A Formal Specification of the Cardano Ledger integrating Plutus Core

Deliverable GL-D2

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

Project: Goguen Ledger

Type: Deliverable
Due Date: 28th February 2021

Responsible team: Formal Methods Team
Editor: Andre Knispel, IOHK
Team Leader: Philipp Kant, IOHK

Version FS-2.2.0

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

Formal Specification of the Cardano Ledger with Plutus Integration

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

Abstract

This document presents modifications to the Shelley ledger specification Formal Methods Team, IOHK (2019c) that will enable it to support the requirements for Plutus Foundation, within the context of the Goguen era of Cardano. This specification is based on the Plutus system and the extended UTxO specification of the mockchain on which it operates, as outlined in Chakravarty et al. (2020). We present a unified way to process both Plutus scripts and Shelley-style multi-signature scripts (see Formal Methods Team, IOHK (2019a)).

List of Contributors

Duncan Coutts, Philipp Kant, Michal Peyton Jones, Jann Mueller, Jared Corduan, Matthias Gudemann, Manuel Chakravarty, Kevin Hammond, Tim Sheard, Nicholas Clarke, Alex Byaly, Yun Lu

Contents

List of Figures

1 Introduction

This document describes the extensions to the multi-asset formal ledger specification Formal Methods Team, IOHK (2019b) that are required for the support of phase two scripts, in particular Plutus Core. This underpins future Plutus development: there should be minimal changes to these ledger rules to support future phase-2 languages (eg. upcoming versions of Plutus). The two major extensions that are described here are:

1. the introduction of *phase-2* scripts, i.e. scripts that are not evaluated internally by the ledger; and 2. additions to the Shelley-era UTxO (unspent transaction output) model that are needed to support Plutus constructs (the "extended UTxO" model).

This document defines these extensions as changes to the multi-asset structured transition system, using the same notation and conventions that were used for the multi-asset specification Formal Methods Team, IOHK (2019b). As with the multi-asset formal specification, these rules will be implemented in the form of an executable ledger specification that will then be integrated with the Cardano node.

1.1 Phase Two Scripts

The Shelley formal specification introduced the concept of "multi-signature" scripts. *Phase one scripts*, such as these, are captured entirely by the ledger rules. Execution costs can therefore be easily assessed *before* they are processed by the implementation, and any fees can be calculated directly within the ledger rule implementation, based on e.g. the size of the transaction that includes the script.

In contrast, *phase-2* scripts can perform arbitrary (and, in principle, Turing-complete) computations. We require transactions that use phase-2 scripts to have a budget in terms of a number of abstract ExUnits. This budget gives a quantitative bound on resource usage in terms of a number of specific metrics, including memory usage or abstract execution steps. The budget is then used as part of the transaction fee calculation.

Every phase-2 scripting language converts the calculated execution cost into a number of ExUnits using a cost model, CostModel, which depends on the language and is provided as a protocol parameter. This allows execution costs (and so transaction fees) to be varied without requiring a major protocol version change ("hard fork"). This may be used, for example, if a more efficient interpreter is produced.

The approach we use to bound execution costs to a pre-determined constant is distinct from the usual "gas" model in the following notable ways :

- The budget to run a script is expressed in terms of computational resources, and included in the transaction data. The *exact* resource budget required can be computed before submitting a transaction, since script execution is deterministic. See Section 2.2. In the gas model, the budget is expressed indirectly as an upper bound on the fee the submitter is willing to pay for execution of the contract (eg. their total available funds).
- The user specifies the UTxO entries containing funds sufficient to cover a percentage (usually 100 or more) of the total transaction fee. These inputs are only collected *in the case of script validation failure*, and are called *collateral inputs*. In the case of script validation success, the fee specified in the fee field of the transaction is collected, but the collateral is not.

  This scheme is different from the gas model in that the exact payment collected in both the case of script validation success and validation failure is known ahead of time, though the collected fees are different in the two cases.

Another important point to make about both phase one and two scripts on Cardano is that running scripts in all languages will be supported indefinitely whenever possible. Making it

impossible to run a script in a particular scripting language makes UTxO entries locked by that script unspendable.

We use the terms Plutus, PlutusV1, and "phase two scripting language" in this specification somewhat interchangeably. The reason for this is that while we intend for the infrastructure set up in this document to be somewhat language-agnostic (in particular, able to support multiple versions of Plutus), it only gives all the details for the first and (currently only) phase-2 script language, PlutusV1, the introduction of which represents the start of the Alonzo era.

1.2 Extended UTxO

The specification of the extended UTxO model follows the description that was given in Chakravarty et al. (2020). All transaction outputs that are locked by phase-2 scripts must include the hash of an additional "datum". The actual datum needs to be supplied by the transaction spending that output, and can be used to encode state, for example. While this datum could instead have been stored directly in the UTxO, our choice of storing it in the transaction body improves space efficiency in the implementation by reducing the UTxO storage requirements. The datum is passed to a script that validates that the output is spent correctly.

All transactions also need to supply a *redeemer* for all items that are validated by a script. This is an additional piece of data that is passed to the script, and that could be considered as a form of user-input to the script. Note that the same script could be used for multiple different purposes in the same transaction, so in general it might be necessary to include more than one redeemer per script. There will, however, always be at least one redeemer per script.

2 Language Versions and Cost Models

We require the following types (see Figure 1) in addition to those that are already defined in the Shelley specification Formal Methods Team, IOHK (2019c).

ExUnits	ExUnits is made up of two integer values, representing abstract notions of the relative memory usage and script execution steps respectively. We give it the structure of a partially ordered monoid via its product structure, i.e. addition and comparisons are defined point-wise.
Prices	Prices consists of two rational numbers that correspond to the components of ExUnits: the price per unit of memory and price per reduction step.
CostModel	A cost model is a vector of coefficients that are used to compute the execution units required to execute a script. Its specifics depend on specific versions of the Plutus interpreter it is used with. We keep this type as abstract in the specification, see Plutus Team (2021a) and Plutus Team (2021b) for details.
Language	This represents phase-2 language types. Currently there is only PlutusV1.
LangDepView	A pair of two byte strings, where the first represents the serialized language tag (eg. the tag for PlutusV1), and the second, the protocol parameters that must be fixed (by the transaction) when carrying a phase-2 script in that language.

The function serialize is left abstract, as these is an implementation-dependent serialization function. It must be implemented for all serializable types, including all collections of protocol parameters needed to construct a LangDepView for each existing language.

2.1 Language Versions and Backwards Compatibility Requirements

In the Language type, each *version* of a language is considered to be a different language (so there might be several versions of the Plutus language, each of which would be considered to be different). Each such language needs to be interpreted by a language-specific interpreter that is called from the ledger implementation. The interpreter is provided with the (language- and version-specific) arguments that it requires. It is necessary for the ledger to be capable of executing scripts in all languages it has ever supported. This implies that it is necessary to maintain all forms of ledger data that is needed by any past or current language, which constrains future ledger designs. Introducing a new language will require a protocol version update, since the datatypes need to support the new language and the ledger rules must be updated to use the new interpreter.

2.2 Determinism of Script Evaluation

The data that is passed to the interpreter includes the validator script, the redeemer, possibly a datum from the UTxO, information about the transaction that embeds the script, any relevant ledger data, and any relevant protocol parameters. It is necessary for the validation outcome of any scripts to remain the same during the entire period between transaction submission and completion of the script processing. In order to achieve this, any data that is passed to the interpreter must be determined by the transaction body. In property C.8, we make precise what deterministic script evaluation means. The transaction body therefore includes a hash of any such data that is not uniquely determined by other parts of the transaction body or the UTXO. When the transaction is processed, as part of the UTXOW rule, this hash is compared with a hash of the data that is passed to the interpreter. This ensures that scripts are only executed if they have been provided with the intended data.

The getLanguageView function is used in the computation of this integrity hash. Shown in (Figure 1), it filters the data relvant to a given language from the protocol parameters, and

returns a pair of byte strings : the serialized representation of the language tag, and the serialized representation of the relevant data in the protocol parameters, which, for PlutusV1, is just its cost model, costmdls pp {PlutusV1}. This function must be defined for all existing script languages that require a cost model (are phase-2), which, in Alonzo, is only PlutusV1. The only relevant parameter for PlutusV1 is the cost model for this language.

2.3 Script Evaluation Cost Model and Prices

To convert resource primitives into the more abstract ExUnits during script execution a cost model needs to be supplied to the interpreter. The cost models required for this purpose are recorded in the *costmdls* protocol parameter. The calculation of the actual cost, in Ada, of running a script that takes *exunits* $\in$ ExUnits resources to run, is done by a formula in the ledger rules, which uses the *prices* parameter. This is a parameter that applies to all scripts and that cannot be varied for individual languages. This parameter reflects the real-world costs in terms of energy usage, hardware resources etc.

In Alonzo, the protocol parameter *minUTxOValue* is deprecated, and replaced by *coinsPerUTxOWord*. This specifies directly the deposit required for storing bytes of data on the ledger in the form of UTxO entries.

Limiting Script Execution Costs. The *maxTxExUnits* and *maxBlockExUnits* protocol parameters are used to limit the total per-transaction and per-block resource use. These only apply to phase-2 scripts. The parameters are used to ensure that the time and memory that are required to verify a block are bounded.

Value size limit as a protocol parameter. The new parameter *maxValSize* replaces the constant maxValSize used (in the ShelleyMA era) to limit the size of the Value part of an output in a serialised transaction.

Collateral inputs. Collateral inputs in a transaction are used to cover the transaction fees in the case that a phase-2 script fails (see Section 3.2). The term *collateral* refers to the total Ada contained in the UTxOs referenced by collateral inputs.

The parameter *collateralPercent* is used to specify the percentage of the total transaction fee its collateral must (at minimum) cover. The collateral inputs must not themselves be locked by a script. That is, they must be VKey inputs. The parameter *maxCollateralInputs* is used to limit, additionally, the total number of collateral inputs, and thus the total number of additional signatures that must be checked during validation.

Abstract types

CostModel	Coefficients for the cost model

Derived types

Language	$=$	$\{\mathsf{PlutusV1}, \cdots\}$	Script Language
Prices	$=$	$\mathsf{Rational} \times \mathsf{Rational}$	Coefficients for ExUnits prices
ExUnits	$=$	$\mathbb{N} \times \mathbb{N}$	Abstract execution units
LangDepView	$=$	$\mathsf{ByteString} \times \mathsf{ByteString}$	Language Tag and PParams view

Deprecated Protocol Parameters

$\mathit{minUTxOValue} \mapsto \mathsf{Coin} \in \mathsf{PParams}$	Min. amount of Ada each UTxO must contain

New Protocol Parameters

$$\begin{aligned}
\mathit{costmdls} &\mapsto (\mathsf{Language} \mapsto \mathsf{CostModel}) \in \mathsf{PParams} \\
\mathit{prices} &\mapsto \mathsf{Prices} \in \mathsf{PParams} \\
\mathit{maxTxExUnits} &\mapsto \mathsf{ExUnits} \in \mathsf{PParams} \\
\mathit{maxBlockExUnits} &\mapsto \mathsf{ExUnits} \in \mathsf{PParams} \\
\mathit{maxValSize} &\mapsto \mathbb{N} \in \mathsf{PParams} \\
\mathit{coinsPerUTxOWord} &\mapsto \mathsf{Coin} \in \mathsf{PParams} \\
\mathit{collateralPercent} &\mapsto \mathbb{N} \in \mathsf{PParams} \\
\mathit{maxCollateralInputs} &\mapsto \mathbb{N} \in \mathsf{PParams}
\end{aligned}$$

Accessor Functions

costmdls, prices, maxTxExUnits, maxBlockExUnits, maxValSize, coinsPerUTxOWord, collateralPercent, maxCollateralInputs

