathsf{Address}$	the “address” of a script in the blockchain
$\mathsf{DataHash}$	the hash of an object of type Data
TxId	the identifier of a transaction
$\mathsf{txId} : \mathsf{Tx} \to \mathsf{TxId}$	a function computing the identifier of a transaction
$\mathsf{lookupTx} : \mathsf{Ledger} \times \mathsf{TxId} \to \mathsf{Tx}$	a function retrieving a transaction via its identifier
Script	the (opaque) type of scripts
$\mathsf{scriptAddr} : \mathsf{Script} \to \mathsf{Address}$	the address of a script
$\mathsf{dataHash} : \mathsf{Data} \to \mathsf{DataHash}$	the hash of a data object
$[\![\cdot]\!] : \mathsf{Script} \to \mathsf{Data} \times \cdots \times \mathsf{Data} \to \mathbb{B}$	application of a script to its arguments

Defined types

$$\begin{aligned}
\mathsf{Output} &= (\mathit{addr} : \mathsf{Address},\\
&\quad \mathit{value} : \mathsf{Quantity},\\
&\quad \mathit{datumHash} : \mathsf{DataHash})\\[1em]
\mathsf{OutputRef} &= (\mathit{id} : \mathsf{TxId}, \mathit{index} : \mathsf{Int})\\[1em]
\mathsf{Input} &= (\mathit{outputRef} : \mathsf{OutputRef},\\
&\quad \mathit{validator} : \mathsf{Script},\\
&\quad \mathit{datum} : \mathsf{Data},\\
&\quad \mathit{redeemer} : \mathsf{Data})\\[1em]
\mathsf{Tx} &= (\mathit{inputs} : \mathsf{FinSet}[\mathsf{Input}],\\
&\quad \mathit{outputs} : \mathsf{List}[\mathsf{Output}],\\
&\quad \mathit{validityInterval} : \mathsf{Interval}[\mathsf{SlotNumber}],\\
&\quad \mathit{datumWitnesses} : \mathsf{FinSup}[\mathsf{DataHash}, \mathsf{Data}],\\
&\quad \mathit{fee} : \mathsf{Quantity},\\
&\quad \mathit{forge} : \mathsf{Quantity})\\[1em]
\mathsf{Ledger} &= \mathsf{List}[\mathsf{Tx}]
\end{aligned}$$

Figure 2: Primitives and basic types for the EUTXO-1 model

3.1.1 Remarks

ETUXO-1 on Cardano. The Cardano implementation of EUTXO-1 uses the primitives given in Figure 3.

$$
\begin{array}{rcl}
\mathsf{Quantity} & = & \mathbb{Z} \\
\mathsf{SlotNumber} & = & \mathbb{N} \\
\mathsf{Address} & = & \mathbb{H} \\
\mathsf{DataHash} & = & \mathbb{H} \\
\mathsf{TxId} & = & \mathbb{H} \\
\mathsf{txId} : \mathsf{Tx} \rightarrow \mathsf{TxId} & = & t \mapsto t^{\#} \\
\mathsf{Script} & & \text{a Plutus Core program} \\
\mathsf{scriptAddr} : \mathsf{Script} \rightarrow \mathsf{Address} & = & s \mapsto s^{\#} \\
[\![\cdot]\!] : \mathsf{Script} \rightarrow \mathsf{Data} \times \cdots \times \mathsf{Data} \rightarrow \mathbb{B} & & \text{running the Plutus Core interpreter with a} \\
 & & \text{script and a number of data objects as input}
\end{array}
$$

Figure 3: Cardano primitives for the EUTXO-1 model

Transaction identifiers. We assume that each transaction has a unique identifier (in Cardano, the hash of a Tx object) and that a transaction can be efficiently retrieved from a ledger using the lookupTx function.

Inputs and outputs. Note that a transaction has a Set of inputs but a List of outputs. See Note 3 for a discussion of why this is.

Validator addresses in outputs. The *addr* field of an output should contain the address of the validator script for that output: this requirement is enforced in Rule 8 of Figure 6 below.

Scripts and hashes. Note that datum objects and validators are provided as parts of transaction inputs, even though they are conceptually part of the output being spent. The reasons for this are explained in Note 6.

Applying scripts A script s may expect some number n of datum objects as arguments (the number n depending on the type of the script). The result of running the script with the datum objects $d_1, \ldots, d_n$ as arguments is denoted by $[\![s]\!](d_1, \ldots, d_n)$. As mentioned at the start of this section, validator scripts take three arguments.

