A Formal Specification of the Cardano Ledger with a Native Multi-Asset Implementation

Deliverable GL-D1

Polina Vinogradova polina.vinogradova@iohk.io
Andre Knispel andre.knispel@iohk.io

Project: Goguen Ledger

Type: Deliverable
Due Date: 31st July 2020

Responsible team: Formal Methods Team
Editor: Polina Vinogradova
Team Leader: Philipp Kant, IOHK

Version 1.0
20st August 2020

Dissemination Level		
PU	Public	√
CO	Confidential, only for company distribution	
DR	Draft, not for general circulation	

Formal Specification of the Cardano Ledger with a Native Multi-Asset Implementation

Polina Vinogradova
polina.vinogradova@iohk.io
Andre Knispel
andre.knispel@iohk.io

Abstract

This document presents modifications to the Shelley ledger specification (see Formal Methods Team, IOHK (2019)) that enable it to support native Multi-Assets (see Zahnentferner (2018)) using a small scripting language that is fully specified by the ledger rules.

List of Contributors

Alex Byaly, Manuel Chakravarty, Nicholas Clarke, Jared Corduan, Duncan Coutts, Matthias Güdemann, Kevin Hammond, Philipp Kant, Jann Mueller, Michael Peyton Jones, Tim Sheard

Contents

List of Figures

1 Introduction

This document gives a specification of the changes required to implement a both an Allegra-era and a Mary-era ledger. These ledgers share a specification (and can share an implementation), with the exception of a difference in one particular type. We abstract over this type and several functions specific to it, which allows us to write a single specification for both versions of the ledger, to which we refer as a *ShelleyMA ledger specification*.

All aspects of the ledger not given in this document are as specified in the Shelley ledger design and implementation Formal Methods Team, IOHK (2019). Only additions and replacements of definitions of Shelley types, functions, rules, and transitions are given here. Replacements are always given the same name as the original definition in the Shelley specification, eg. the UTXO rule in the Shelley specification is replaced with the UTXO rule given here.

1.1 ShelleyMA, Allegra and Mary Eras

The additional functionality that is introduced in the Allegra and Mary eras is as follows :

- *Allegra* : Timelock scripts replace multi-signature scripts as the native ledger-iterpreted script type. This type of script, in addition to the clauses of the m-of-n multi-signature language, has constructors that allow users to constrain the upper and lower slots of the validity of a clause of the script. For example, one may specify that up to a given slot s, either key k or k' can sign the transaction for the script to be valid, whereas from s onwards, only k's signature can make the script valid.
- *Mary* : Multi-asset support is added to the Allegra ledger, ie. to the Shelley ledger with timelock scripts. This includes changes to ledger accounting to accommodate multiple types of assets, as well as support for minting and burning of user-defined assets. This approach taken here was described initially in Plutus Team, IOHK (2020).

Adding both timelock script and multi-asset support is done in a single specification of the ShelleyMA ledger, as the transition rules are exactly the same for both Mary and Allegra. The difference between the two is that all accounting is done in terms of lovelace, ie. Coin, in the Allegra era, whereas the Mary era replaces the Coin type with a type the terms of which are bundles of arbitrary assets (user-defined ones, as well as lovelace), which we call Value.

In this specification, we represent all the places where the Coin type is used in Allegra, and the Value type in Mary using an algebraic structure called a Token algebra. We choose an abstract but fixed Token algebra **TA** for most of this specification. See Section 3 for details.

This

- makes this document is a specification for Allegra if **TA** = Coin, and
- makes it a specification for Mary if **TA** = Value.

Note that this specification is for two distinct eras, but we give only one translations, which is from Shelley to Mary, see Section 7. This is because no changes to chain or ledger types are required for Allegra.

2 Notation

In addition to the notation in the Shelley specification, we use the following notations:

Functions with finite support For a monoid M, $A \to_0 M$ are the functions $f : A \to M$ with finite support, i.e. such that there exist only finitely many $a \in A$ such that $fa \neq 0$. We denote the support of f by supp f, which is defined as supp $f := \{a \in A : fa \neq 0\}$.

3 Coin and Multi-Asset Token algebras

In this chapter we introduce the concept of a *Token algebra*, which is an abstraction used to generalize this specification to two different eras depending on the Token algebra chosen.

Definition 3.1 (Token algebra)**.** A *Token algebra* is a partially ordered commutative monoid T, written additively, (i.e. a commutative monoid together with a partial order, such that addition is monotonic in both variables) together with the functions and properties as described in Figure 1.

Functions

$\mathsf{coin} \in T \to \mathsf{Coin}$

Return the Ada contained inside the ValType element

$\mathsf{inject} \in \mathsf{Coin} \to T$

Create a ValType element containing only this amount of Ada

$\mathsf{policies} \in T \to \mathbb{P}\ \mathsf{PolicyID}$

The set of policy IDs required for minting

$\mathsf{size} \in T \to \mathsf{MemoryEstimate}$

Return the size, in words, of a ValType element

Properties

$$\mathsf{coin} \circ \mathsf{inject} = \mathsf{id}_{\mathsf{Coin}}$$

Figure 1: Additional functions and properties required for a Token algebra

A Token algebra is precisely the structure required to generalize the Coin type of the Shelley specification in transaction outputs for this ledger. We can then describe the ShelleyMA transaction processing rules without fixing a concrete Token algebra.

Depending on the Token algebra chosen, we obtain distinct ledgers. In particular, we get

- the Allegra ledger rules with Coin, and
- the Mary ledger rules with Value (defined below).

When multi-asset support on the ledger is introduced, Ada (Coin) will still be the most common type of asset on the ledger, as the ledger rules enforce that some quantity of it (specified via the coinsPerUTxOWord protocol parameter) must be contained in every UTxO on the ledger. It is the only type of asset used for all non-UTxO ledger accounting, including deposits, fees, rewards, treasury, and the proof of stake protocol. For this reason, not all occurrences of Coin inside a transaction or in the ledger state can or should be replaced by the chosen Token algebra.