Helper Functions

$$\begin{aligned}
&\mathsf{getLanguageView} : \mathsf{PParams} \to \mathsf{Language} \to \mathsf{LangDepView} \\
&\mathsf{getLanguageView}\ \mathit{pp}\ \mathsf{PlutusV1} = (\mathsf{serialize}\ \mathsf{PlutusV1},\ (\mathsf{serialize}\ (\mathsf{costmdls}\ \mathit{pp}\ \{\mathsf{PlutusV1}\})))
\end{aligned}$$

Figure 1: Definitions Used in Protocol Parameters

3 Transactions

This section outlines the changes that are needed to the transaction structure to enable phase-2 scripts to validate the minting of tokens, spending outputs, verifying certificates, and verifying withdrawals. Figure 2 gives the modified transaction types and helper functions. We make the following changes and additions:

- ScriptIntegrityHash is the type of a hash of script execution related data. It is the hash of txrdmrs and relevant protocol parameters.
- $\mathsf{Script}^{\mathsf{ph1}}$ is the type of phase-1 scripts.
- $\mathsf{Script}^{\mathsf{ph2}}$ is the type of phase-2 scripts.
- Data is a type for communicating data to script. It is kept abstract here.
- Script is the type of all scripts, both phase-1 and phase-2.
- IsValid is a tag that indicates that a transaction expects that all its phase-2 scripts will be validated. This tag is added by the block creator when constructing a block, and its correctness is verified as part of the script execution.
- Datum is a type alias for the Data type, used to signify that terms of this type are intended to be used strictly as datum objects.
- Redeemer is also a type alias for the Data type, used to signify that terms of this type are intended to be used strictly as redeemers.
- TxOut is the type of transaction outputs. These are extended to include an optional hash of a datum. Note that *any* output can optionally include such a hash, even though only phase-2 scripts can actually use the hash value. Since it is in general impossible for the ledger implementation to know at the time of transaction creation whether or not an output belongs to a phase-2 script, we simply allow any output to contain a hash of a datum.
- Tag lets us differentiate what a script can validate, i.e.
  Spend to validate spending a script UTxO entry;
  Mint to validate minting tokens;
  Cert to validate certificates with script credentials; and
  Rewrd to validate reward withdrawals from script addresses.
- RdmrPtr is a pair of a tag and an index. This type is used to index the Plutus redeemers that are included in a transaction, as discussed below.

We add the following helper functions:

- language selects the language of a phase-2 script, and $\diamond$ in case the script is phase-1.
- hashData hashes a value of type Data.
- hashScriptIntegrity hashes the protocol parameters and data relevant to script execution. In particular, the redeemer structure, and the datum objects.

Auxiliary Data. The auxiliary data in an Alonzo transaction has the same structure as in a ShelleyMA transaction, but can additionally contain phase-2 scripts.

Abstract types

$$\begin{array}{rr}
\mathsf{ScriptIntegrityHash} & \text{Script data hash} \\
\mathsf{Script}_{plc} & \text{Plutus core scripts} \\
\mathsf{Data} & \text{Generic script data}
\end{array}$$

Script types

$$\begin{array}{lcl}
\mathsf{Script}^{\mathsf{ph2}} & = & \mathsf{Script}_{plc} \\
\mathsf{Script} & = & \mathsf{Script}^{\mathsf{ph1}} \uplus \mathsf{Script}^{\mathsf{ph2}} \\
\mathsf{IsValid} & = & \mathsf{Bool} \\
\mathsf{Datum} & = & \mathsf{Data} \\
\mathsf{Redeemer} & = & \mathsf{Data}
\end{array}$$

Derived types

$$\begin{array}{lcl}
\mathsf{ValidityInterval} & = & \mathsf{Slot}^{?} \times \mathsf{Slot}^{?} \\
\mathsf{TxOut} & = & \mathsf{Addr} \times \mathsf{Value} \times \mathsf{DataHash}^{?} \\
\mathsf{Tag} & = & \{\mathsf{Spend},\ \mathsf{Mint},\ \mathsf{Cert},\ \mathsf{Rewrd}\} \\
\mathsf{RdmrPtr} & = & \mathsf{Tag} \times \mathsf{Ix}
\end{array}$$

Helper Functions

$$\mathsf{language} : \mathsf{Script} \to \mathsf{Language}^{?}$$
$$\mathsf{hashData} : \mathsf{Data} \to \mathsf{DataHash}$$

$$\mathsf{getCoin} : \mathsf{TxOut} \to \mathsf{Coin}$$
$$\mathsf{getCoin}\ (\_, v, \_) = \mathsf{valueToCoin}\ v$$

$$\mathsf{hashScriptIntegrity} : \mathsf{PParams} \to \mathbb{P}\ \mathsf{Language} \to (\mathsf{RdmrPtr} \mapsto \mathsf{Redeemer} \times \mathsf{ExUnits}) \to (\mathsf{DataHash} \mapsto \mathsf{Datum}) \to \mathsf{ScriptIntegrityHash}^{?}$$

$$\mathsf{hashScriptIntegrity}\ pp\ langs\ rdmrs\ dats = \begin{cases} \diamond & rdmrs = \emptyset \wedge langs = \emptyset \wedge dats = \emptyset \\ \mathsf{hash}(rdmrs, dats, \{\mathsf{getLanguageView}\ pp\ l \mid l \in langs\}) & \text{otherwise} \end{cases}$$

Figure 2: Definitions for Transactions

3.1 Witnessing

Figure 3 defines the witness type, TxWitness. This contains everything in a transaction that is needed for witnessing, namely:

- VKey signatures;
- a map of scripts indexed by their hashes, including phase-2 scripts;
- a map of terms of type Datum indexed by their hashes, containing all required datum objects as well as any optional ones that are posted for communication; and
- a map of a pair of a Redeemer object and an ExUnits value indexed by RdmrPtr, containing the redeemers and execution units budgets.

Note that there is a difference between the way scripts and datum objects are included in a transaction (as a set) versus how redeemers are included (as an indexed structure).

Transaction Types

$$
\begin{array}{rlll}
\mathsf{TxWitness} = & (\mathsf{VKey} \mapsto \mathsf{Sig}) & \mathsf{txwitsVKey} & \text{VKey signatures} \\
& \times\ (\mathsf{ScriptHash} \mapsto \mathsf{Script}) & \mathsf{txscripts} & \text{All scripts} \\
& \times\ (\mathsf{DataHash} \mapsto \mathsf{Datum}) & \mathsf{txdats} & \text{All datum objects} \\
& \times\ (\mathsf{RdmrPtr} \mapsto \mathsf{Redeemer} \times \mathsf{ExUnits}) & \mathsf{txrdmrs} & \text{Redeemers/budget}
\end{array}
$$

$$
\begin{array}{rlll}
\mathsf{TxBody} = & \mathbb{P}\ \mathsf{TxIn} & \mathsf{txins} & \text{Inputs} \\
& \times\ \mathbb{P}\ \mathsf{TxIn} & \mathsf{collateral} & \text{Collateral inputs} \\
& \times\ (\mathsf{Ix} \mapsto \mathsf{TxOut}) & \mathsf{txouts} & \text{Outputs} \\
& \times\ \mathsf{DCert}^* & \mathsf{txcerts} & \text{Certificates} \\
& \times\ \mathsf{Value} & \mathsf{mint} & \text{A minted value} \\
& \times\ \mathsf{Coin} & \mathsf{txfee} & \text{Total fee} \\
& \times\ \mathsf{ValidityInterval} & \mathsf{txvldt} & \text{Validity interval} \\
& \times\ \mathsf{Wdrl} & \mathsf{txwdrls} & \text{Reward withdrawals} \\
& \times\ \mathsf{Update}^? & \mathsf{txUpdates} & \text{Update proposals} \\
& \times\ \mathbb{P}\ \mathsf{KeyHash} & \mathsf{reqSignerHashes} & \text{Hashes of VKeys passed to scripts} \\
& \times\ \mathsf{ScriptIntegrityHash}^? & \mathsf{scriptIntegrityHash} & \text{Hash of script-related data} \\
& \times\ \mathsf{AuxiliaryDataHash}^? & \mathsf{txADhash} & \text{Auxiliary data hash} \\
& \times\ \mathsf{Network}^? & \mathsf{txnetworkid} & \text{Tx network ID}
\end{array}
$$

$$
\begin{array}{rlll}
\mathsf{Tx} = & \mathsf{TxBody} & \mathsf{txbody} & \text{Body} \\
& \times\ \mathsf{TxWitness} & \mathsf{txwits} & \text{Witnesses} \\
& \times\ \mathsf{IsValid} & \mathsf{isValid} & \text{Validation tag} \\
& \times\ \mathsf{AuxiliaryData}^? & \mathsf{auxiliaryData} & \text{Auxiliary data}
\end{array}
$$

Figure 3: Definitions for transactions, cont.

Hash reference (script/datum object): Scripts and datum objects are referred to explicitly via their hashes, which are included in the UTxO or the transaction. Thus, they can be looked up in the transaction without any key in the data structure.

No hash reference (redeemers): For redeemers, we use a reverse pointer approach and index the redeemer by a pointer to the item for which it will be used. For details on how a script finds its redeemer, see Section 4.1.

3.2 Transactions

We have also made the following changes to the body of the transaction:

- The new field collateral is used to specify the *collateral inputs* that are collected (into the fee pot) to cover a percentage of transaction fees (usually 100 percent or more) *in case the transaction contains failing phase-2 scripts*. They are not collected otherwise. The purpose of the collateral is to cover the resource use costs incurred by block producers running scripts that do not validate.

  Collateral inputs behave like regular inputs, except that they must be VKey locked and can only contain Ada. See 4.

  It is permitted to use the same inputs as both collateral and a regular inputs, as exactly one set of inputs is ever collected: collateral ones in the case of script failure, and regular inputs in the case when all scripts validate.

- There is a new field called reqSignerHashes that is used to specify a set of hashes of keys that must sign the transaction in addition to the signatures required to validate spending from VKey addresses, checking certificate validity, and validating reward withdrawals. This field is there to allow users to add signatures that
  - cannot be stripped from a transaction without invalidating it in phase 1 (ie. the validation phase where fees are not collected for the validation work done)
  - are not attached to any validation being done in phase 1, such as VKey output spending

  This is a field where users can add signatures that a phase-2 script requires internally - the transaction and the ledger are otherwise unaware of what signatures are expected by these scripts.
- We include a hash of some script-related data, specifically the redeemers and protocol parameters, with accessor scriptIntegrityHash.
- We include the ID of the network on which the transaction lives, txnetworkid. This is in addition to the network ID's already included in the reward and output addresses.

A complete transaction is comprised of:

1. the transaction body,
2. all the information that is needed to witness transactions,
3. the IsValid tag, which indicates whether all scripts that are executed during the script execution phase validate. The correctness of the tag is verified as part of the ledger rules, and the block is deemed to be invalid if it is applied incorrectly. It can be used to re-apply blocks without script re-validation. Since this tag is not part of the body, i.e. it is not signed, the block producer can (and must) choose the correct value.
4. any auxiliary data.

NOTE

In the implementation the IsValid tag is part of a user-submitted transaction, but it is ignored and re-computed by the block producer.

3.3 Additional Role of Signatures on TxBody

The transaction body and the UTxO must uniquely determine all the data that can influence on-chain transfer of value. This is so that the signatures can ensure that this data is not tampered with. As before, a transaction that does not include the correct signatures is completely invalid. Thus, all data that influences script validation outcomes must also be determined by the body.

Scripts and datum objects whose hashes are specified on-chain do not require to be signed when included in a transaction. That is, script hashes locking UTxOs (as well as with the datum hashes these UTxOs contain), and also the posted certificates, minting policies, and reward addresses do not need to be signed. The optional datum objects, however, could be stripped from the transaction without making it invalid. The optional datums are stored in the same map the required ones, so, for this reason, we do include all datum objects in the script integrity hash calculation. The hash of the indexed redeemer structure and the protocol parameters that are used by script interpreters are included in the body of the transaction, as these are composed by

the transaction author rather than fixed via a hash on the ledger. In the future, other parts of the ledger state may also need to be included in this hash, if they are passed as arguments to a new script interpreter.