Datum witnesses. The transaction may include the full value of the datum for each output that it creates. See Note 7 for more discussion.

Fees. Users are charged a fee for the on-chain storage and execution costs of a transaction, and this is included in the EUTXO models. The details are not important for the purposes of the models, but see Note 5 for some more discussion.

Special types of transaction. In a practical implementation it might be useful to include special cases for common transaction types such as pay-to-pukbey transactions in order to increase efficiency and decrease storage requirements (and hence reduce fees). These have been omitted from this model because it subsumes all of the other transaction types we're likely to encounter, and also because it's difficult to give a definitive list of such special cases.

Ledger structure. We model a ledger as a simple list of transactions: a real blockchain ledger will be more complex than this, but the only property that we really require is that transactions in the ledger have some kind of address which allows them to be uniquely identified and retrieved.

3.1.2 The Context type

Recall from the introduction to Section 3 that when a transaction input is being validated, the validator is supplied with an object of type Context which contains information about the pending transaction. The Context type for the current version of EUTXO-1 is defined in Figure 4, along with some related types.

$$
\begin{aligned}
\mathsf{OutputInfo} \quad &= \quad (\mathit{value} : \mathsf{Quantity}, \\
&\qquad \mathit{validatorHash} : \mathsf{Address}, \\
&\qquad \mathit{datumHash} : \mathsf{DataHash}) \\[1em]
\mathsf{InputInfo} \quad &= \quad (\mathit{outputRef} : \mathsf{OutputRef}, \\
&\qquad \mathit{validatorHash} : \mathsf{Address}, \\
&\qquad \mathit{datumHash} : \mathsf{DataHash}, \\
&\qquad \mathit{redeemerHash} : \mathsf{DataHash}, \\
&\qquad \mathit{value} : \mathsf{Quantity}) \\[1em]
\mathsf{Context} \quad &= \quad (\mathit{inputInfo} : \mathsf{List}[\mathsf{InputInfo}], \\
&\qquad \mathit{thisInput} : \mathbb{N}, \\
&\qquad \mathit{outputInfo} : \mathsf{List}[\mathsf{OutputInfo}], \\
&\qquad \mathit{validityInterval} : \mathsf{Interval}[\mathsf{SlotNumber}], \\
&\qquad \mathit{datumWitnesses} : \mathsf{FinSup}[\mathsf{DataHash}, \mathsf{Data}], \\
&\qquad \mathit{fee} : \mathsf{Quantity}, \\
&\qquad \mathit{forge} : \mathsf{Quantity})
\end{aligned}
$$

$$
\begin{array}{rl}
\mathsf{mkContext} : \mathsf{Tx} \times \mathsf{Input} \times \mathsf{Ledger} \rightarrow \mathsf{Context} & \text{summarises a transaction in the context} \\
 & \text{of an input and a ledger state} \\[1em]
\mathsf{toData} : \mathsf{Context} \rightarrow \mathsf{Data} & \text{encodes a } \mathsf{Context} \text{ as } \mathsf{Data}
\end{array}
$$

Figure 4: The Context type for the EUTXO-1 model

3.2 Remarks

The contents of Context. The Context type is essentially a summary of the information contained in the Tx type in Figure 2. The *fee*, *forge*, and *validityInterval* fields are copied directly from the pending transaction. The *outputInfo* field contains information about the outputs which will be produced if the pending transaction validates successfully: it contains only the address of the relevant validator, and the hash of the datum.[1] The *inputInfo* field contains information about the inputs to the pending transaction, but provides only the hashes of the

[1] See Note 8 for further explanation.

validators and redeemers for the inputs. The *thisInput* field is an index pointing to the element of *inputInfo* relating to the input currently undergoing validation.

Defining mkContext and toData. Assuming we have an appropriate hashing function, it is straightforward to define mkContext. For the implementation of toData, note that the *inputs* field is a FinSet[] in Tx, but a List[] in Context. Therefore toData has to introduce an ordering of the transaction inputs. Contract authors cannot make any assumptions about this ordering and therefore should ensure that their scripts pass or fail regardless of what particular permutation of transaction inputs they are presented with.

Apart from that, the function toData is implementation-dependent and we will not discuss it further.