Below we give the definitions of all the functions that must be defined on Coin and Value in order for them to have the structure of a Token algebra. In Section **??** we give several other types which we can meaningfully support the definition of the required functions (addition, size, etc.), including an optimized representation that more accurately represents the implementation used in the Haskell implementation of the multi-asset type. These types are convertible to and from the Value type, and appear in other parts of the system as a multi-asset representation which is more suitable in particular use cases.

3.1 Coin as a Token algebra

This section defines the Token algebra structure for the Coin type, see Figure 2. The structure of a partially ordered monoid is inherited from the (additive) integers.

For Coin, no policies are associated with Coin, since minting of Ada is not allowed.

$$\mathsf{coin} = \mathsf{id}_{\mathsf{Coin}}$$

$$\mathsf{inject} = \mathsf{id}_{\mathsf{Coin}}$$

$$\mathsf{policies}\ v = \emptyset$$

$$\mathsf{size}\ v = 0$$

Figure 2: The Token algebra structure of Coin

3.2 Multi-assets and the Token algebra Value

Elements of Value represent heterogeneous collections of assets, both user-defined and Ada. The Mary era ledger uses Value as its Token algebra in order to support multi-assets. Value and its Token algebra structure are given in Figure 3.

- PolicyID identifies monetary policies. A policy ID *pid* is associated with a script s such that hashScript $s = pid$. When a transaction attempts to create or destroy assets that fall under the policy ID *pid*, s verifies that the transaction respects the restrictions that are imposed by the monetary policy. See sections 4 and 5 for details.
- AssetName is a byte string used to distinguish different assets with the same PolicyID. Each *aname* identifies a particular kind of asset out of all the assets under the *pid* policy (but not necessarily among assets under other policies). The maximum length of this byte string is 32 bytes (this is not explicitly enforced in this specification).
- AssetID is either AdaID or a pair of a policy ID and an asset name. It is a unique and permanent identifier of an asset. That is, there are is no mechanism to change it or any part of it for any assets.

  Mary MA assets are fungible with each other if and only if they have to the same AssetID. The reason the unique identifier is a pair of two elements (except for the non-mintable Ada case) is to allow minting arbitrary collections of unique assets under a single policy.
- AdaID is a special asset ID for Ada, different than all other asset IDs. It is a term of the single-term type AdaIDType. It does not include a policy, so instead, the validation outcome in the presence of Ada in the mint field of the transaction is specified in the UTXO ledger rule. The rule disallows the mint field to contain Ada.
- Quantity is an integer type that represents an amount of a specific AssetName. We associate a $q \in$ Quantity with a specific asset to track how much of that asset is contained in a given asset value.
- Value is the multi-asset type that is used to represent a collection of assets, including Ada. This type is a finitely supported map.

  If *aid* is an AssetID and $v \in$ Value, the quantity of assets with that assed ID contained in v is v *aid*. Elements of Value are sometimes also referred to as *asset bundles*.

Derived types

$$
\begin{array}{rcll}
\mathit{aname} & \in & \mathsf{AssetName} & = & \mathsf{ByteString} \\
\mathit{pid} & \in & \mathsf{PolicyID} & = & \mathsf{ScriptHash} \\
\mathit{adaID} & \in & \mathsf{AdaIDType} & = & \{\ \mathsf{AdaID}\ \} \\
\mathit{aid} & \in & \mathsf{AssetID} & = & \mathsf{AdaIDType} \uplus (\mathsf{PolicyID} \times \mathsf{AssetName}) \\
\mathit{quan} & \in & \mathsf{Quantity} & = & \mathbb{Z} \\
v, w & \in & \mathsf{Value} & = & \mathsf{AssetID} \mapsto_0 \mathsf{Quantity}
\end{array}
$$

Token algebra structure of Value

$$\mathsf{coin}\ v = v\ \mathsf{AdaID}$$

$$\mathsf{inject}\ c = \mathsf{AdaID}\ \mapsto_0\ c$$

$$\mathsf{policies}\ v = \{\ \mathit{pid} \mid (\mathit{pid},\ \_) \in \mathsf{supp}\ v\ \}$$

size see Section B

Figure 3: Value and its Token algebra structure

$$v + w = \{\ \mathit{aid} \mapsto v\ \mathit{aid} + w\ \mathit{aid} \mid \mathit{aid} \in \mathsf{dom}\ v \cup \mathsf{dom}\ w\ \}$$
$$v \leq w \Leftrightarrow \forall\ \mathit{aid} \in \mathsf{AssetID}, v\ \mathit{aid} \leq w\ \mathit{aid}$$

Figure 4: Pointwise operations on Value

To give Value the structure of a partially ordered monoid, we define the required operations pointwise, in accordance with the usual definitions of these operations on finitely supported maps. See Figure 4.

3.3 Special Ada representation

Although all assets are native on the Cardano ledger (ie. the accounting and transfer logic for them is done directly by the ledger), Ada is still treated in a special way by the ledger rules, and is the most common type of asset on the ledger. It can be used for special purposes (such as fees) for which other assets cannot be used. The underlying consensus algorithm relies on Ada in a way that cannot be extended to user-defined assets. Ada can also neither be minted nor burned.

Note that in the Value definition above, we pick a special asset ID for Ada, that is not part of the type which represents the asset IDs for all other assets. Combining the asset name and policy for Ada gives a type-level guarantee that there is exactly one kind of asset that is associated with it in any way, rather than deriving this guarantee as an emergent property of minting rules.