3.4 Data required for script validation.

The body of the transaction, alongside the witness data, are required for script validation. Script validation should not depend on the auxiliary data in any way. Script validation may require certain optional datums to be posted. For this reason, we include optional datums in the part of the transaction (the witnesses) which scripts do see. Any metadata that a user wants to expose to a transaction can be included in the redeemer, whose purpose is to contain all user-supplied data relevant to validation.

For more about what scripts do and do not inspect, see Section A.

4 UTxO

Functions

$$\mathsf{isTwoPhaseScriptAddress} : \mathsf{Tx} \to \mathsf{Addr} \to \mathsf{Bool}$$

$$\mathsf{isTwoPhaseScriptAddress}\ tx\ a = \begin{cases} \mathsf{True} & a \in \mathsf{Addr}^{\mathsf{script}} \wedge \mathsf{validatorHash}\ a \mapsto s \in \mathsf{txscripts}(\mathsf{txwits}\ tx) \wedge s \in \mathsf{Script}^{\mathsf{ph2}} \\ \mathsf{False} & \text{otherwise} \end{cases}$$

$$\mathsf{totExUnits} : \mathsf{Tx} \to \mathsf{ExUnits}$$

$$\mathsf{totExUnits}\ tx = \sum_{r \in \mathsf{txrdmrs}\ tx} \mathsf{exunits}\ r$$

$$\mathsf{feesOK} : \mathsf{PParams} \to \mathsf{Tx} \to \mathsf{UTxO} \to \mathsf{Bool}$$

$$\begin{aligned} &\mathsf{feesOK}\ pp\ tx\ utxo\ = \\ &\ \mathsf{minfee}\ pp\ \ tx \leq \mathsf{txfee}\ txb \wedge (\mathsf{txrdmrs}\ tx \neq \emptyset \Rightarrow \\ &\quad ((\forall (a, \_, \_) \in \mathsf{range}\ (\mathsf{collateral}\ txb \lhd utxo), a \in \mathsf{Addr}^{\mathsf{vkey}}) \\ &\quad \wedge\ \mathsf{isAdaOnly}\ balance \\ &\quad \wedge\ balance * 100 \geq \mathsf{txfee}\ txb * (\mathsf{collateralPercent}\ pp) \\ &\quad \wedge\ \mathsf{collateral}\ tx \neq \emptyset) \\ &\quad \textbf{where} \\ &\qquad txb = \mathsf{txbody}\ tx \\ &\qquad balance = \mathsf{ubalance}\ (\mathsf{collateral}\ txb \lhd utxo) \end{aligned}$$

$$\mathsf{cbalance} : \mathsf{UTxO} \to \mathsf{Coin}$$

$$\mathsf{cbalance}\ utxo = \mathsf{valueToCoin}\ (\mathsf{ubalance}\ utxo)$$

$$\mathsf{txscriptfee} : \mathsf{Prices} \to \mathsf{ExUnits} \to \mathsf{Coin}$$

$$\mathsf{txscriptfee}\ (pr_{mem}, pr_{steps})\ (mem, steps) = \mathsf{ceiling}\ (pr_{mem} * mem + pr_{steps} * steps)$$

$$\mathsf{minfee} : \mathsf{PParams} \to \mathsf{Tx} \to \mathsf{Coin}$$

$$\mathsf{minfee}\ pp\ tx = (\mathsf{a}\ pp) \cdot \mathsf{txSize}\ tx + (\mathsf{b}\ pp) + \mathsf{txscriptfee}\ (\mathsf{prices}\ pp)\ (\mathsf{totExunits}\ (\mathsf{txbody}\ tx))$$

Figure 4: Functions related to fees and collateral

We have added or changed several functions that deal with fees and collateral as shown in Figure 4.

- isTwoPhaseScriptAddress is a predicate that checks whether an address is used as a script address with a phase-2 script.
- totExunits calculates the total ExUnits in a transaction by summing the per-script units stored in the indexed redeemer structure
- The predicate feesOK checks whether the transaction is paying the necessary fees, and that it does it correctly. That is, it checks that:

(i) the fee amount that the transaction states it is paying suffices to cover the minimum fee that the transaction is obligated to pay; and if the transaction uses phase-2 scripts, that

(ii) the collateral inputs refer only to UTxOs with key-hash addresses;

(iii) all the collateral inputs refer to UTxOs containing Ada only and no other kinds of token; and

(iv) the collateral provided is sufficient to cover the percentage of the transaction fee amount specified by the protocol parameter collateralPercent

(v) the set of collateral inputs is non-empty

Note that checking that a transaction is carrying redeemers is the most simple way to check that it is carrying phase-2 scripts. A separate check is done in the rule prior to calling the feesOK function that ensure that this is the case.

- txscriptfee calculates the fee that a transaction must pay for script execution based on the amount of ExUnits it has budgeted, and the prices in the current protocol parameters for each component of ExUnits. The prices are expressed as rational numbers, so we take the ceiling of the result of the calculation to get an integer-valued Coin.
- The minimum fee calculation, minfee, includes the script fees that the transaction is obligated to pay in order to run its scripts.

Note that when creating a transaction, the wallet is responsible for determining the fees. Thus, it also has to execute the phase-2 scripts and include the fees for their execution.

4.1 Combining Scripts with Their Inputs

Figure 5 shows the functions that are needed to retrieve all the data that is relevant to Plutus script validation. These include:

- EI is a global constant needed to compute the current system time.
- SysSt is another global constant needed to compute the system time.
- UTCTime is the system time (UTC time zone)
- ScriptPurpose is a sum type of all parts of a transaction that may require a script witness to validate. Note that this contains the data (eg. a certificate $c \in$ txcerts txb, or a transaction input $tin \in$ txcerts txb) of the item being validated, not just a tag indicating the type.
- indexof is a helper function that finds the index of a given certificate, value, input, or withdrawal in a list, finite map, or set of such objects. The implementation is not given directly, but we give a high-level description of how it relies on the underlying ordering. For each of these, it computes the index as follows :
  - certificates in the DCert list are indexed in the order in which they arranged in the (full, unfiltered) list of certificates inside the transaction
  - the index of a reward account *ract* in the reward withdrawals map is the index of *ract* as a key in the (unfiltered) map. The keys of the Wdrl map are arranged in the order defined on the RewardAcnt type, which is a lexicographical (abbrv. lex) order on the pair of the Network and the Credential.

  - the set of inputs is an ordered set (according to the order defined on the type TxIn) - this also is the order in which the elements of the set are indexed (lex order on the pair of TxId and Ix). All inputs of a transaction are included in the set being indexed (not just the ones that point to a Plutus script UTxO)
  - the set of policy IDs is ordered according to the order defined on PolicyID (lex). The index of a PolicyID in this set of policy IDs is computed according to this order. Note that at the use site, the set of policy IDs passed to indexof is the (unfiltered) domain of the Value map in the mint field of the transaction.

  Note that in all the orderings above, long and short lex are one in the same, since for each of the ordered types, all the terms of that type have the same length.

- indexedRdmrs indexes the pair of a redeemer and an ExUnits value for a given script by its script purpose (eg. the input, or certificate, etc.), and $\diamond$ if no associated data is found. It uses the script purpose to generate the RdmrPtr key, then find the corresponding entry in the redeemer structure.

- rdptr builds a redeemer pointer from a script purpose by setting the tag according to the type of the script purpose, and the index according to the placement of script purpose inside its container (ie. by using indexof)

4.2 Plutus Script Validation

Figure 6 shows the abstract functions that are used for script validation. In this figure, the function txInfo, which creates a summary of the transaction and ledger state as a TxInfo, is defined in Section A, as is the TxInfo type itself. The function valContext, also defined in A, constructs the validation context from the TxInfo summary.

- epochInfoSlotToUTCTime translates a slot number to system time if possible. This translation is implemented by the consensus algorithm (not the ledger), and requires two additional parameters to do this. Note that this does not depend on the clock of the node.

  If it is not possible to do this translation, $\diamond$ is returned. The reason it may not be possible to translate is that the slot number is too far in the future for the system to accurately predict the exact time to which it refers.

- runPLCScript validates Plutus scripts. It takes the following arguments:

  - A cost model, that is used to calculate the ExUnits that are needed for script execution;
  - A script to execute;
  - the execution unit budget; and
  - A list of terms of type Data that will be passed to the script.

  It outputs the validation result. Note that script execution stops if the full budget has been spent before validation is complete. It is left abstract because it is implemented as part of the phase-1 script interpreter, not the ledger. The interpreter called depends on the language of the script.

Slot to time translation. One of the inputs to scripts is the transaction validity interval (recall here that they do not actually see the current slot number). The length of a slot may change in a future era. In this case, for a script written in a previous era, if we were to pass the transaction validity interval expressed as slot numbers, we get that

Global Constants

$$\begin{aligned} \mathsf{EI} &\in \mathsf{EpochInfo} \quad \text{Alonzo epoch info} \\ \mathsf{SysSt} &\in \mathsf{SystemStart} \text{System start time} \end{aligned}$$

Abstract Types

$$tm \in \mathsf{UTCTime} \text{System time}$$

Derived types

$$sp \in \mathsf{ScriptPurpose} \quad = \quad \mathsf{PolicyID} \uplus \mathsf{TxIn} \uplus \mathsf{Addr}_{\mathsf{rwd}} \uplus \mathsf{DCert}$$

Abstract functions

$$\begin{aligned} &\mathsf{indexOf} : \mathsf{DCert} \to \mathsf{DCert}^* \to \mathsf{Ix}^? \\ &\mathsf{indexOf} : \mathsf{Addr}_{\mathsf{rwd}} \to \mathsf{Wdrl} \to \mathsf{Ix}^? \\ &\mathsf{indexOf} : \mathsf{TxIn} \to \mathbb{P}\ \mathsf{TxIn} \to \mathsf{Ix}^? \\ &\mathsf{indexOf} : \mathsf{PolicyID} \to \mathbb{P}\ \mathsf{PolicyID} \to \mathsf{Ix}^? \end{aligned}$$

Indexing functions

$$\mathsf{indexedRdmrs} : \mathsf{Tx} \to \mathsf{ScriptPurpose} \to (\mathsf{Redeemer} \times \mathsf{ExUnits})^?$$

$$\mathsf{indexedRdmrs}\ tx\ sp = \begin{cases} r & (\mathsf{rdptr}\ txb\ sp) \mapsto r \in \mathsf{txrdmrs}\ (\mathsf{txwits}\ tx)\} \\ \diamond & \text{otherwise} \end{cases}$$

where

$$txb = \mathsf{txbody}\ tx$$

$$\mathsf{rdptr} : \mathsf{TxBody} \to \mathsf{ScriptPurpose} \to \mathsf{RdmrPtr}^?$$

$$\mathsf{rdptr}\ txb\ sp = \begin{cases} (\mathsf{Cert}, \mathsf{indexOf}\ sp\ (\mathsf{txcerts}\ txb)) & sp \in \mathsf{DCert} \\ (\mathsf{Rewrd}, \mathsf{indexOf}\ sp\ (\mathsf{txwdrls}\ txb)) & sp \in \mathsf{Addr}_{\mathsf{rwd}} \\ (\mathsf{Mint}, \mathsf{indexOf}\ sp\ (\mathrm{dom}(\mathsf{mint}\ txb))) & sp \in \mathsf{PolicyID} \\ (\mathsf{Spend}, \mathsf{indexOf}\ sp\ (\mathsf{txinputs}\ txb)) & sp \in \mathsf{TxIn} \end{cases}$$

Figure 5: Indexing script and data objects

- the script logic is expressed in terms of slot numbers which likely assume the slot length of the era in which it was created
- the slot numbers in the validity interval of the transaction are used assuming slot length of the current era

Therefore, the slot numbers inside the contract and the slot numbers in the transaction map differently onto points in time. The ledger does not have access to data (or conversion functions) that is needed to convert transaction validity interval slots numbers to slots numbers that correspond to the contracts world view. To address this, we decided to pass phase-2 contracts (in all languages and all eras) the system time instead of slot numbers. The conversion function is implemented by consensus (see de Vries et al. (2021)), which has the information to do this correctly for

- all slots prior to the current slot
- a number of slots after the current slot (this number is determined by the consensus's forecast window)

Note that because of this potential slot length change issue, Plutus scripts will not receive any information in terms of slot numbers (no phase-1 scripts, no validity intervals as slots, and not the current slot). Phase-1 timelock scripts continue to operate on validity interval data expressed as slot numbers, and are therefore vulnerable to the consequences of changing the slot length.