Determinism. The information provided in the Context structure is sufficiently limited that the validation process becomes *deterministic*, which has important implications for fee calculations. See Note 9 for further discussion.

3.3 Validity of EUTXO-1 transactions

A number of conditions must be satisfied in order for a transaction t to be considered valid with respect to a ledger l.

Figure 5 defines some auxiliary functions used in validation.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathsf{unspentTxOutputs} : \mathsf{Tx} \rightarrow \mathsf{FinSet}[\mathsf{OutputRef}]} \\
\mathsf{unspentTxOutputs}(t) & = & \{(\mathsf{txId}(t), 1), \ldots, (\mathsf{txId}(id), |t.outputs|)\} \\
\\
\multicolumn{3}{l}{\mathsf{unspentOutputs} : \mathsf{Ledger} \rightarrow \mathsf{FinSet}[\mathsf{OutputRef}]} \\
\mathsf{unspentOutputs}([]) & = & \{\} \\
\mathsf{unspentOutputs}(t :: l) & = & (\mathsf{unspentOutputs}(l) \setminus t.inputs) \cup \mathsf{unspentTxOutputs}(t) \\
\\
\multicolumn{3}{l}{\mathsf{getSpentOutput} : \mathsf{Input} \times \mathsf{Ledger} \rightarrow \mathsf{Output}} \\
\mathsf{getSpentOutput}(i, l) & = & \mathsf{lookupTx}(l, i.outputRef.id).outputs[i.outputRef.index]
\end{array}$$

Figure 5: Auxiliary functions for transaction validation

It is perhaps not immediately obvious that the unspentOutputs function only yields a *finite* set of outputs: however, this can be proved by induction on the slot number, using the facts that the initial ledger is empty and that each transaction only produces a finite number of outputs.

Note also that getSpentOutput uses the lookupTx function, which can of course fail if the ledger contains no transaction with the relevant identifier; however we only use getSpentOutput during transaction validation, and our validity rules ensure that in that case transaction lookup will always succeed: see Note 10.

We can now define what it means for a transaction t of type Tx to be valid for a ledger l during the slot currentSlot: see Figure 6. Our definition combines Definitions 6 and 14 from [Zahnentferner, 2018b], differing from the latter in Rule 7.

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

$$\mathsf{currentSlot} \in t.\mathit{validityInterval}$$

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

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

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

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

4. **Forging**

   A transaction with a non-zero *forge* field is only valid if the ledger l is empty (that is, if it is the initial transaction).

5. **Value is preserved**

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

6. **No output is double spent**

$$\text{If } i_1, i_2 \in t.\mathit{inputs} \text{ and } i_1.\mathit{outputRef} = i_2.\mathit{outputRef} \text{ then } i_1 = i_2.$$

7. **All inputs validate**

$$\text{For all } i \in t.\mathit{inputs},\ [\![i.\mathit{validator}]\!](i.\mathit{datum}, i.\mathit{redeemer}, \mathsf{toData}(\mathsf{mkContext}(t, i, l))) = \mathsf{true}.$$

8. **Validator scripts match output addresses**

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

9. **Datum objects match output hashes**

$$\text{For all } i \in t.\mathit{inputs},\ \mathsf{dataHash}(i.\mathit{datum}) = \mathsf{getSpentOutput}(i, l).\mathit{datumHash}$$

Figure 6: Validity of a transaction t in the EUTXO-1 model

We say that 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'.

In practice, validity imposes a limit on the sizes of the *validator* Script, the *redeemer* and *datum* Data fields, and the result of toData. The validation of a single transaction must take place within one slot, so the evaluation of $[\![\cdot]\!]$ cannot take longer than one slot.

4 EUTXO-2: multicurrency support and non-fungible tokens

We now extend the EUTXO-1 model further by introducing features which allow, among other things, the implementation of new currencies and *non-fungible tokens* (NFTs).

Multiple currencies. The EUTXO-2 model allows an unlimited number of *currencies*. Each custom currency has a unique identifier and a *monetary policy script* which may be used to limit the way in which the currency is used (for example, by only allowing specified users to create units of the currency).

NFTs. A non-fungible token (NFT) is a unique object which can be transferred to another user, but not duplicated. NFTs have proven useful in a number of blockchain applications (see [Ethereum, 2017] for example); for example, they can represent ownership of some object in a game. We can implement NFTs as custom currencies whose supply is limited to a single coin.