Additionally, not giving Ada an actual policy ID (that could have a hash-associated policy) eliminates the possibility certain cryptographic attacks. We sometimes refer to Ada as the primary or principal currency. Ada does not, for the purposes of the Mary ledger specification, have a PolicyID or an AssetName.

3.4 Fixing a Token algebra

For the remainder of this specification, let **TA** be an arbitrary but fixed Token algebra. As described above, choosing Coin results in the Allegra ledger, and choosing Value results in the Mary ledger.

Abstract types

$$s_{v2} \in \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}} = \text{extended script language}$$

Derived types

$$\begin{aligned} \mathit{txout} &\in \mathsf{TxOut} = \mathsf{Addr} \times \mathbf{TA} \\ s &\in \mathsf{Script} = \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}} \end{aligned}$$

Transaction Type

$$\begin{aligned} \mathit{txbody} \in \mathsf{TxBody} = {} & \mathbb{P}\,\mathsf{TxIn} && \mathsf{txinputs} && \text{inputs} \\ & \times (\mathsf{Ix} \mapsto \mathsf{TxOut}) && \mathsf{txouts} && \text{outputs} \\ & \times \mathsf{DCert}^* && \mathsf{txcerts} && \text{certificates} \\ & \times \mathbf{TA} && \mathsf{mint} && \text{value minted} \\ & \times \mathsf{Coin} && \mathsf{txfee} && \text{non-script fee} \\ & \times \mathsf{Slot}^? \times \mathsf{Slot}^? && \mathsf{txvldt} && \text{validity interval} \\ & \times \mathsf{Wdrl} && \mathsf{txwdrls} && \text{reward withdrawals} \\ & \times \mathsf{Update}^? && \mathsf{txUpdates} && \text{update proposals} \\ & \times \mathsf{AuxiliaryDataHash}^? && \mathsf{txADhash} && \text{auxiliary data hash} \end{aligned}$$

$$\begin{aligned} \mathit{ad} \in \mathsf{AuxiliaryData} = {} & \mathbb{P}\,\mathsf{Script} && \mathsf{scripts} && \text{Optional scripts} \\ & \times \mathsf{Metadata}^? && \mathsf{md} && \text{metadata} \end{aligned}$$

$$\begin{aligned} \mathit{tx} \in \mathsf{Tx} = {} & \mathsf{TxBody} && \mathsf{txinputs} && \text{Transaction body} \\ & \times \mathsf{TxWitness} && \mathsf{txouts} && \text{Witnesses} \\ & \times \mathsf{AuxiliaryData}^? && \mathsf{txADhash} && \text{Auxiliary data} \end{aligned}$$

Accessor Functions

$$\begin{aligned} \mathsf{getValue} &\in \mathsf{TxOut} \to \mathbf{TA} && \text{output value} \\ \mathsf{getAddr} &\in \mathsf{TxOut} \to \mathsf{Addr} && \text{output address} \end{aligned}$$

Figure 5: Type Definitions used in the UTxO transition system

4 Transactions

This section describes the changes that are made to the transaction structure to support both timelock script and native multi-asset (MA) functionality in the Cardano ledger.

New Output Type

A key change needed to introduce MA functionality in the Mary era is changing the type of the transaction and UTxO outputs to contain multi-asset values to accommodate transacting with these types of assets natively, using the same ledger accounting scheme as is used for Ada. We set up the infrastructure for this change in transition system update common to both Allegra and Mary, ie. ShelleyMA. That is, TxOut now contains an element of **TA** (which could be a Coin, Value, or another type, see Section 3), rather than only Coin.

New Script Type

The multi-signature scripting language has been renamed to $\mathsf{Script}^{\mathsf{native}}_{\mathsf{v1}}$ and the type Script has been extended to include a new scripting language, $\mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$, that is backwards compatible with the multi signature scripting language, see Section 7. We specify the evaluation function for the new script type in Section A.

The new type of scripts can be used for all the same purposes, which means, for this specification, as

- output-locking scripts,
- certificate validation,
- reward withdrawals, or
- as minting scripts (see below).

The Mint Field

The body of a transaction in the ShelleyMA era contains one additional field, the mint field. The mint field contains an element of **TA**, which is used to specify the assets a transaction is putting into or taking out of circulation. Here, by "circulation", we mean specifically "the UTxO on the ledger". Since the administrative fields cannot contain assets other than Ada, and Ada cannot be minted (this is enforced by the UTxO rule, see Figure 7), they are not affected in any way by minting.

Putting assets into circulation is done with positive values in the Quantity fields mint field, and taking assets out of circulation can be done with negative quantities.

A transaction cannot simply mint arbitrary assets. In the Mary era, restrictions on multi-asset are imposed, for each asset with policy ID *pid*, by a script with the hash *pid*. Whether a minting transaction adheres to the restrictions prescribed by the preimage script is verified as part of the processing of the transaction. The minting mechanism is detailed in Section 5.

Note that the mint field exists in both Allegra and Mary eras, but can only be used in the Mary era, when multi-assets are introduced.

Transaction Body

The following changes were made to TxBody:

- a change in the type of TxOut — instead of Coin, the transaction outputs now have type **TA**.
- the addition of the mint field to the transaction body
- the time-to-live slot number (which had the accessor txttl), has been replaced with a validity interval with accessor txvldt, both endpoints of which are optional

The only change to the types related to transaction witnessing is the addition of minting policy scripts to the underlying Script type, so we do not include the whole Tx type here.

Auxiliary Data

The auxiliary data section of the transaction consists of two parts: md which is the same metadata type used in Shelley, and scripts, which is used to allow transactions to contain optional scripts that are not used for witnessing. The intended purpose of this field is to give users the ability to transmit a script that belongs to a script address via this auxiliary data. There are no restrictions on this field however, and users can use it to put arbitrary scripts into the auxiliary data.