Validation context construction. As additional phase-2 scripting languages become supported in the future, scripts of different languages may expect different (or differently structured) transaction and ledger data summary. The txInfo function will be implemented differently for each new language. So, the construction is dependent on both the language of the script being validated and the ledger/transaction structure of the current era.

In order to ensure that running scripts of all script languages is supported indefinitely across future eras, the txInfo function must be total. Recall here that while *running* all scripts must be supported across all future ledger changes, it is not a requirement that every script must *validate* within the context of some transaction.

The txInfo output is computed once for the whole transaction. The output of the function valContext is computed separately for each script purpose. The script purpose is passed to it to allow the script to reference itself via its hash, and to be aware of what it is validating.

Know your contract arguments. A Plutus validator script may receive either a list of three terms of type Data, in case it validates the spending of script outputs or two terms (redeemer and context, with no datum), for all other uses. Script authors must keep this in mind when writing scripts, since the ledger call to the interpreter is oblivious to what arguments are required.

The following is a summary of what arguments are required for what scripts:

- Datum objects are required for all output-locking phase-2 scripts. That is, a phase-2 script-locked output that does not include a datum hash is unspendable.
- Non-output locking scripts do not expect datum objects - that is, it is not possible to pass a datum to a script used for another purpose (certificate, etc.).
- Redeemers are required for all phase-2 scripts.

If a transaction carrying a Plutus script is missing any inputs required to run a script, we have specified this to be a phase 1 ledger failure, and the script is never run (even if its code does not look at the missing input). The reason for this approach is that not passing all inputs results in a script never actually getting run, rather than running and failing.

Abstract Script Validation Functions

$$\mathsf{epochInfoSlotToUTCTime} : \mathsf{EpochInfo} \to \mathsf{SystemStart} \to \mathsf{Slot} \to \mathsf{UTCTime}^?$$

Translate slot number to system time or fail

$$\mathsf{runPLCScript} : \mathsf{CostModel} \to \mathsf{Script}_{plc} \to \mathsf{ExUnits} \to \mathsf{Data}^* \to \mathsf{IsValid}$$

Validate a Plutus script, taking resource limits into account

Notation

$$[\![script_v]\!]_{cm,eu}\ d = \qquad \mathsf{runPLCScript}\ cm\ script_v\ eu\ d$$

Figure 6: Script Validation, cont.

Figure 7 contains the functions used to match scripts with their corresponding inputs and pass them to the evaluator.

- getDatum looks for a datum associated with a given script purpose. Note that only an TxIn-type script purpose can result in finding an associated datum hash. If no datum is found, an empty list is returned. A list containing the found datum is returned otherwise.
- collectTwoPhaseScriptInputs builds a list of scripts, paired with their inputs. Specifically, each tuple in the list contains :
  - the script, only if it is a phase-2 one;
  - a list of the following script arguments, in this order:
    - the hash of the required datum, if any (returned by the getDatum function, wrapped in a list type instead of a Data? type).
    - a redeemer, returned as the first term of the pair by the indexedRdmrs function. and an ExUnits amount. We are assuming this value is not $\diamond$ here because that must have already been checked by the UTXOW rule.
    - the validation context, built by the valContext function using the transaction summary built by txInfo, together with the current item being validated;
  - an ExUnits amount, which is the second term of the pair returned by the indexedRdmrs function.
  - the cost model for PlutusV1, specified in the protocol parameters
- evalScripts evaluates a whole list of scripts. For PlutusV1 scripts, it evaluates them paired with all their inputs by calling the phase-2 script validator function, runPLCScript (see A for details). For phase-1 (timelock) scripts, it calls evalTimelock. This case is unused though, as the list of scripts and inputs that gets passed to the this function is constructed to contain only Plutus scripts. So, phase-1 scripts do not get evaluated twice.

Note that no "checks" are performed within these functions (though they may be, in the implementation). Missing validators, missing inputs, incorrect hashes, the wrong type of script etc, are caught during the application of the UTXOW rule (before these functions are ever applied).

4.3 Two-Phase Transaction Validation for Phase-2 Scripts

Transactions are validated in two phases: the first phase consists of every aspect of transaction validation apart from executing the phase-2 scripts; and the second phase involves actually executing those scripts. This ensures that users pay for the computational resources that are needed to validate phase-2 scripts, even if script validation fails. In order to handle script execution, an additional transition system is used, called UTXOS. It performs the appropriate UTxO state changes, based on the value of the IsValid tag, which it checks using the evalScripts function.

In general, there is no way to check *a-priori* that the budget that has been supplied is sufficient for the transaction. This can only be done by actually running the scripts. From the perspective of the ledger, there is no difference between a script that exhausts the ExUnits budget during validation, and one that fails to validate.

Transaction integrity and charging for failing scripts. If a transaction contains a failing script, the only change to the ledger that is made is that the collateral is collected into the fee pot. It is important to note here that it can be the case that the only signatures on a transaction are those of the keys for the collateral UTxOs (but this must include at least one signature).

The implication of this is that the collateral key owners may be the only users that attest to the integrity of the data in the body of the transaction. These same key owners, however, are also the only users who stand to lose money if the transaction is modified in some way that results

$$\mathsf{getDatum} : \mathsf{Tx} \to \mathsf{UTxO} \to \mathsf{ScriptPurpose} \to \mathsf{Datum}^*$$

$$\mathsf{getDatum}\ tx\ utxo\ sp = \begin{cases} [d] & sp \mapsto (\_, \_, h_d) \in utxo, h_d \mapsto d \in \mathsf{txdats}\ (\mathsf{txwits}\ tx) \\ \epsilon & \text{otherwise} \end{cases}$$

$$\mathsf{collectTwoPhaseScriptInputs} : \mathsf{EpochInfo} \to \mathsf{SystemStart} \to \mathsf{PParams} \to \mathsf{Tx} \to \mathsf{UTxO} \to (\mathsf{Script} \times \mathsf{Data}^* \times \mathsf{ExUnits} \times \mathsf{CostModel})^*$$

$$\begin{aligned}
&\mathsf{collectTwoPhaseScriptInputs}\ ei\ sysSt\ pp\ tx\ utxo = \\
&\ \mathsf{toList}\{(script, (\mathsf{getDatum}\ tx\ utxo\ sp; rdmr; \mathsf{valContext}\ txinfo\ sp), eu, cm) \mid \\
&\quad (sp, scriptHash) \in \mathsf{scriptsNeeded}\ utxo\ (\mathsf{txbody}\ tx), \\
&\quad scriptHash \mapsto script \in \mathsf{txscripts}\ (\mathsf{txwits}\ tx), \\
&\quad (rdmr, eu) := \mathsf{indexedRdmrs}\ tx\ sp, \\
&\quad \mathsf{language}\ script \mapsto cm \in \mathsf{costmdls}\ pp, \\
&\quad script \in \mathsf{Script}^{\mathsf{ph2}}\} \\
&\ \textbf{where} \\
&\quad txinfo = \mathsf{txInfo}\ ei\ sysSt\ (\mathsf{language}\ script)\ pp\ utxo\ tx
\end{aligned}$$

$$\mathsf{evalScripts} : \mathsf{Tx} \times (\mathsf{Script} \times \mathsf{Data}^* \times \mathsf{ExUnits} \times \mathsf{CostModel})^* \to \mathsf{IsValid}$$

$$\mathsf{evalScripts}\ tx\ \epsilon = \mathsf{True}$$

$$\mathsf{evalScripts}\ tx\ ((sc, d, eu, cm); \Gamma) = \begin{cases} [\![sc]\!]_{cm,eu} d \wedge \mathsf{evalScripts}\ tx\ \Gamma & sc \in \mathsf{Script}_{plc} \\ \mathsf{evalTimelock}\ tx\ sc \wedge \mathsf{evalScripts}\ tx\ \Gamma & sc \in \mathsf{Script}^{\mathsf{ph1}} \end{cases}$$

Figure 7: Scripts and their arguments

in phase-2 failure. Transactions with the same body will necessarily have the same outcome of phase-2 script validation (we give the details in the deterministic script validation property, C.8). Therefore, signing the body ensures any modification to any part of a transaction (including witnesses, etc.) or update to the ledger state that affect the transaction's validity will result in a phase-1 failure, and no collateral collected. The implication of this is that the collateral-locking keys have full control over the outcome of phase-2 validation.

The IsValid **Tag.** It is always in the interest of the slot leader to have the new block validate, and for it to contain only valid transactions. This motivates the slot leader to:

1. Correctly apply the IsValid tag;
2. Include all transactions that validate within the block, *even when there is a 2nd phase script validation failure*;
3. Exclude any transactions that are phase-1 invalid

One important reason for adding the validation tag to a transaction is that re-applying blocks will not require repeat execution of scripts in the transactions inside a block, which would increase execution costs. In fact, when replaying blocks, all the witnessing information can be thrown away.

4.4 The UTXOS transition system

We have defined a separate transition system, UTXOS, to represent the two distinct UTxO state changes: i) when all the scripts in a transaction validate; and ii) when at least one fails to validate. Its transition types are identical to the UTXO transition (Figure 8).

State transitions

$$\_ \vdash \_ \xrightarrow[\text{UTXOS}]{} \_ \subseteq \mathbb{P}\ (\mathsf{UTxOEnv} \times \mathsf{UTxOState} \times \mathsf{Tx} \times \mathsf{UTxOState})$$

Figure 8: UTxO script state update types

There are two rules, corresponding to the two possible state changes of the UTxO state in the UTXOS transition system (Figure 9). In both cases, the evalScripts function is called upon to verify that the IsValid tag has been applied correctly. The function collectTwoPhaseScriptInputs is used to build the input list *sLst* for the evalScripts function. The first rule applies when the validation tag is True. In this case, the states of the UTxO, fee and deposit pots, and updates are updated exactly as in the current Shelley ledger spec. The second rule applies when the validation tag is False. In this case, the UTxO state changes as follows:

1. All the UTxO entries corresponding to the collateral inputs are removed;
2. The sum total of the value of the collateral UTxO entries is added to the fee pot.

$$txb := \mathsf{txbody}\ tx \qquad sLst := \mathsf{collectTwoPhaseScriptInputs}\ \mathsf{EI}\ \mathsf{SysSt}\ pp\ tx\ utxo$$

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

$$refunded := \mathsf{keyRefunds}\ pp\ txb$$

$$depositChange := \mathsf{totalDeposits}\ pp\ poolParams\ (\mathsf{txcerts}\ txb)\ -\ refunded$$

$$\text{Scripts-Yes}\ \frac{\mathsf{isValid}\ tx = \mathsf{evalScripts}\ tx\ sLst = \mathsf{True}}{\begin{matrix} slot \\ pp \\ poolParams \\ genDelegs \end{matrix} \vdash \begin{pmatrix} utxo \\ deposits \\ fees \\ pup \end{pmatrix} \xrightarrow[\text{UTXOS}]{tx} \begin{pmatrix} (\mathbf{txins}\ txb \lhd utxo) \cup \mathbf{outs}\ txb \\ deposits + depositChange \\ fees + \mathbf{txfee}\ txb \\ pup' \end{pmatrix}} \quad (1)$$

$$txb := \mathsf{txbody}\ tx \qquad sLst := \mathsf{collectTwoPhaseScriptInputs}\ \mathsf{EI}\ \mathsf{SysSt}\ pp\ tx\ utxo$$

$$\text{Scripts-No}\ \frac{\mathsf{isValid}\ tx = \mathsf{evalScripts}\ tx\ sLst = \mathsf{False}}{\begin{matrix} slot \\ pp \\ poolParams \\ genDelegs \end{matrix} \vdash \begin{pmatrix} utxo \\ deposits \\ fees \\ pup \end{pmatrix} \xrightarrow[\text{UTXOS}]{tx} \begin{pmatrix} \mathbf{collateral}\ txb \lhd utxo \\ deposits \\ fees + \mathbf{cbalance}\ (\mathbf{collateral}\ txb \lhd utxo) \\ pup \end{pmatrix}} \quad (2)$$