4.1 The definition of EUTXO-2

In order to support these extensions, we introduce several new types. Custom currencies are represented by unique *currency identifiers* and each currency has a number of *tokens* which partition each custom currency into a number of sub-currencies. The basic idea is that ordinary currencies have a single token whose sub-currency has an unlimited supply and NFTs have a number of tokens with the sub-currency for each token limited to a supply of one.

The changes to the basic EUTXO-1 types are quite simple: see Figure 7. We change the type of the *value* field in the Output type to be Quantities, representing values of all currencies. We also change the type of the *forge* field on transactions to Quantities, to allow the creation and destruction of funds in all currencies; the supply of a currency can be reduced by forging a negative amount of that currency, as in EUTXO-1. In addition, transactions now have a set *forgeScripts* of monetary policy scripts, each of which takes a single Data argument summarising the current transaction; we assume that there is a function $\mathsf{toTxData} : \mathsf{Tx} \rightarrow \mathsf{Data}$ which creates such objects.

Ledger primitives

Token	a type consisting of identifiers for individual tokens
$\mathsf{toTxData} : \mathsf{Tx} \to \mathsf{Data}$	encode a transaction as Data

Defined types

$$
\begin{aligned}
\mathsf{CurrencyId} &= \mathsf{Address}\ \text{(an identifier for a custom currency)}\\[1ex]
\mathsf{Quantities} &= \mathsf{FinSup}[\mathsf{CurrencyId}, \mathsf{FinSup}[\mathsf{Token}, \mathsf{Quantity}]]\\[1ex]
\mathsf{Output}_2 &= (\mathit{addr} : \mathsf{Address},\\
&\qquad \mathit{value} : \mathsf{Quantities}\\
&\qquad \mathit{datumHash} : \mathsf{DataHash})\\[1ex]
\mathsf{OutputRef}_2 &= (\mathit{id} : \mathsf{TxId}, \mathit{index} : \mathsf{Int})\\[1ex]
\mathsf{Input}_2 &= (\mathit{outputRef} : \mathsf{OutputRef}_2,\\
&\qquad \mathit{validator} : \mathsf{Script},\\
&\qquad \mathit{datum} : \mathsf{Data},\\
&\qquad \mathit{redeemer} : \mathsf{Data})\\[1ex]
\mathsf{Tx}_2 &= (\mathit{inputs} : \mathsf{FinSet}[\mathsf{Input}_2],\\
&\qquad \mathit{outputs} : \mathsf{List}[\mathsf{Output}_2],\\
&\qquad \mathit{validityInterval} : \mathsf{Interval}[\mathsf{SlotNumber}],\\
&\qquad \mathit{datumWitnesses} : \mathsf{FinSup}[\mathsf{DataHash}, \mathsf{Data}],\\
&\qquad \mathit{fee} : \mathsf{Quantities},\\
&\qquad \mathit{forge} : \mathsf{Quantities},\\
&\qquad \mathit{forgeScripts} : \mathsf{FinSet}[\mathsf{Script}])\\[1ex]
\mathsf{Ledger}_2 &= \mathsf{List}[\mathsf{Tx}_2]
\end{aligned}
$$

Figure 7: Extra primitives and basic types for the EUTXO-2 model

4.1.1 Remarks

ETUXO-2 on Cardano. The Cardano implementation of EUTXO-2 uses the primitives given in Figure 8. Cardano also defines an *native currency* and *native currency token*. This allows defining a native currency that behaves as a simple $\mathsf{Quantity}$. This is used in Fig 11.

$$
\begin{aligned}
\mathsf{CurrencyId} &= \mathbb{H}\\
\mathsf{Token} &= \mathbb{H}\\
\text{nativeC} &= \emptyset\\
\text{nativeT} &= \emptyset
\end{aligned}
$$

Figure 8: Cardano primitives for the EUTXO-2 model

Quantities. The Quantities type represents a collection of funds from a number of currencies and their subcurrencies.

Quantities is a finitely-supported function *to* another finitely-supported function. This is well-defined, since finitely-supported functions form a monoid.