5 UTxO

UTxO Helper Functions

Figure 6 defines additional calculations that are needed for the UTxO transition system with multi-assets:

- The MemoryEstimate type represents the size, in *words* (8 bytes), of a term (eg. a transaction input, or output). Note that this estimate is calculated according to fomulas we provide in this specification (see Appendix B), as opposed to obtained using an existing function of the programming language selected for implementation.
- The utxoEntrySize function provides a rough estimate of the amount of memory the storage of an output will consume, which is used in the UTXO rule. Its implementation is described in Appendix B.
- The function ininterval returns True whenever the given slot is inside the given interval. If an endpoint of the validity interval is $\diamond$, the comparison of the slot to that endpoint is True by default.
- The function getCoin returns the Ada in a given output as a Coin value.
- The ubalance function calculates the sum total in a given UTxO.
- The consumed and produced calculations are similar to their Shelley counterparts, with the following changes: 1) They return elements of **TA**, which the administrative fields of type Coin have to be converted to, via inject. 2) consumed also contains the mint field of the transaction. This is explained below.

Minting and the Preservation of Value

What does it mean to preserve the value of non-Ada assets, since they are put in and taken out of circulation by the users themselves? This question becomes relevant when we pick a Token algebra that can represent assets other than Ada.

If a transaction *tx* has an empty mint field, the preservation of value condition reduces to the same equality as the POV in Shelley. If a transaction mints assets, the value of the mint field has to reflect exactly the increase or decrease of the total quantities of the assets being minted or burned. Note that this means that the mint field can also contain negative quantities in the case assets are being burned.

To balance the preservation of value equation, the mint field could be included in either consumed or produced, with the only difference being the sign of the mint field. We include it on the consumed side, because this means that minting a positive quantity increases the total quantity of assets on the ledger, and minting a negative quantity reduces the total quantity of assets on the ledger.

Note that the UTXO rule specifically forbids the minting of Ada, and thus in the case of Ada, the preservation of value equation is exactly the same as in Shelley.

The minting scripts themselves are not evaluated at part of the UTXO, but instead as part of witnessing, i.e. in the UTXOW rule, see Figure 8.

The UTXO Transition Rule

In Figure 7, we give the UTXO transition rule for ShelleyMA. There are the following changes to the preconditions of this rule as compared to the original Shelley UTXO rule:

- The check that the time-to-live of a transaction is after the current slot is replaced with the check that the current slot is inside the validity interval
- In the preservation of value calculation (which looks the same as in Shelley), the value in the mint field is taken into account.
- The transaction is not minting any Ada. This condition, in the Allegra era, ensures that no assets can ever be minted (since Ada is the only existing asset).
- The **TA** element in each output is constrained from below by a **TA** element that contains some amount of Ada and no other assets. The amount of Ada that is contained in this element (and therefore, the output element which it constrains) depends on the size of the output. To get the minimum Ada amount, the size of the output is multiplied by the function *coinsPerUTxOWord* applied to protocol parameters (see Section B for the function definition). Note that this check implies that no quantity of any asset appearing in that output can be negative.

  In the case that **TA** $=$ Coin, this constraint exactly mimics the minUTxOValue constraint in Shelley, see B.2.
- The serialized size of the **TA** element in each output is no greater than MaxValSize (specified in Section B). This ensures that each individual output is never so large that any transaction carrying all the witness data (eg. large scripts, etc.) necessary for spending such an output will exceed the transaction size limit. See Section B for details. Note that this precondition can be assumed to hold for the Token algebra Coin, as it is a single integer.

Note that updating the UTxO with the inputs and the outputs of the transaction looks the same as in the Shelley rule. There is a type-level difference however, as the outputs of a transaction contain an element of **TA**, rather than Coin.

Witnessing

Figure 8 contains the changed definition of the function scriptsNeeded, which also collects the scripts necessary for minting. Recall that in Allegra, no assets can ever be minted, so there are no scripts to run that validate minting.

The witnessing rule UTXOW only needs a minor change for verifying the metadata hash, which is replacing the condition $md = \diamond$ with $md = (\emptyset, \diamond)$.

Type Definition

$$mest \in \mathsf{MemoryEstimate} = \mathbb{N}$$

Abstract Helper Function

$\mathsf{utxoEntrySize} \in \mathsf{TxOut} \to \mathsf{MemoryEstimate}$ Memory estimate for TxOut

Helper Functions

$\mathsf{ininterval} \in \mathsf{Slot} \to (\mathsf{Slot}^? \times \mathsf{Slot}^?) \to \mathsf{Bool}$

$$\mathsf{ininterval}\ slot\ (i_s, i_f) = \begin{cases} \mathsf{True} & (i_s = \diamond) \wedge (i_f = \diamond) \\ slot < i_f & (i_s = \diamond) \wedge (i_f \neq \diamond) \\ i_s \leq slot & (i_s \neq \diamond) \wedge (i_f = \diamond) \\ i_s \leq slot < i_f & (i_s \neq \diamond) \wedge (i_f \neq \diamond) \end{cases}$$

$\mathsf{getCoin} \in \mathsf{TxOut} \to \mathsf{Coin}$
$\mathsf{getCoin}\ (\_, out) = \mathsf{coin}\ out$

$\mathsf{ubalance} \in \mathsf{UTxO} \to \mathbf{TA}$
$$\mathsf{ubalance}\ utxo = \sum_{\_ \mapsto u \in utxo} \mathsf{getValue}\ u$$

$\mathsf{isAdaOnly} \in \mathsf{ValType} \to \mathsf{Bool}$
$\mathsf{isAdaOnly}\ v = v \in \mathrm{range}\ \mathsf{inject}$