Figure 9: State update rules

Figure 10 shows the UTxO − inductive transition rule for the UTXO transition type. This rule has the following changes:

1. The transaction pays fees and supplies collateral Ada correctly, as defined above;
2. The end of the transaction validity interval is translatable into system time (ie. within the consensus's forecast window). This is checked by epochInfoSlotToUTCTime, which returns $\diamond$ if the end slot is outside. Note that we do not need to check that the start slot can be converted to time, because all pasts slots can be converted into time correctly.
3. coinsPerUTxOWord is now a protocol parameter explicitly, the utxoEntrySize calculation is defined differently than for ShelleyMA (see Section B)
4. maxValSize is now also a protocol parameter (not a constant). It represents a size (in bytes) of the total transaction size that the size of a Value in an output can be. Otherwise, this check is the same as in ShelleyMA.
5. The network ID field in the transaction body must match the network ID constant
6. The execution unit budget for a transaction is within the maximum permitted number of units for a transaction;
7. The number of maximum allowed collateral inputs is not exceeded
8. The UTXOS state transition is valid (this is the transition that runs the phase-2 scripts)

The resulting state transition is defined entirely by the application of the UTXOS rule.

4.5 Witnessing

Because of two-phase transaction validation, phase-2 script validation is not part of phase one of transaction witnessing (it is done in phase 2, once the rest of the transaction is validated). However, phase-1 script validation does remain part of transaction witnessing. When witnessing a transaction in phase one, we therefore need to validate only the phase-1 scripts.

We construct the following helper functions :

- witsVKeyNeeded is a Shelley function adjusted to include keys that are specified in the reqSignerHashes field of the transaction, as well as keys locking the UTxOs to which the collateral inputs of the transaction point.
- scriptsNeeded assembles the all the ScriptPurpose terms for validation of every transaction action that requires script validation, paired with the hashes of corresponding the witnessing scripts. This function collects hashes of both phase-1 and phase-2 scripts.
- languages returns the set of (phase-2) script languages of all the scripts included in the transaction

We have made the following changes and additions to the UTXOW preconditions:

- All the phase-1 scripts in the transaction validate;
- The transaction contains exactly those scripts that are required for witnessing and no additional ones;
- The datums included in the witnesses contain all the datums required for validating in phase 2. That is, datums for all output-locking phase-2 scripts for the payment credentials of the addresses of the UTxOs the transaction is spending must have attached datum objects. This check will also fail if the datum hash fields of such UTxOs are $\diamond$, as $\diamond$ is not a hash of a datum (and therefore is not a key of the DataHash-indexed map).

$$txb := \text{txbody } tx \qquad \text{ininterval } slot\ (\text{txvldt } txb) \qquad (\_, i_f) := \text{txvldt } tx$$

$$\diamond \notin \{\text{txrdmrs } tx, i_f\} \Rightarrow \text{epochInfoSlotToUTCTime EI SysSt } i_f \neq \diamond$$

$$\text{txins } txb \neq \emptyset \qquad \text{feesOK } pp\ tx\ utxo \qquad \text{txins } txb \cup \text{collateral } txb \subseteq \text{dom } utxo$$

$$\text{consumed } pp\ utxo\ txb = \text{produced } pp\ poolParams\ txb$$

$$\text{adaID} \notin \text{supp mint } tx$$

$$\forall txout \in \text{txouts } txb,$$
$$\text{getValue } txout \geq \text{inject } (\text{utxoEntrySize } txout * \text{coinsPerUTxOWord } pp)$$

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

$$\forall (\_ \mapsto (a, \_)) \in \text{txouts } txb, a \in \text{Addr}_{\text{bootstrap}} \Rightarrow \text{bootstrapAttrsSize } a \leq 64$$
$$\forall (\_ \mapsto (a, \_)) \in \text{txouts } txb, \text{netId } a = \text{NetworkId}$$
$$\forall (a \mapsto \_) \in \text{txwdrls } txb, \text{netId } a = \text{NetworkId}$$
$$(\text{txnetworkid } txb = \text{NetworkId}) \vee (\text{txnetworkid } txb = \diamond)$$

$$\text{txsize } tx \leq \text{maxTxSize } pp$$

$$\text{totExunits } tx \leq \text{maxTxExUnits } pp \qquad \|\text{collateral } tx\| \leq \text{maxCollateralInputs } pp$$

$$\text{UTxO-inductive}\ \frac{\begin{pmatrix} slot \\ pp \\ poolParams \\ genDelegs \end{pmatrix} \vdash \begin{pmatrix} utxo \\ deposits \\ fees \\ pup \end{pmatrix} \xrightarrow[\text{UTXOS}]{tx} \begin{pmatrix} utxo' \\ deposits' \\ fees' \\ pup' \end{pmatrix}}{\begin{pmatrix} slot \\ pp \\ poolParams \\ genDelegs \end{pmatrix} \vdash \begin{pmatrix} utxo \\ deposits \\ fees \\ pup \end{pmatrix} \xrightarrow[\text{UTXO}]{tx} \begin{pmatrix} utxo' \\ deposits' \\ fees' \\ pup' \end{pmatrix}} \tag{3}$$

Figure 10: UTxO inference rules

- The only datums included in a transaction have hashes that are either in a UTxO corresponding to a transaction input, or in a transaction output. The output datums are for communication only, and are therefore optional, hence the use of subset equality. No additional datums are permitted.
- For every item that needs to be validated by a phase-2 script, and that phase-2 script is present, the transaction contains an entry in the indexed redeemer structure (ie. the execution units and redeemer for it are specified). Note here that if the full script is not present, we cannot tell from the hash that the script is phase-2, and therefore requires a redeemer;
- Each redeemer pointer in the indexed redeemer structure corresponds to a pointer constructed using rdptr from some script purpose of a phase-2 script of the transaction. This ensures that there are no additional entries present in the structure (ie. those that are not required for validation of any script);

$$\begin{aligned}
&\mathsf{witsVKeyNeeded} : \mathsf{UTxO} \to \mathsf{Tx} \to (\mathsf{KeyHash}_{\mathsf{G}} \mapsto \mathsf{VKey}) \to \mathbb{P}\ \mathsf{KeyHash}\\
&\mathsf{witsVKeyNeeded}\ utxo\ tx\ genDelegs =\\
&\quad \{\mathsf{paymentHK}\ a \mid i \mapsto (a, _) \in utxo,\ i \in \mathsf{txinsVKey}\ tx \cup \mathsf{collateral}\ tx\}\\
&\quad \cup\ \{\mathsf{stakeCred}_{\mathsf{r}}\ a \mid a \mapsto _ \in \mathsf{Addr}^{\mathsf{vkey}}_{\mathsf{rwd}} \lhd \mathsf{txwdrls}\ tx\}\\
&\quad \cup\ (\mathsf{Addr}^{\mathsf{vkey}} \cap \mathsf{certWitsNeeded}\ tx)\\
&\quad \cup\ \mathsf{propWits}\ (\mathsf{txup}\ tx)\ genDelegs\\
&\quad \cup \bigcup_{\substack{c \in \mathsf{txcerts}\ tx\\ c \in \mathsf{DCert}_{\mathsf{regpool}}}} \mathsf{poolOwners}\ c\\
&\quad \cup\ \mathsf{reqSignerHashes}\ tx
\end{aligned}$$

$$\begin{aligned}
&\mathsf{scriptsNeeded} : \mathsf{UTxO} \to \mathsf{TxBody} \to \mathbb{P}\ (\mathsf{ScriptPurpose} \times \mathsf{ScriptHash})\\
&\mathsf{scriptsNeeded}\ utxo\ txb =\\
&\quad \{(i, \mathsf{validatorHash}\ a) \mid i \in \mathsf{txinsScript}\ (\mathsf{txins}\ txb)\ utxo, i \mapsto (a, _, _) \in utxo\}\\
&\quad \cup\ \{(a, \mathsf{stakeCred}_{\mathsf{r}}\ a) \mid a \in \mathrm{dom}(\mathsf{Addr}^{\mathsf{script}}_{\mathsf{rwd}} \lhd \mathsf{txwdrls}\ txb)\}\\
&\quad \cup\ \{(cert, c) \mid cert \in (\mathsf{DCert}_{\mathsf{delegate}} \cup \mathsf{DCert}_{\mathsf{deregkey}}) \cap \mathsf{txcerts}\ txb,\\
&\qquad c \in \mathsf{cwitness}\ cert \cap \mathsf{Addr}^{\mathsf{script}}\}\\
&\quad \cup\ \{(pid, pid) \mid pid \in \mathrm{supp}\ (\mathsf{mint}\ txb)\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{languages} : \mathsf{TxWitness} \to \mathbb{P}\ \mathsf{Language}\\
&\mathsf{languages}\ txw = \{\mathsf{language}\ s \mid s \in \mathrm{range}(\mathsf{txscripts}\ txw) \cap \mathsf{Script}^{\mathsf{ph2}}\}
\end{aligned}$$

Figure 11: UTXOW helper functions

- The signatures of the keys whose hashes are specified in the reqSignerHashes field in a transaction have all indeed signed it;
- The hash of the subset of protocol parameter values and witnessing data that have been included in the transaction body is the same as the hash of the witness data and the subset of protocol parameters that are currently contained in the ledger;

If these conditions are all satisfied, then the resulting UTxO state change is fully determined by the UTXO transition (the application of which is also part of the conditions).

State transitions

$$\_ \vdash \_ \xrightarrow[\text{UTXOW}]{} \_ \subseteq \mathbb{P}\ (\mathsf{UTxOEnv} \times \mathsf{UTxOState} \times \mathsf{Tx} \times \mathsf{UTxOState})$$

Figure 12: UTxO with witnesses state update types

$$txb := \text{txbody } tx \qquad txw := \text{txwits } tx$$
$$(utxo, \_, \_, \_) := utxoSt$$
$$witsKeyHashes := \{\text{hashKey } vk | vk \in \text{dom}(\text{txwitsVKey } txw)\}$$
$$inputHashes := \{h \mid (\_ \mapsto (a, \_, h)) \in \text{txins } txb \lhd utxo, \text{isTwoPhaseScriptAddress } tx\ a\}$$
$$\forall s \in \text{range}(\text{txscripts } txw) \cap \text{Script}^{\text{ph1}}, \text{validateScript } s\ tx$$
$$\{h \mid (\_, h) \in \text{scriptsNeeded } utxo\ txb\} = \text{dom}(\text{txscripts } txw)$$
$$inputHashes \subseteq \text{dom}(\text{txdats } txw)$$
$$\text{dom}(\text{txdats } txw) \subseteq inputHashes \cup \{h \mid (\_, \_, h) \in \text{txouts } tx\}$$
$$\text{dom}(\text{txrdmrs } tx) = \{\text{rdptr } txb\ sp \mid (sp, h) \in \text{scriptsNeeded } utxo\ (\text{txbody } tx),$$
$$h \mapsto s \in \text{txscripts } txw, s \in \text{Script}^{\text{ph2}}\}$$
$$txbodyHash := \text{hash } (\text{txbody } tx)$$
$$\forall vk \mapsto \sigma \in \text{txwitsVKey } tx, \mathcal{V}_{vk}[\![txbodyHash]\!]_\sigma$$
$$\text{witsVKeyNeeded } utxo\ tx\ genDelegs \subseteq witsKeyHashes$$
$$genSig := \{\text{hashKey } gkey | gkey \in \text{dom } genDelegs\} \cap witsKeyHashes$$
$$\{c \in \text{txcerts } txb \cap \text{DCert}_{\text{mir}}\} \neq \emptyset \implies |genSig| \geq \text{Quorum}$$
$$adh := \text{txADhash } txb \qquad ad := \text{auxiliaryData } tx$$
$$(adh = \diamond \wedge ad = \diamond) \vee (adh = \text{hashAD } ad)$$
$$\text{languages } txw \subseteq \text{dom}(\text{costmdls } tx)$$
$$\text{scriptIntegrityHash } txb = \text{hashScriptIntegrity } pp\ (\text{languages } txw)\ (\text{txrdmrs } txw)\ (\text{txdats } txw)$$

$$\text{UTxO-witG}\ \frac{\begin{array}{r} slot \\ pp \\ poolParams \\ genDelegs \end{array} \vdash utxoSt \xrightarrow[\text{UTXO}]{tx} utxoSt'}{\begin{array}{r} slot \\ pp \\ poolParams \\ genDelegs \end{array} \vdash utxoSt \xrightarrow[\text{UTXOW}]{tx} \mathbf{utxoSt'}} \quad (4)$$