4.2 The Context type for EUTXO-2

The Context type must be also be updated for the EUTXO-2 model. All that is required is to replace Quantity by Quantities everywhere in Figure 4 except for the *fee* field, and to add the monetary policy scripts: for reference the details are given in Figure 9.

$$
\begin{array}{rcl}
\mathsf{OutputInfo}_2 & = & (\mathit{value} : \mathsf{Quantities}, \\
 & & \;\mathit{validatorHash} : \mathsf{Address}, \\
 & & \;\mathit{datumHash} : \mathsf{DataHash}) \\[1em]
\mathsf{InputInfo}_2 & = & (\mathit{outputRef} : \mathsf{OutputRef}, \\
 & & \;\mathit{validatorHash} : \mathsf{Address}, \\
 & & \;\mathit{datumHash} : \mathsf{DataHash}, \\
 & & \;\mathit{redeemerHash} : \mathsf{DataHash}), \\
 & & \;\mathit{value} : \mathsf{Quantities}) \\[1em]
\mathsf{Context}_2 & = & (\mathit{inputInfo} : \mathsf{List}[\mathsf{InputInfo}_2], \\
 & & \;\mathit{thisInput} : \mathbb{N}, \\
 & & \;\mathit{outputInfo} : \mathsf{List}[\mathsf{OutputInfo}_2], \\
 & & \;\mathit{validityInterval} : \mathsf{Interval}[\mathsf{SlotNumber}], \\
 & & \;\mathit{datumWitnesses} : \mathsf{FinSup}[\mathsf{DataHash}, \mathsf{Data}], \\
 & & \;\mathit{fee} : \mathsf{Quantities}, \\
 & & \;\mathit{forge} : \mathsf{Quantities}, \\
 & & \;\mathit{forgeScripts} : \mathsf{FinSet}[\mathsf{Script}])
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{mkContext}_2 : \mathsf{Tx}_2 \times \mathsf{Input} \times \mathsf{Ledger} \rightarrow \mathsf{Context}_2 & \text{summarises a transaction in the context} \\
 & \text{of an input and a ledger state} \\[1em]
\mathsf{toData}_2 : \mathsf{Context}_2 \rightarrow \mathsf{Data} & \text{encodes a } \mathsf{Context}_2 \text{ object}
\end{array}
$$

Figure 9: The Context type for the EUTXO-2 model

4.3 Validity of EUTXO-2 transactions

The validity conditions from Figure 6 must also be updated to take account of multiple currencies.

We can now adapt the definition of validity for EUTXO-1 (Figure 6) to obtain a definition of validity for EUTXO-2: see Figure 10.

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

$$\mathsf{currentSlot} \in t.\mathit{validityInterval}$$

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

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

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

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

4. **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 $h \in \mathsf{support}(t.\mathit{forge})$, there exists $s \in t.\mathit{forgeScripts}$ with $\mathsf{scriptAddr}(s) = h$.

5. **Values are preserved**

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

6. **No output is double spent**

$$\text{If } i_1, i_2 \in t.\mathit{inputs} \text{ and } i_1.\mathit{outputRef} = i_2.\mathit{outputRef} \text{ then } i_1 = i_2.$$

7. **All inputs validate**

$$\text{For all } i \in t.\mathit{inputs},\ [\![i.\mathit{validator}]\!](i.\mathit{datum}, i.\mathit{redeemer}, \mathsf{toData}_2(\mathsf{mkContext}_2(t, i, l))) = \mathsf{true}$$

8. **Validator scripts match output addresses**

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

9. **Datum objects match output hashes**

$$\text{For all } i \in t.\mathit{inputs},\ \mathsf{dataHash}(i.\mathit{datum}) = \mathsf{getSpentOutput}(i, l).\mathit{datumHash}$$

10. **All monetary policy scripts evaluate to true**

$$\text{For all } s \in t.\mathit{forgeScripts},\ [\![s]\!](\mathsf{toTxData}(t)) = \mathsf{true}$$

Figure 10: Validity of a transaction t in the EUTXO-2 model

4.4 Remarks

Monetary policies. Rules 4b and 10 can be used to enforce monetary policies for custom currencies: see Note 11 for a detailed explanation.

Preservation of value over $\mathsf{Quantities}$. In Rule 5, $+$ and $\sum$ operate over $\mathsf{Quantities}$, which is a finitely-supported function (which, with their operations, are defined in Section 2.2). Preservation of value in this model essentially requires that the quantities of each of the individual currencies involved in the transaction are preserved.

Preservation of value and forging. Recall that values in *forge* can be negative whereas values in outputs must be non-negative. This allows currency to be destroyed as well as created. Rule 5 implies that a transaction is invalid if it attempts to destroy more of a currency than is actually available in its inputs.