$\mathsf{scaledMinDeposit} \in \mathbf{TA} \to \mathsf{Coin} \to \mathsf{Coin}$
$$\mathsf{scaledMinDeposit}\ v\ mv = \begin{cases} mv & \mathsf{isAdaOnly}\ v \\ \max\ (mv,\ \mathsf{utxoEntrySize}\ v * \mathsf{coinsPerUTxOWord}\ mv)) & \text{otherwise} \end{cases}$$

Produced and Consumed Calculations

$\mathsf{consumed} \in \mathsf{PParams} \to \mathsf{UTxO} \to \mathsf{TxBody} \to \mathbf{TA}$
$\mathsf{consumed}\ pp\ utxo\ txb =$
$\mathsf{ubalance}\ (\mathsf{txins}\ txb \lhd utxo) + \mathsf{mint}\ txb$
$+ \mathsf{inject}\ (\mathsf{wbalance}\ (\mathsf{txwdrls}\ txb) + \mathsf{keyRefunds}\ pp\ txb)$

$\mathsf{produced} \in \mathsf{PParams} \to \mathsf{StakePools} \to \mathsf{TxBody} \to \mathbf{TA}$
$\mathsf{produced}\ pp\ stpools\ txb =$
$\mathsf{ubalance}\ (\mathsf{outs}\ txb)$
$+ \mathsf{inject}\ (\mathsf{txfee}\ txb + \mathsf{totalDeposits}\ pp\ stpools\ (\mathsf{txcerts}\ txb))$

Figure 6: UTxO Calculations

$$txb := \text{txbody } tx \qquad \text{ininterval } slot\ (\text{txvldt } tx)$$
$$\text{txins } txb \neq \emptyset \qquad \text{minfee } pp\ tx \leq \text{txfee } txb \qquad \text{txins } txb \subseteq \text{dom } utxo$$
$$\text{consumed } pp\ utxo\ txb = \text{produced } pp\ stpools\ txb$$

$$\begin{matrix} slot \\ pp \\ genDelegs \end{matrix} \vdash pup \xrightarrow[\text{PPUP}]{\text{txup } tx} pup'$$

$$\text{coin } (\text{mint } txb) = 0$$

$$\forall txout \in \text{txouts } txb,$$
$$\text{getValue } txout \geq \text{inject } (\text{scaledMinDeposit } v\ (\text{minUTxOValue } pp))$$

$$\forall txout \in \text{txouts } txb,$$
$$\text{serSize } (\text{getValue } txout) \leq \text{MaxValSize}$$

$$\forall (\_ \mapsto (a, \_)) \in \text{txouts } txb, a \in \text{Addr}_{\text{bootstrap}} \Rightarrow \text{bootstrapAttrsSize } a \leq 64$$
$$\forall (\_ \mapsto (a, \_)) \in \text{txouts } txb, \text{netId } a = \text{NetworkId}$$
$$\forall (a \mapsto \_) \in \text{txwdrls } txb, \text{netId } a = \text{NetworkId}$$
$$\text{txsize } tx \leq \text{maxTxSize } pp$$

$$\text{UTxO-inductive} \frac{\begin{matrix} refunded := \text{keyRefunds } pp\ txb \\ depositChange := \text{totalDeposits } pp\ stpools\ (\text{txcerts } txb) - refunded \end{matrix}}{\begin{matrix} slot \\ pp \\ stpools \\ genDelegs \end{matrix} \vdash \begin{pmatrix} utxo \\ deposits \\ fees \\ pup \end{pmatrix} \xrightarrow[\text{UTXO}]{tx} \begin{pmatrix} (\text{txins } txb \lhd utxo) \cup \text{outs } txb \\ deposits + depositChange \\ fees + \text{txfee } txb \\ pup' \end{pmatrix}} \quad (1)$$

Figure 7: UTxO inference rules

$$\text{scriptsNeeded} \in \text{UTxO} \to \text{Tx} \to \mathbb{P}\ \text{ScriptHash} \qquad \text{required script hashes}$$
$$\text{scriptsNeeded } utxo\ tx =$$
$$\{\text{validatorHash } a \mid i \mapsto (a, \_) \in utxo,$$
$$i \in \text{txinsScript } (\text{txins } txb)\ utxo\}$$
$$\cup\ \{\text{stakeCred}_r\ a \mid a \in \text{dom}(\text{Addr}^{\text{script}}_{\text{rwd}} \lhd \text{txwdrls } txb)\}$$
$$\cup\ (\text{Addr}^{\text{script}} \cap \text{certWitsNeeded } txb)$$
$$\cup\ \text{policies } (\text{mint } txb)$$
$$\textbf{where}$$
$$txb = \text{txbody } tx$$

Figure 8: Scripts Needed

6 Rewards and the Epoch Boundary

In order to handle rewards and staking, we must change the stake distribution calculation function to add up only the Ada in the UTxO before performing any calculations. In Figure 9 below, we do so using the function utxoAda, which returns the amount of Ada in an address.

Helper functions

$$\begin{aligned}&\mathsf{utxoAda} \in \mathsf{UTxO} \to \mathsf{Addr} \to \mathsf{Coin}\\ &\mathsf{utxoAda}\ \mathit{utxo}\ \mathit{addr} = \sum_{\mathit{out} \in \mathsf{range}\ \mathit{utxo}, \mathsf{getAddr}\ \mathit{out} = \mathit{addr}} \mathsf{getCoin}\ \mathit{out}\end{aligned}$$