Figure 13: UTxO with witnesses inference rules for Tx

5 Ledger State Transition

Figure 14 separates the case where all the scripts in a transaction successfully validate from the case where there is one or more that does not. These two cases are distinguished by the use of the IsValid tag. Besides collateral collection, no side effects should occur when processing a transaction that contains a script that does not validate. That is, no delegation or pool state updates, update proposals, or any other observable state change, should be applied. The UTxO rule is still applied, however, as this is where the correctness of the validation tag is verified, and where collateral inputs are collected.

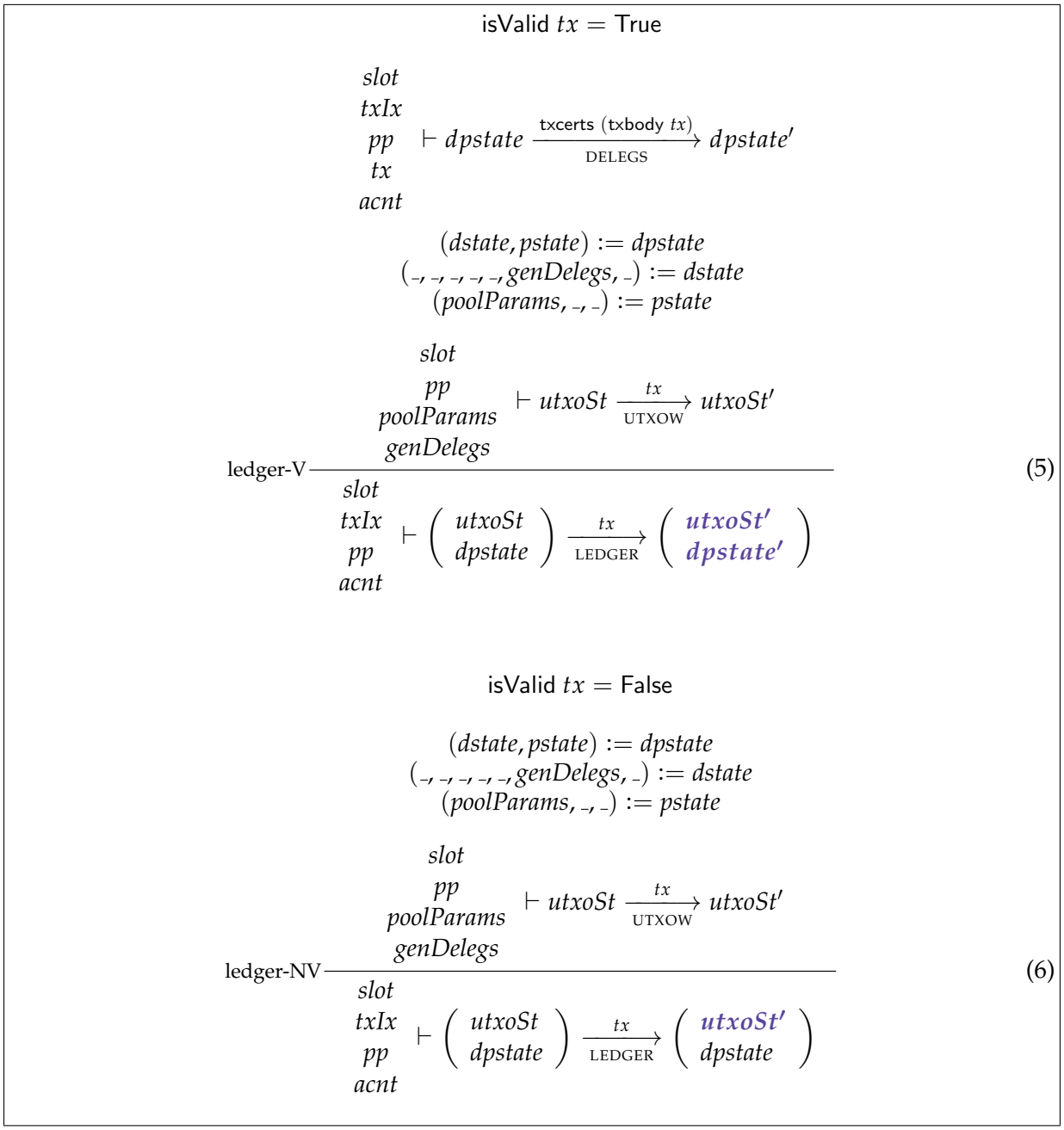

Figure 14: Ledger inference rules

6 Blockchain layer

6.1 Block Body Transition

Figure 15 includes an additional check that the sum of the execution units of all transactions in a block does not exceed the maximum total units per block (specified in a protocol parameter).

$$txs := \mathsf{bbody}\ block \qquad bhb := \mathsf{bhbody}\ (\mathsf{bheader}\ block) \qquad hk := \mathsf{hashKey}\ (\mathsf{bvkcold}\ bhb)$$

$$\mathsf{bBodySize}\ txs = \mathsf{hBbsize}\ bhb \qquad \mathsf{bbodyhash}\ txs = \mathsf{bhash}\ bhb$$

$$slot := \mathsf{bslot}\ bhb \qquad fSlot := \mathsf{firstSlot}\ (\mathsf{epoch}\ slot)$$

$$\sum_{tx \in txs} \mathsf{totExunits}\ tx \leq \mathsf{maxBlockExUnits}\ pp$$

$$\text{Block-Body}\ \frac{\begin{matrix}\mathsf{bslot}\ bhb \\ pp \\ acnt\end{matrix} \vdash ls \xrightarrow[\text{LEDGERS}]{txs} ls'}{\begin{matrix}pp \\ acnt\end{matrix} \vdash \begin{pmatrix} ls \\ b \end{pmatrix} \xrightarrow[\text{BBODY}]{block} \begin{pmatrix} ls' \\ \mathbf{incrBlocks}\ (\mathbf{isOverlaySlot}\ fSlot\ (\mathbf{d}\ pp)\ slot)\ hk\ b \end{pmatrix}} \tag{7}$$

Figure 15: BBody rules

We have also defined a function that transforms the Shelley ledger state into the corresponding Alonzo one, see Figure 16. We refer to the Shelley-era protocol parameter type as ShelleyPParams, and the corresponding Alonzo type as PParams. 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.

Types and Constants

$$\mathsf{NewParams} = (\mathsf{Language} \mapsto \mathsf{CostModel}) \times \mathsf{Prices} \times \mathsf{ExUnits} \times \mathsf{ExUnits} \times \mathbb{N} \times \mathsf{Coin} \times \mathbb{N} \times \mathbb{N}$$

the type of new parameters to add for Alonzo

$$ivPP \in \mathsf{NewParams}$$

the initial values for new Alonzo parameters

Shelley to Alonzo Transition Functions

$$\mathsf{toAlonzo} : \mathsf{ShelleyChainState} \to \mathsf{ChainState}$$

$$\mathsf{toAlonzo}\ chainstate = chainstate'$$

where

$$chainstate'_{pparams} = (pp \cup ivPP) - minUTxOValue$$

transform Shelley chain state to Alonzo state

Figure 16: Shelley to Alonzo State Transtition

7 MIR Certificates

In the Alonzo era, MIR certs have:

- Slightly different behavior - prior to the Alonzo era, MIR certificates within the same epoch override values for repeated stake addresses. In the Alonzo era, repetitions are handled by adding the values. Moreover, negative values are allowed, so long as the change never results in an overall negative sum. See Figure 18.
- Extended functionality - MIR certificates can now be used to transfer funds between the reserves and the treasury. See Figure 19.

Figure 17 describes the necessary changes to the types. Note that mirTarget was named credCoinMap in the Shelley specification. Figure 18 is a modification to the DELEG rule, and Figure 19 is an addition to the DELEG rule. Lastly, Figure 20 is the updated MIR rule.

Derived types

$$\begin{array}{rll} \mathsf{InstantaneousRewards} = & (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) & \text{rewards from reserves} \\ & \times (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) & \text{rewards from treasury} \\ & \times \mathsf{Coin} & \text{pending modification to reserves} \\ & \times \mathsf{Coin} & \text{pending modification to treasury} \end{array}$$

New MIR Cert accessor

$$\mathsf{mirTarget} \;:\; \mathsf{DCert}_{\mathsf{mir}} \to (\mathsf{Credential}_{stake} \mapsto \mathsf{Coin}) \uplus \mathsf{MIRPot}$$

Figure 17: Alonzo Types for MIR updates

$$\frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{mir}} \qquad \mathsf{mirTarget}\ c \in \mathsf{Credential}_{stake} \mapsto \mathsf{Coin} \\ slot < \mathsf{firstSlot}\ ((\mathsf{epoch}\ slot) + 1) - \mathsf{StabilityWindow} \\ (irR,\ irT,\ dR,\ dT) := i_{rwd} \qquad (treasury,\ reserves) := acnt \\ (pot,\ deltaPot,\ irPot) := \begin{cases} (reserves,\ dR,\ irR) & \mathsf{mirPot}\ c = \mathsf{ReservesMIR} \\ (treasury,\ dT,\ irT) & \mathsf{mirPot}\ c = \mathsf{TreasuryMIR} \end{cases} \\ combined := (\mathsf{mirTarget}\ c) \cup_{+} irPot \\ \sum_{\_ \mapsto val \in combined} val \leq pot + deltaPot \\ \forall r \in \mathsf{range}\ combinedR,\ r \geq 0 \\ i'_{rwd} := \begin{cases} (combined,\ irT,\ dR,\ dT) & \mathsf{mirPot}\ c = \mathsf{ReservesMIR} \\ (irR,\ combined,\ dR,\ dT) & \mathsf{mirPot}\ c = \mathsf{TreasuryMIR} \end{cases} \end{array}}{slot \\ ptr \\ acnt \vdash \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix} \xrightarrow[\text{DELEG}]{c} \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i'_{rwd} \end{pmatrix}} \text{Deleg-Mir} \tag{8}$$