Validation on Cardano. Cardano adds an additional rule in Fig 11, which asserts that fees are paid exclusively in the native currency.

Fees are paid in the native currency

$$\mathsf{support}(t.\mathit{fee}) = \{\mathrm{nativeC}\} \text{ and } \mathsf{support}(t.\mathit{fee}(\mathrm{nativeC})) = \{\mathrm{nativeT}\}$$

Figure 11: Validity of a transaction t in the EUTXO-2 model

4.5 The EUTXO-2 model in practice.

See Brünjes and Vinogradova [2019] for examples of contracts which make use of the features of the EUTXO-2 model. See also Notes 11 to 13 for comments on some technical aspects of the model.

A Comments

Note 1. Computing with finitely-supported functions. We intend that finitely-supported functions are implemented as finite maps, with a failed map lookup corresponding to returning 0.

However, there are two apparent difficulties:

1. The domain of a map does not correspond to the support of the function: values may be mapped to zero, thus appearing in the domain but not the support.

2. Pointwise equality is hard to compute.

However, both of these are easily ameliorated. We say that a set w is a *weak support* of a finitely-supported function f if $\mathsf{support}(f) \subseteq w$. That is, a weak support contains all the points that are non-zero, but possibly also some points that are zero. It is easy to see that the domain of a map is a weak support for the finitely-supported function it represents.

We can compute the support from the weak support by simply checking the function at each value and removing those that are zero. This is potentially expensive, but we only need to do it when we need the support, which we only do during the computation of Rule 5.

Pointwise equality between two finitely-supported functions f and g is equivalent to checking pointwise equality only over the union of $\mathsf{support}(f)$ and $\mathsf{support}(g)$; or similarly over the union of a weak support of f and of g. In particular, for finitely-supported functions represented as maps, we can check pointwise equality over the union of their domains.

The same applies to checking partial ordering pointwise, which can similarly be done over the union of the weak support.

Mathematically, most of our finiteness restrictions are not strictly required. For example, in the EUTXO-2 model (Section 4) we could allow transactions with infinitely many monetary policy scripts and outputs with non-finitely-supported quantities of token currencies; as long as the number of inputs and outputs of each transaction is finite the model remains mathematically sound. However, the finite model presented in this document is more realistic from the point of view of real-world implementation.

Note 2. The Basic UTXO model: Outputs and scripts. There is no well-defined notion of ownership for UTXOs. In many transactions an output will accrue to a single user who is then entitled to spend it at a later date. However, in general the notion of ownership is more complex: an output of a transaction might require the cooperation of several users before it could be spent, or it might not be spendable until some other condition has been met (for example a certain period of time may have to pass). At the extremes, an output could be spendable by anyone, or by no-one.

In order to deal with this complexity, an output can be locked by a *script*[2] which must be supplied with suitable evidence to unlock the output. In the basic model, each input to a transaction comes with a *validator* script which checks that the transaction is allowed to spend the output. In order to spend an output, the transaction supplies an object of type Data, called the *redeemer*, which provides evidence that the transaction has the authority to do so;[3] a process called *validation* is then performed which checks that the redeemer satisfies the conditions required by the validator. Before a transaction can proceed, all inputs must be successfully validated: if one or more inputs fails to validate then the transaction is rejected.

A simple example of this is a *pay-to-pubkey* script, where the redeemer consists of a signature for the current transaction produced using a private key belonging to the owner of the output. The validator script (provided by the owner of the output) would check the signature using a known public key: if the public key corresponds to the private key then validation succeeds, otherwise it fails. Thus the output can only be spent by the owner of the relevant private key

See Note 6 for more information about validators in the EUTXO setting.

Note 3. Inputs and outputs. A transaction has a Set of inputs but a List of outputs. This is for two reasons:

- We need a way to uniquely identify a transaction output, so that it can be referred to by a transaction input that spends it. The pair of a transaction id and an output index is sufficient for this, but other schemes are conceivable.

- Equality of transaction outputs is defined structurally. But that means that if we had two outputs paying X to address A, then they would be equal and therefore if we kept them in a Set one would be lost.

[2] In the Cardano setting, scripts are Plutus Core programs [IOHK, 2019].

[3] The validator plays a role similar to that of BitCoin's `scriptPubKey` and the redeemer to `scriptSig`.

An alternative design would be to include a unique nonce in transaction outputs (effectively: their index in the list), and then we could use this to identify them (and distinguish them from each other), and so we could keep them in a Set instead.