Stake Distribution (using functions and maps as relations)

$$\begin{aligned}&\mathsf{stakeDistr} \in \mathsf{UTxO} \to \mathsf{DState} \to \mathsf{PState} \to \mathsf{Snapshot}\\ &\mathsf{stakeDistr}\ \mathit{utxo}\ \mathit{dstate}\ \mathit{pstate} =\\ &\quad \left((\mathrm{dom}\ \mathit{activeDelegs}) \lhd (\mathsf{aggregate}_+\ \mathit{stakeRelation})\right),\ \mathit{delegations},\ \mathit{poolParams}\right)\\ &\textbf{where}\\ &\quad (\ \mathit{rewards},\ \mathit{delegations},\ \mathit{ptrs},\ \_,\ \_,\ \_) = \mathit{dstate}\\ &\quad (\ \mathit{poolParams},\ \_,\ \_) = \mathit{pstate}\\ &\quad \mathit{stakeRelation} = \left(\left(\mathsf{stakeCred_b}^{-1} \cup (\mathsf{addrPtr} \circ \mathit{ptr})^{-1}\right) \circ \left(\mathsf{utxoAda}\ \mathit{utxo}\right)\right) \cup \mathit{rewards}\\ &\quad \mathit{activeDelegs} = (\mathrm{dom}\ \mathit{rewards}) \lhd \mathit{delegations} \rhd (\mathrm{dom}\ \mathit{poolParams})\end{aligned}$$

Figure 9: Stake Distribution Function

7 Blockchain layer

No translation between Shelley and Allegra chain state types are required to transition between the eras. A Shelley Tx can be decoded as an Allegra one, including decoding a multi-sig script as a timelock one (the difference between them is that timelock scripts have additional constructurs). The one type that changes during era transition is only part of the transaction, but does not appear in the chain state.

In Figure 10, we give the function translateEra that specifies the translation of an Allegra-era chain state into a chain state that provides multi-asset support (the Mary era). We use $\mathsf{ChainState}_{\mathsf{Allegra}}$ to denote the type of the chain state in the Allegra era, and ChainState for the Mary era chain state. We use the notation $chainstate_x$ to represent variable x in the chain state. We do not specify the variables that remain unchanged during the transition. The only part of the state that is affected by the era transition is the UTxO, as each UTxO map entry must be updated to contain a Value instead of Coin.

$$\mathsf{translateEra} \in \mathsf{ChainState}_{\mathsf{Allegra}} \to \mathsf{ChainState}$$
$$\mathsf{translateEra}\ (chainstate_{utxo}) = \{\ txin \mapsto (a, \mathsf{inject}\ c) \mid txin \mapsto (a, c) \in utxo\ \}$$

Figure 10: Allegra to Multi-Asset Chain State Transtition

8 Properties

The properties of this section are modifications and additions to the properties of the Shelley ledger. See Formal Methods Team, IOHK (2019) for definitions used here. We need to amend the definition of Val to the following:

$$\mathsf{Val}(x \in \mathbf{TA}) = x$$
$$\mathsf{Val}(x \in \mathsf{Coin}) = \mathsf{inject}\ x$$
$$\mathsf{Val}((\_ \mapsto (y \in \mathbf{TA}))^*) = \sum y$$

Lemma 8.1. For all environments e, transactions t, and states s, s', if

$$e \vdash s \xrightarrow[\text{UTXO}]{t} s'$$

then

$$\mathsf{Val}(s) + wm = \mathsf{Val}(s')$$

where $wm = \mathsf{inject}(\mathsf{wbalance}\ (\mathsf{txwdrls}\ t)) + \mathsf{mint}\ (\mathsf{txbody}\ t)$.

Proof. The proof is identical to the corresponding one in the previous specification, except that unfolding consumed gives an additional mint txb term and all Coin quantities have to be converted to terms of type **TA**. □

We also need to track the sum of the mint fields of all transactions in a block, for which we write $\mathsf{mint}(b)$.

Theorem 8.2 (Preservation of Value)**.** For all environments e, blocks b, and states s, s', if

$$e \vdash s \xrightarrow[\text{CHAIN}]{b} s'$$

then

$$\mathsf{Val}(s) + \mathsf{mint}(b) = \mathsf{Val}(s')$$

Proof. Similar to the corresponding proof in the Shelley specification, for a given x and transition TR, let $\mathsf{PresOfVal}(x, TR, \mathsf{b})$e the statement:

$$\text{for all environments } e, \text{ signals } \sigma, \text{ and states } s, s',\ e \vdash s \xrightarrow[\text{TR}]{\sigma} s' \implies \mathsf{Val}(s) + x = \mathsf{Val}(s').$$

Then, $\mathsf{PresOfVal}(\mathsf{mint}\ txb, \mathsf{LEDGER})$ follows from Lemma 8.1. By induction, we have $\mathsf{PresOfVal}(\mathsf{mint}(b), \mathsf{LEDGERS})$. The rest of the proof works as previously, using $\mathsf{PresOfVal}(0, \mathsf{TR})$ or $\mathsf{PresOfVal}(\mathsf{mint}(b), \mathsf{TR})$ as needed. □

Theorem 8.3 (Preservation of Ada)**.** For all environments e, blocks b, and states s, s', if

$$e \vdash s \xrightarrow[\text{CHAIN}]{b} s'$$

then

$$\mathsf{coin}(\mathsf{Val}(s)) = \mathsf{coin}(\mathsf{Val}(s'))$$

Proof. The hypothesis implies that for all transactions tx included in b, there exist some e_{tx}, s_{tx} and s'_{tx} such that $e_{tx} \vdash s_{tx} \xrightarrow[\text{UTXO}]{tx} s'_{tx}$, so $\mathsf{coin}(\mathsf{mint}\ tx) = 0$ for all transactions included in b. This implies $\mathsf{coin}(\mathsf{mint}(b)) = 0$, so the claim follows by Lemma 8.2. □

References

Formal Methods Team, IOHK. A Formal Specification of the Cardano Ledger, 2019. URL https://github.com/input-output-hk/cardano-ledger/tree/master/eras/shelley/formal-spec/ledger-spec.tex.

Plutus Team, IOHK. $UTXO_{ma}$: UTXO with Multi-Asset Support, 2020.