Figure 18: Move Instantaneous Rewards Inference Rule

$$\text{Deleg-Mir-Trans}\ \frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{mir}} \qquad \mathsf{mirTarget}\ c \in \mathsf{Coin} \\ slot < \mathsf{firstSlot}\ ((\mathsf{epoch}\ slot) + 1) - \mathsf{StabilityWindow} \\ (irR,\ irT,\ dR,\ dT) := i_{rwd} \\ coin := \mathsf{mirTarget}\ c \qquad coin \geq 0 \qquad (treasury,\ reserves) := acnt \\ available := \begin{cases} reserves + dR + \left(\sum\limits_{\_ \mapsto val \in irR} val\right) & \mathsf{mirPot}\ c = \mathsf{ReservesMIR} \\ treasury + dT + \left(\sum\limits_{\_ \mapsto val \in irT} val\right) & \mathsf{mirPot}\ c = \mathsf{TreasuryMIR} \end{cases} \\ coin \leq available \\ i'_{rwd} := \begin{cases} (irR,\ irT,\ dR - coin,\ dT + coin) & \mathsf{mirPot}\ c = \mathsf{ReservesMIR} \\ (irR,\ irT,\ dR + coin,\ dT - coin) & \mathsf{mirPot}\ c = \mathsf{TreasuryMIR} \end{cases} \end{array}}{\begin{array}{l} slot \\ ptr \\ acnt \end{array} \vdash \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i_{rwd} \end{pmatrix} \xrightarrow[\text{DELEG}]{c} \begin{pmatrix} rewards \\ delegations \\ ptrs \\ fGenDelegs \\ genDelegs \\ i'_{rwd} \end{pmatrix}} \tag{9}$$

Figure 19: MIR transfer Inference Rule

$$\text{MIR}\ \frac{\begin{array}{c} (rewards,\ delegations,\ ptrs,\ fGenDelegs,\ genDelegs,\ i_{rwd}) := ds \\ (treasury,\ reserves) := acnt \\ (irReserves,\ irTreasury\ ,\ deltaReserves,\ deltaTreasury\ ) := i_{rwd} \\ \\ irwdR := \{\mathsf{addr}_{\mathsf{rwd}}\ hk \mapsto val \mid hk \mapsto val \in (\mathrm{dom}\ rewards) \lhd irReserves\} \\ irwdT := \{\mathsf{addr}_{\mathsf{rwd}}\ hk \mapsto val \mid hk \mapsto val \in (\mathrm{dom}\ rewards) \lhd irTreasury\} \\ availableReserves := reserves + deltaReserves \\ availableTreasury := treasury + deltaTreasury \\ \\ totR := \sum_{\_ \mapsto v \in irwdR} v \qquad totT := \sum_{\_ \mapsto v \in irwdT} v \\ enough := totR \leq availableReserves \wedge totT \leq availableTreasury \\ acnt' := \begin{cases} (treasury - totT,\ reserves - totR) & enough \\ acnt & \text{otherwise} \end{cases} \\ rewards' := \begin{cases} rewards \cup_+ irwdR \cup_+ irwdT & enough \\ rewards & \text{otherwise} \end{cases} \\ ds' := (rewards',\ delegations,\ ptrs,\ fGenDelegs,\ genDelegs,\ (\emptyset,\ \emptyset,\ \mathbf{0},\ \mathbf{0})) \end{array}}{\vdash \begin{pmatrix} acnt \\ ss \\ (us,\ (ds,\ ps)) \\ prevPP \\ pp \end{pmatrix} \xrightarrow[\text{MIR}]{} \begin{pmatrix} acnt' \\ ss \\ (us,\ (ds',\ ps)) \\ prevPP \\ pp \end{pmatrix}} \tag{10}$$

Figure 20: MIR rules

References

Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *International Conference on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020. URL https://iohk.io/en/research/library/papers/the-extended-utxo-model/.

Edsko de Vries, Thomas Winant, and Duncan Coutts. The Cardano Consensus and Storage Layer. Technical report, IOHK, 2021. URL https://hydra.iohk.io/build/5265971/download/1/report.pdf.

Formal Methods Team, IOHK. A Formal Specification of a Multi-Signature Scheme using Scripts, 2019a. URL https://github.com/input-output-hk/cardano-ledger/blob/master/eras/shelley/formal-spec/multi-sig.tex.

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

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

Plutus Team. Formal Specification of the Plutus Core Language, 2021a. URL https://hydra.iohk.io/build/7244320/download/1/plutus-core-specification.pdf.

Plutus Team. The Plutus Platform: An IOHK technical report, 2021b. URL https://hydra.iohk.io/build/7244348/download/1/plutus.pdf.

A TxInfo Construction

This section specifies exactly what parts of the transaction and ledger state are used by the txInfo function to construct the TxInfo element that, together with the script purpose for which the script is being run, gets passed as a Data argument to the Plutus interpreter.

A.1 Type Translations

We give the Plutus types corresponding to the ledger counterparts in Figures 21 and 22. Details for types that have non-identity translation functions between ledger and Plutus types are in Figure 24.

Notation. In this Section we use the notation P.PlutusType for the Plutus type PlutusType (specified in the Plutus library) to distinguish it from ledger types.

If LT is a ledger type, and P.LT is the corresponding Plutus type to which elements of LT must be translated, we denote the translation function by:

$$\mathsf{toPlutusType}_{\mathsf{LT}} : \mathsf{LT} \to \mathsf{P.LT}$$

As a shorthand, we always write t_P for $\mathsf{toPlutusType}_{LT}\ t$, if t is of type LT.

In many cases this function is simply the identity function, and all the cases in which it is not the identity will be described below.

Untranslatable Types. Certain types, such as the bootstrap address type, cannot be passed to scripts, and are therefore translated as $\diamond$. Any type that cannot be fully translated as a Plutus-library type is also translated to $\diamond$. For example, since a bootstrap address $a \in \mathsf{Addr}_{\mathsf{bootstrap}}$ is not translatable, neither is $a \in \mathsf{Addr}$, and $(a, \_, \_) \in \mathsf{TxOut}$ also translates to $\diamond$.

Certificates. Translating certain kinds of certificates drops the data in the certificates, in particular, the $\mathsf{DCert}_{\mathsf{genesis}}$ and $\mathsf{DCert}_{\mathsf{mir}}$ ones. The $\mathsf{DCert}_{\mathsf{regpool}}$ pool registration certificate has a non-identity translation function, transPoolCert.

Time Translation. Functions needed to implement conversion of a slot number to POSIX time are given in Figure 23.

Pointer Address Resolution. Note that the Ptr addresses translated and passed to Plutus scripts are not resolvable to their corresponding key or script staking credencials. This is because doing a lookup in the resolution map, which part of the ledger state, could break determinism.

Well-Formed Scripts and Data. During the transaction encoding and decoding process, a transaction is discarded if it is not encoded correctly. This includes, in particular, a check that Script and Data elements contained in the transaction are well-formed. Functions that perfom these checks are in Figure 25. The P.validateScript function is a Plutus-library function which checks whether a bytestring represents a Plutus script.

A.2 Building Transaction Summary

The functions in Figure 26 are needed to build a Data summary of a transaction. The function P.toData converts a Plutus-library element into a Data element.

The txInfo **Function.** txInfo summarizes all the necessary transaction and chain state information that needs to be passed to the script interpreter. Below, we specify how to represent relevant transaction data in terms of a Plutus library-defined type. The txInfo function builds this representation.

The Language argument is required because different languages have different expectations of the format and contents of the TxInfo summary. The EpochInfo and SystemStart arguments are needed for translating slot numbers to POSIX time.

Note that txInfo has a UTxO argument. Even though the full ledger UTxO is passed to it, we define this function in such a way that the only entries in the ledger UTxO map that a Plutus

script actually sees via the argument txInfo builds are the ones corresponding to the transaction inputs (ie. its realized inputs). This is done in order to maintain the locality of evaluation. For details, see the deterministic script evaluation property C.8.

The valContext **Function.** valContext constructs the *validation context*, also referred to as the script context. A validation context is a Data element which encodes both the summary of the transaction and ledger information (this is supplied by the txInfo summarization function), and the script purpose.

Hash Types

ScriptHash	P.ValidatorHash
KeyHash	P.PubKeyHash
DataHash	P.DatumHash
TxId	P.TxId

Transaction Input, ie. Output Reference

TxIn	P.TxOutRef

Credential Types

Ptr	P.StakingPtr
Credential	P.Credential
Credential	P.StakingHash
$\text{Credential}_{stake}$	P.StakingCredential

Value Types

Wdrl	P.StakingCredential $\times$ Integer
PolicyID	P.CurrencySymbol
AssetName	P.TokenName
Coin	Integer
Quantity	Integer

Certificate Types

DCert	P.DCert
$\text{DCert}_{\text{regkey}}$	P.DCertDelegRegKey
$\text{DCert}_{\text{deregkey}}$	P.DCertDelegDeRegKey
$\text{DCert}_{\text{delegate}}$	P.DCertDelegDelegate
$\text{DCert}_{\text{retirepool}}$	P.DCertPoolRetire

Data-dropping Types

RewardAcnt	P.StakingHash
$\text{DCert}_{\text{genesis}}$	P.DCertGenesis
$\text{DCert}_{\text{mir}}$	P.DCertMir

Figure 21: TxInfo and Constituent Types

Script Purpose Types

Cert	P.Certifying
Rwrd	P.Rewarding
Mint	P.Minting
Spend	P.Spending

Execution Budget Types

Integer	P.ExCPU
Integer	P.ExMemory
ExUnits	P.ExBudget

TxInfo

$$
\begin{array}{lr}
\mathsf{TxInfo} = & \\
\quad \mathsf{P.TxInInfo}^{?*} & \text{realized inputs} \\
\quad \times \mathsf{P.TxOut}^{?*} & \text{outputs} \\
\quad \times \mathsf{P.Value} & \text{fee} \\
\quad \times \mathsf{P.Value} & \text{mint field} \\
\quad \times \mathsf{P.DCert}^{*} & \text{list of certificates} \\
\quad \times (\mathsf{P.StakingCredential} \times \mathsf{Integer}) & \text{reward withdrawal} \\
\quad \times \mathsf{P.POSIXTimeRange} & \text{validity interval in POSIX time} \\
\quad \times \mathsf{P.PubKeyHash}^{*} & \text{key hashes of required signatures} \\
\quad \times (\mathsf{DataHash} \times \mathsf{Data})^{*} & \text{list of datums and their hashes} \\
\quad \times \mathsf{P.TxId} & \text{transaction ID}
\end{array}
$$

Figure 22: TxInfo and Constituent Types

Abstract Types

$$scs \in \mathsf{Seconds} \qquad \text{time in seconds}$$

Abstract Conversion Functions

$$\mathsf{utcTimeToPOSIXSecondsUTCTime} \to \mathsf{Seconds}$$

Convert UTC Time to seconds

Rounding Functions

$$\mathsf{truncateFraction} \to \mathsf{Integer}$$

Round a fractional value to the nearest integer

Slot to Time Conversion

$$\mathsf{slotToPOSIXTime} \in \mathsf{PParams} \to \mathsf{EpochInfo} \to \mathsf{SystemStart} \to \mathsf{Slot} \to \mathsf{P.POSIXTime}$$

$$\mathsf{slotToPOSIXTime}\ pp\ ei\ sysS\ vs\ =$$

$$\mathsf{truncate}\ ((\mathsf{utcTimeToPOSIXSeconds}\ (\mathsf{epochInfoSlotToUTCTime}\ pp\ ei\ sysS\ vs))\ *\ 1000)$$

Translate validity interval to a POSIX time range

Figure 23: Types and Functions Used in Time Conversion

Conversion Functions

$$\mathsf{toPlutusType}_{\mathsf{Addr}} \in \mathsf{Addr} \to \mathsf{P.Address}$$

$$\mathsf{toPlutusType}_{\mathsf{Addr}}\ a = \begin{cases} \diamond & \text{if } a \in \mathsf{Addr}_{\mathsf{bootstrap}} \\ (ob,\ st) & \text{if } a = (\_, ob_P,\ st_P) \end{cases}$$

Address translation

$$\mathsf{toPlutusType}_{\mathsf{Value}} \in \mathsf{Value} \to \mathsf{P.Value}$$

$$\mathsf{toPlutusType}_{\mathsf{Value}}\ (c,\ mp) = \{\ pid_P \mapsto (aid_P \mapsto q_P) \mid pid \mapsto (aid \mapsto q) \in mp\ \}$$
$$\cup \{\ \mathsf{P.adaSymbol} \mapsto (\mathsf{P.adaToken} \mapsto c_P)\ \}$$