Note 4. The datum. The introduction of the datum increases the expressivity of the model considerably. For example, one can use a datum to propagate state between transactions, and this can be used to give a contract the structure of a finite state machine; the fact that the datum is part of the output and not the transaction means that the state can change without the transaction changing, which makes it easier to have an "identity" for an ongoing contract.

Note 5. Fees and Costs. Users may have to pay a fee in order to have a transaction executed. In a public blockchain an important reason for this is to deter hostile agents from carrying out denial of service attacks by submitting transactions which take a long time or use excessive amounts of memory. The precise details of fees in Cardano are outwith the scope of this document, and indeed have not been fully decided at the time of writing. However, we expect that the fee will include a component based on the size of the transaction (including its associated scripts), and also a so-called *gas* charge to cover execution costs. We will have a model specifying the costs of individual operations during script execution; costs will be monitored dynamically during execution, and if the gas consumed ever exceeds the amount covered by the fee then the transaction will fail.

Note 6. Scripts and Hashes. The spendability of an output is determined by its validator, and thus the validator for an output must be known at the time when the output is created (a completely new validator may be created, or an existing validator may be re-used).

Conceptually the validator is part of the output, so it may be rather unexpected that Figure 2 defines the validator to be part of an *input*, with the output only containing the address of the validator. The rationale for this is that a validator V for an output O is not required until O is actually spent, which may be some time after O was created. Recall from Note 5 that the cost of executing a transaction depends on the size of transaction, including the associated scripts. Thus the transaction that produces the validator only pays for the size of a hash (32 bytes) and the transaction that runs it pays for the full size of the script.

This strategy also helps to reduce on-chain storage requirements, since validators can be stored off-chain until needed (and the presence of the hash in the output can be used to check that the correct validator is in fact being used when validation occurs), but unspent outputs persist on-chain in the UTXO set until they are eventually spent.

The same strategy applies to datum objects.

Note 7. Datum witnesses. Although a datum is only recorded as a hash in a transaction output, it is useful to be able to record the full value of the datum on the transaction that *creates* an output: this allows observers to determine the full datum without it having to be kept in the UTXO set.

This mechanism is *optional*, since it incurs an increase in transaction size (and hence cost), and some clients may want to transmit the information off-chain instead to minimise these costs.

Hence there is a $\mathit{datumWitnesses}$ field on transactions, which *may* contain mappings from the $\mathsf{DataHashes}$ used in the transaction to their Data values. This information is also present in $\mathsf{Context}$.

Note 8. Datum objects in $\mathsf{Context}$. In Figures 4 and 9 the $\mathsf{OutputInfo}$ does not include the datum attached to the output. These may be found in $\mathit{datumWitnesses}$.

Having access to the value of the datum allows a validator to inspect an outgoing datum, for example to confirm that its contents are correct in some sense. This can be useful when a datum is used to propagate information about the state of a contract to later transactions. See Brünjes and Vinogradova [2019] for examples of this.

Note 9. Determinism of the validation process. The Context type is the only information about the "outside world" available to a validator at the time of validation. Allowing the validator access to this information gives the EUTXO models a considerable amount of power, as can be seen from the example contracts in Brünjes and Vinogradova [2019]. However, it is important not to make too much information available to the validator. The choice of the Context type above means that the information available to the validator is essentially independent of the state of the blockchain, and in particular, it is independent of time (note that the check that the current slot number is within a transaction's validity range takes place *before* validation is initiated, and the slot number is not passed to the validator (although the validity range is)). This implies that validation is *determinisitic* and validators can be run off-chain in order to determine their execution cost before on-chain validation actually occurs. This helps users to calculate transaction fees in advance and avoid the possibility of their transactions failing due to an insufficient fee having been paid (and also avoids overpayment due to overestimating the fees).

Note 10. Transaction lookup during validation. Note that the getSpentOutput function of Figure 5 uses lookupTx, which could fail if the relevant transaction does not exist. However, we only use getSpentOutput during transaction validation, and in that case Rule 3 of Figure 6 and Rule 3 of Figure 10 ensure that all of the transaction inputs refer to existing unspent outputs, and in these circumstances lookupTx will always succeed for the transactions of interest.

Note 11. Monetary policies for custom currencies. The new **Forging** rule in Figure 10 enables custom currencies to implement their own monetary policies: for example, one might wish to place some limit on the amount of a currency that can be forged, or restrict the creation of the currency to owners of particular public keys.