Joachim Zahnentferner. Multi-currency ledgers. *??*, 2018.

A Script Constructors and Evaluation

$$
\begin{aligned}
&\mathsf{validateScript} \in \mathsf{Script} \to \mathsf{Tx} \to \mathsf{Bool} \\
&\mathsf{validateScript}\ s\ tx\ =\ \mathsf{evalTimelock}\ vhks\ itr\ s \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad vhks := \{\mathsf{hashKey}\ vk | vk \in \mathrm{dom}(\mathsf{txwitsVKey}\ tx)\} \\
&\qquad\qquad itr := \mathsf{txvldt}\ (\mathsf{txbody}\ \ tx)
\end{aligned}
$$

Figure 11: Script Validation

In the ShelleyMA era, the validateScript function performs validation of only timelock scripts, see Figure 11. Note that while there is no explicit support for validating multi-signature scripts in validateScript, these scripts remain effectively usable as the encodings of multi-sig and timelock scripts are compatible. An existing multi-sig script-locked output is spent by interpreting the encoded script as a timelock script (one without any validity interval clauses), then validating it. For details, see the CDDL specification.

The arguments that are passed to the validateScript function include all those that are needed for $\mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$ script evaluation :

- The script getting evaluated
- The transaction

The semantics of the Timelock language are specified in Figure 12. The evaluation of the scripts constructed by Signature, AllOf, AnyOf, MOfN is done the same as for their $\mathsf{Script}^{\mathsf{native}}_{\mathsf{v1}}$ counterparts. The RequireTimeStart evaluation checks that the start of the transaction validity interval is not $\diamond$, and is in or after the slot specified by the constructor. The RequireTimeExpire evaluation checks that the end of the transaction validity interval is not $\diamond$, and is in or before the slot specified by the constructor.

Timelock Script Constructor Types

evalTimelock	$\in \mathbb{P}\ \mathsf{KeyHash} \to (\mathsf{Slot}^? \times \mathsf{Slot}^?) \to \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}} \to \mathsf{Bool}$

The type of the Timelock script evaluator

Signature	$\in \mathsf{KeyHash} \to \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$
AllOf	$\in \mathsf{Script}^{\mathsf{native}*}_{\mathsf{v2}} \to \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$
AnyOf	$\in \mathsf{Script}^{\mathsf{native}*}_{\mathsf{v2}} \to \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$
MOfN	$\in \mathbb{N} \to \mathsf{Script}^{\mathsf{native}*}_{\mathsf{v2}} \to \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$
RequireTimeStart	$\in \mathsf{Slot} \to \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$
RequireTimeExpire	$\in \mathsf{Slot} \to \mathsf{Script}^{\mathsf{native}}_{\mathsf{v2}}$

Timelock Script Evaluation

$$\mathsf{evalTimelock}\ (\mathsf{Signature}\ hk)\ vhks\ \_ = hk \in vhks$$

$$\mathsf{evalTimelock}\ (\mathsf{AllOf}\ ts)\ vhks\ \_ = \forall t \in ts : \mathsf{evalTimelock}\ t\ vhks$$

$$\mathsf{evalTimelock}\ (\mathsf{AnyOf}\ ts)\ vhks\ \_ = \exists t \in ts : \mathsf{evalTimelock}\ t\ vhks$$

$$\mathsf{evalTimelock}\ (\mathsf{MOfN}\ m\ ts)\ vhks\ \_ = m \leq \Sigma(\mathrm{card}\{ts.t.t \leftarrow ts \wedge \mathsf{evalTimelock}\ t\ vhks\})$$

$$\mathsf{evalTimelock}\ (\mathsf{RequireTimeStart}\ lockStart)\ vhks\ (txStart, \_) = \begin{cases} \mathsf{False} & txStart = \diamond \\ lockStart \leq txStart & \text{otherwise} \end{cases}$$

$$\mathsf{evalTimelock}\ (\mathsf{RequireTimeExpire}\ lockExp)\ vhks\ (\_, txExp) = \begin{cases} \mathsf{False} & txExp = \diamond \\ txExp \leq lockExp & \text{otherwise} \end{cases}$$

Figure 12: Timelock Script Constructor Types and Evaluation

B Output Size

In Shelley, the protocol parameter *minUTxOValue* is used to disincentivize attacking nodes by putting many small outputs on the UTxO, permanently blocking memory. In the Mary era and onwards, the **TA** is Value, rather than Coin (as in Allegra). Elements of Value can potentially be arbitrarily large, so the ledger requires each UTxO entry to contain a minimum amount of Ada, proportional to its size.

There is also another Value_C size consideration with respect to spendability of an output. The restriction on the total serialized size of the transaction (set by the parameter *maxTxSize*) serves as an implicit upper bound on the size of a Value_C contained in an output of a transaction. Without tighter limits on the output Value_C size, one of the following situations could arise, causing the output to be come unspendable (these are just a two examples) :

- The script locking the very large Value_C-containing UTxO is too large to fit inside the transaction alongside the Value_C itself while still respecting the max transaction size
- The large Value_C cannot be split into several outputs, because the outputs are impossible to fit inside a single transaction

The same considerations apply for any underlying **TA** we choose to fix. In the ShelleyMA eras, the two types that are used to define concrete ledgers are Coin and Value_C. The size calculations for Coin, in practice, result in either trivial restrictions in the ledger rules, or ones that align with Shelley (as discussed in Section 5).

B.1 Value Size

Figure 13 contains abstract and helper functions used in calculating the in-memory and serialized representation sizes of Value_C elements.

For a serializable type, the function serialize is defined on that type. We do not specify serialization functions here, but the CDDL specification gives the encoding details (see **?**). We give this function here because it is the first time it is used in the specification, but it is not specific to multi-assets, Mary, or Allegra in any way.