The idea is that a custom currency has a monetary policy which is defined by some script H, and the address $h = \mathsf{scriptAddr}(H)$ is used as the identifier of the currency.

Whenever a new quantity of the currency is forged, Rules 4b and 10 of Figure 10 imply that H must be contained in the *forgeScripts* field of the transaction, and that it must be successfully executed; H is provided with the details of the transaction via the Data object produced by toTxData, so it has access to the *forge* field of the transaction and knows how much of the currency is to be forged and can respond appropriately.[4]

The advantage of this scheme is that custom currencies can be handled entirely within the smart contract system, without the need to introduce any extra blockchain infrastructure such as a central registry of custom currencies.

In practice some refinement of this scheme will be required in order to (a) allow re-use of a monetary policy for different currencies, and (b) prevent unauthorised forging of a currency. To deal with (a) we can make the monetary policy script unique by including a nonce. This still doesn't prevent unauthorised people from using the script H to produce currency, but this can be prevented by, for instance, embedding a reference to an unspent output in the script and

[4] We do not insist that every monetary policy script in a transaction is associated with a currency which the transaction is actually forging, so a transaction may include apparently unnecessary monetary policy scripts. The creator of the transaction is responsible for providing the contents of the *forgeScripts* field and is free to include or exclude such scripts as they see fit.

requiring that the currency can only be forged if the referenced output is spent at the same time, so it can only be forged once.

Note 12. Implications of the EUTXO-2 model. The EUTXO-2 model and the techniques described in Note 11 allow us to implement fungible (normal) and non-fungible token currencies, as well as "mixed states":

- Standard (fungible) currencies are implemented by issuing currencies with a single Token.
- Non-fungible token currencies are implemented by only ever issuing single quantities of many unique Tokens.
- Note that there is nothing in this model which enforces uniqueness: having multiples of a single Token merely means that those can be used fungibly. If a currency wants to make sure it only issues unique tokens it must track this itself. These "mixed" token currencies can have many Tokens, but these can have more than unit quantities in circulation. These can be useful to model distinct categories of thing where there are fungible quantities within those, for example share classes.

Note 13. Performance issues for EUTXO-2. The EUTXO-2 model will lose some efficiency in comparison to the EUTXO-1 model, simply because the data structures are more complicated. This would even apply to transactions which only involve the native currency (if there is one), since it would be necessary to check whether the Quantities contains anything that needs to be processed. If this is a concern then one could implement a model with two types of transaction, essentially just the disjoint union of the EUTXO-1 and EUTXO-2 transaction types. A simple case distinction at the start of a transaction could then select either a fast native-currency-only computation or a slower multicurrency computation. This would be harder to maintain though.

Another optimisation would be possible if one wished to implement custom currencies but not NFTs: since in this case every currency would only have a single token, the tokens could be omitted and the Quantities replaced with a map from currency ids to quantities.

A more significant cost may be that we can no longer use `{-# UNPACK #-}` when our Quantity type stops being a simple combination of wrappers and products around primitives, but this is again an issue with any multi-currency proposal.

References

Cardano. https://www.cardano.org/en/home/, 2015-2019.

Lars Brünjes and Polina Vinogradova. *Plutus: Writing Reliable Smart Contracts.* 2019. Available at https://github.com/IntersectMBO/plutus/tree/master/plutus-book.

Jared Corduan, Polina Vinogradova, and Matthias Güdemann. A formal specification of the Cardano ledger. Technical report, IOHK, 2019. Available at https://github.com/IntersectMBO/cardano-ledger-specs.

Ethereum. ERC-721 standard for non-fungible tokens in Ethereum. http://erc721.org/, 2017.

IOHK. Formal Specification of the Plutus Core Language. Technical report, IOHK, 2019. Available at https://github.com/IntersectMBO/plutus/tree/master/plutus-core-spec.

Joachim Zahnentferner. Chimeric ledgers: Translating and unifying UTxO-based and account-based cryptocurrencies. *IACR Cryptology ePrint Archive*, 2018:262, 2018a. URL http://eprint.iacr.org/2018/262.

Joachim Zahnentferner. An abstract model of UTxO-based cryptocurrencies with scripts. *IACR Cryptology ePrint Archive*, 2018:469, 2018b. URL https://eprint.iacr.org/2018/469.