The function serSize returns the actual number of bytes in the bytestring that is the serialized representation of a **TA** element. The specific underlying **TA** is required to be serializable in every era.

The serSize function is used constrain the serialized representation of the transaction (in particular, the size of Value_C elements in outputs), whereas the min-Ada requirement is a calculation based on the in-memory representation size. A transparently-calculated size estimate is not necessary for limiting the size of values in outputs, since this size-bound check does not place any additional accounting/monetary constranits on transaction construction, unlike the min-Ada requirement.

B.2 Min UTxO Value

Figure 14 gives the types of constants used in the estimation of the size of a UTxO entry, and the associated min-Ada-value.

The size function returns the estimated size of a Value_C element. The size function on Value is defined via the isomorphism in Section **??**,

$$\mathsf{size}_{\mathsf{Value}}\ v = \mathsf{size}\ (\mathsf{iso}_v\ v)$$

The size of a Value_C element is constant in the case when it contains only Ada. If there are other types of assets contained in it, the size depends on

Abstract Functions

$\mathsf{serialize} \in \_ \to \mathsf{ByteString}$
Serialization function on an arbitrary (serializable) type

$\mathsf{anameLen} \in \mathsf{AssetName} \to \mathsf{MemoryEstimate}$
Returns the length (in bytes) of an asset name

Helper Functions

$\mathsf{serSize} \in \mathbf{TA} \to \mathsf{MemoryEstimate}$
$\mathsf{serSize}\ v = |\mathsf{serialize}\ v|$
Gives the size of the serialized representation of a **TA**

$\mathsf{numAssets} \in \mathsf{Value}_C \to \mathbb{N}$
$\mathsf{numAssets}\ vl = |\{\ (pid, an) \mid pid \mapsto (an \mapsto \_) \in vl\ \}|$
Returns the number of distinct asset IDs in a Value_C

$\mathsf{sumALs} \in \mathsf{Value}_C \to \mathbb{N}$
$\mathsf{sumALs}\ vl = \sum_{\{\ an \mid \_ \mapsto (an \mapsto \_) \in vl\ \}} \mathsf{anameLen}\ an$
Returns the sum of the lengths (in bytes) of distinct asset names in a Value_C

$\mathsf{numPids} \in \mathsf{Value}_C \to \mathbb{N}$
$\mathsf{numPids}\ vl = |\mathsf{pids}\ vl|$
The number of policy IDs in a Value_C

Figure 13: Value Size Helper Functions

- the number of distinct asset types (asset IDs)
- the number of distinct policy IDs, and
- the sum of the lengths of distinct asset names.

The parameter minUTxOValue specifies the min-Ada value for a UTxO containing only Ada. This type of UTxO varies in size somewhat (eg. Byron style addresses may be a different length than Shelley ones), but we estimate the size of the most commonly used type of Ada-only UTxO as the constant value adaOnlyUTxOSize. This constant is in fact an upper bound on UTxOs which have have only Shelley credentials.

We use this size estimate to calculate what minUTxOValue implies to be the min-Ada value requirement *per word* of UTxO data. The function coinsPerUTxOWord performs this calculation by dividing the min-Ada value by the Ada-only UTxO size (and taking the floor).

The utxoEntrySizeWithoutVal is the constant representing the size of a UTxO entry, not counting the size of the **TA** element it contains. Here, again, the actual size of a UTxO (excluding the **TA** element) can vary, but we use an upper bound on the size of Shelley-credential UTxOs.

The function utxoEntrySize estimates the size of an arbitrary ShelleyMA-era UTxOs. It adds the size estimate of the **TA** element in a UTxO and the utxoEntrySizeWithoutVal constant.

The constants used in the implementation of the ShelleyMA eras are as follows :

Constants

$$
\begin{array}{lcl}
(k_0, k_1, k_2, k_3, k_4) & \in & \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
\mathsf{UtxoEntrySizeWithoutVal} & \in & \mathsf{MemoryEstimate} \\
\mathsf{AdaOnlyUTxOSize} & \in & \mathsf{MemoryEstimate} \\
\mathsf{MaxValSize} & \in & \mathsf{MemoryEstimate}
\end{array}
$$

Size and Min-Ada Functions

$$\mathsf{size} \in \mathsf{Value}_C \to \mathsf{MemoryEstimate}$$

$$\mathsf{size}\ vl = \begin{cases} k_0 & \mathsf{isAdaOnly}\ vl \\ k_1 + \lfloor\ \mathsf{numAssets}\ vl * k_2 + \mathsf{sumALs}\ vl & \\ \quad + \mathsf{numPids}\ vl * k_3 + k_4 - 1 /\ k_4\ \rfloor & \text{otherwise} \end{cases}$$

Calculate the size of a Value_C

$$\mathsf{coinsPerUTxOWord} \in \mathsf{Coin} \to \mathsf{Coin}$$

$$\mathsf{coinsPerUTxOWord}\ mv = \lfloor\ mv\ /\ \mathsf{adaOnlyUTxOSize}\ \rfloor$$

Calculate the cost of storing a memory unit of data as a UTxO entry

$$\mathsf{utxoEntrySize} \in \mathbf{TA} \to \mathsf{MemoryEstimate}$$

$$\mathsf{utxoEntrySize}\ v = \mathsf{utxoEntrySizeWithoutVal} + \mathsf{size}\ v$$

Calculate the size of a UTxO entry

Figure 14: Value Size Calculation

- $(k_0, k_1, k_2, k_3, k_4) = (1, 6, 12, 28, 8)$
- $\mathsf{utxoEntrySizeWithoutVal} = 27$ words (8 bytes)
- $\mathsf{adaOnlyUTxOSize} = 27$ words (8 bytes)
- $\mathsf{MaxValSize} = 4000$ bytes, ie. 500 words.