Value translation

$$\mathsf{toPlutusType}_{\mathsf{TxOut}} \in \mathsf{TxOut} \to \mathsf{P.TxOut}$$

$$\mathsf{toPlutusType}_{\mathsf{TxOut}}\ (a, v, h) = (a_P, v_P, h_P)$$

Output translation

$$\mathsf{toPlutusType}_{\mathsf{DCert}} \in \mathsf{DCert} \to \mathsf{P.DCert}$$

$$\mathsf{toPlutusType}_{\mathsf{DCert}}\ c = \begin{cases} (\mathsf{poolId}\ c,\ \mathsf{poolVrf}\ c) & \text{if } c \in \mathsf{DCert}_{\mathsf{regpool}} \\ c_P & \text{otherwise} \end{cases}$$

Certificate translation

$$\mathsf{transVITime} \in \mathsf{PParams} \to \mathsf{EpochInfo} \to \mathsf{SystemStart} \to \mathsf{ValidityInterval} \to \mathsf{P.POSIXTimeRange}$$

$$\mathsf{transVITime}\ pp\ ei\ sysS\ (vs,\ vf) =$$

$$\begin{cases} \mathsf{P.always} & \text{if } vs = \diamond \wedge vf = \diamond \\ \mathsf{P.to}\ (\mathsf{slotToPOSIXTime}\ pp\ ei\ sysS\ vf) & \text{if } vs = \diamond \wedge vf \neq \diamond \\ \mathsf{P.from}\ (\mathsf{slotToPOSIXTime}\ pp\ ei\ sysS\ vs) & \text{if } vs \neq \diamond \wedge vf = \diamond \\ (\mathsf{slotToPOSIXTime}\ pp\ ei\ sysS\ vs,\ \mathsf{slotToPOSIXTime}\ pp\ ei\ sysS\ vf) & \text{if } vs \neq \diamond \wedge vf \neq \diamond \end{cases}$$

Figure 24: TxInfo Constituent Type Translation Functions

$$\mathsf{validPlutusdata} \in \mathsf{P.Data} \to \mathsf{Bool}$$

$$\mathsf{validPlutusdata}\ (\mathsf{P.Constr}\ \_\ ds) = \bigwedge_{d \in ds} \mathsf{validPlutusdata}\ d$$

$$\mathsf{validPlutusdata}\ (\mathsf{P.Map}\ ds) = \bigwedge_{(x,y) \in ds} \mathsf{validPlutusdata}\ x \wedge \mathsf{validPlutusdata}\ y$$

$$\mathsf{validPlutusdata}\ (\mathsf{P.List}\ ds) = \bigwedge_{d \in ds} \mathsf{validPlutusdata}\ d$$

$$\mathsf{validPlutusdata}\ (\mathsf{P.I}\ \_) = \mathsf{True}$$

$$\mathsf{validPlutusdata}\ (\mathsf{P.B}\ bs) = \mathsf{length}\ bs \leq 64$$

Checks if a Data element is constructed correctly

$$\mathsf{validScript} \in \mathsf{Script} \to \mathsf{Bool}$$

$$\mathsf{validScript}\ sc = \begin{cases} \mathsf{True} & \text{if } sc \in \mathsf{Script}^{\mathsf{ph1}} \\ \mathsf{P.validateScript}\ sc & \text{if } sc \in \mathsf{Script}^{\mathsf{ph2}} \end{cases}$$

Checks if a script is constructed correctly

Figure 25: Script and Data construction correctness checks

Plutus Library Functions

$$\mathsf{P.toData} \in \mathsf{P.T} \to \mathsf{Data}$$

Constructs a Data element from a Plutus-library-type element of type P.T

Ledger Functions

$$\mathsf{txInfo} \in \mathsf{Language} \to \mathsf{PParams} \to \mathsf{EpochInfo} \to \mathsf{SystemStart} \to \mathsf{UTxO} \to \mathsf{Tx} \to \mathsf{TxInfo}$$

$$\begin{aligned}
&\mathsf{txInfo}\ pp\ ei\ sysS\ utxo\ tx = \\
&\quad (\{\ (txin_P, txout_P) \mid txin \in \mathsf{txinputs}\ tx,\ (txin \mapsto txout) \in utxo\ \}, \\
&\quad \{\ tout_P \mid tout \in \mathsf{txouts}\ tx\ \}, \\
&\quad (\mathsf{inject}\ (\mathsf{txfee}\ tx))_P, \\
&\quad (\mathsf{mint}\ tx)_P, \\
&\quad [\ c_P \mid c \in \mathsf{txcerts}\ tx\ ], \\
&\quad \{\ (s_P,\ c_P) \mid s \mapsto c \in \mathsf{txwdrls}\ tx\ \}, \\
&\quad \mathsf{transVITime}\ pp\ ei\ sysS\ (\mathsf{txvldt}\ tx), \\
&\quad \{\ k_P \mid k \in \mathsf{dom}\ \mathsf{txwitsVKey}\ tx\ \}, \\
&\quad \{\ (h_P,\ d_P) \mid h \mapsto d \in \mathsf{txdats}\ tx\ \}, \\
&\quad (\mathsf{txid}\ tx)_P)
\end{aligned}$$

Summarizes transaction data

$$\mathsf{valContext} \in \mathsf{P.TxInfo} \to \mathsf{ScriptPurpose} \to \mathsf{Data}$$

$$\mathsf{valContext}\ txinfo\ sp = \mathsf{P.toData}\ (txinfo,\ sp_P)$$

Pairs transaction data with a script purpose

Figure 26: Transaction Summarization Functions

B Output Size

Figure 27 gives the formula for calculating the size of a UTxO entry in the Alonzo era. In addition to the data found in the UTxO in the ShelleyMA era, the hash of a datum (or $\diamond$) is added to the output type, which is accounted for in the size calculation.

Constants

$$\mathsf{JustDataHashSize} \in \mathsf{MemoryEstimate}$$

The size of a datum hash wrapped in $\mathsf{DataHash}^?$

$$\mathsf{NothingSize} \in \mathsf{MemoryEstimate}$$

The size of $\diamond$ wrapped in $\mathsf{DataHash}^?$

Helper Functions

$$\mathsf{dataHashSize} \in \mathsf{DataHash}^? \to \mathsf{MemoryEstimate}$$
$$\mathsf{dataHashSize}\ \diamond = \mathsf{NothingSize}$$
$$\mathsf{dataHashSize}\ \_ = \mathsf{JustDataHashSize}$$

Return the size of $\mathsf{DataHash}^?$

$$\mathsf{utxoEntrySize} \in \mathsf{TxOut} \to \mathsf{MemoryEstimate}$$
$$\mathsf{utxoEntrySize}\ (a, v, d) = \mathsf{utxoEntrySizeWithoutVal} + (\mathsf{size}(\mathsf{getValue}\ (a, v, d))) + \mathsf{dataHashSize}\ d$$

Calculate the size of a UTxO entry

Figure 27: Value Size

NOTE
Get dataHashSize from heapwords on the code

There is a change of constant value from the ShelleyMA era, specifically:

- $\mathsf{adaOnlyUTxOSize} = 29$ instead of 27
- $k_0 = 2$ instead of 0

Additionally, the new constants used in Alonzo have values :

- $\mathsf{JustDataHashSize} = 10$ words
- $\mathsf{NothingSize} = 0$ words

C Formal Properties

This appendix collects the main formal properties that the new ledger rules are expected to satisfy. Note here that in every property in this section we consider a only phase-1 valid transactions, ie. ones that can indeed get processed.

P1: *Consistency with Shelley.*

- properties 15.6 - 15.16 (except 15.8 and 15.13) hold specifically in the txValTag $tx = \mathsf{True}$ case, because the calculations refer to the UTxO as if it was updated by a scripts-validating transaction
- other properties hold as-is

P2: *Consistency with Multi-Asset.*

- properties 8.1 and 8.2 hold specifically in the txValTag $tx = \mathsf{True}$ case, because the calculations refer to the UTxO as if it was updated by a scripts-validating transaction
- other properties hold as-is

Definition C.1. For a state s that is used in any subtransaction of CHAIN, we define $\mathsf{Utxo}(s) \in \mathsf{UTxO}$ to be the UTxO contained in s, or an empty map if it does not exist. This is similar to the definition of Val in the Shelley specification.

Similarly, we also define $\mathsf{field}_v\ (s)$ for the field v that is part of the ledger state s, referenced by the typical variable name (eg. *fees* for the fee pot on the ledger).

We also define a helper function φ as follows:

$$\varphi(x, tx) := \begin{cases} x & \mathsf{txValTag}\ tx = \mathsf{True} \\ 0 & \mathsf{txValTag}\ tx = \mathsf{False} \end{cases}$$

This function is useful to distinguish between the two cases where a transaction can mint tokens or not, depending on whether its scripts validate.

Property C.1 (General Accounting)**.** The *general accounting* property in Alonzo encompasses two parts

- preservation of value property expressed via the produced and consumed calculations, applicable for transactions with txValTag $tx = \mathsf{True}$, which is specified as in the ShelleyMA POV.
- the preservation of value for txValTag $tx = \mathsf{False}$, when only the collateral fees are collected into the fee pot.

Both of these are expressed in the following lemma.

Lemma C.2. For all environments e, transactions tx and states s, s', if

$$e \vdash s \xrightarrow[\text{UTXOS}]{tx} s'$$

then

$$\mathsf{Val}(s) + \varphi(\mathsf{mint}\ (\mathsf{txbody}\ tx), tx) = \mathsf{Val}(s')$$

Proof. In the case that txValTag tx = True holds, the proof is identical to the proof of Lemma 9.1 of the multi-asset specification. Otherwise, the transition must have been done via the Scripts − No rule, which removes collateral (txbody tx) from the UTxO, and increases the fee pot by the amount of the sum of Ada in the collateral UTxO entries. The value contained in s is not changed.

Note that in the feesOK function, there is a check that verifies that, in the case that there are phase-2 scripts, there are no non-Ada tokens in the UTxOs which the collateral inputs reference, so non-Ada tokens do not get minted, burned, or transferred in this case. □

Property C.2 (Extended UTxO Validation)**.** If a phase-1 valid transaction extends the UTxO, all its scripts validate (i.e. txValTag tx = True).

If a transaction is accepted and marked as paying collateral only (i.e. txValTag tx = False), then the only change to the ledger when processing the transaction is that the collateral inputs are moved to the fee pot. In particular, it does not extend the UTxO.

Lemma C.3. For all environments e, transactions tx and states s, s', if and

$$e \vdash s \xrightarrow[\text{LEDGER}]{tx} s'$$

then

$$\text{Utxo}(s') \not\subseteq \text{Utxo}(s) \;\Rightarrow\; \text{txValTag } tx = \text{True}$$

Also,

$$\text{txvaltag } tx = \text{False} \;\Rightarrow$$

- $\text{field}_v\ (s') = \text{field}_v\ (s)$ for all $v \neq fees,\ v \neq utxo$
- $\text{field}_{fees}\ (s') = \text{field}\ fees\ (s) + \text{coin}\ (\text{Val}\ (\text{collateral}\ tx \lhd \text{Utxo}(s)))$
- $\text{Utxo}(s') = \text{collateral}\ tx \lhd \text{Utxo}(s)$

Proof. A ledger UTxO update is specified explicitly only by the UTXOS transition, and propagated (ie. applied as-specified) to a chain-state update via the UTXO and UTXOW transitions.

The only case in which the UTXOS transition adds entries to the UTxO map is in the Scripts − No rule, when txValTag tx = True. Adding entries to the UTxO map implies exactly that the updated map is not a submap of the original, so we get that

$$\text{Utxo}(s') \not\subseteq \text{Utxo}(s) \;\Rightarrow\; \text{txValTag } tx = \text{True}$$

In the case that txValTag tx = False, LEDGER calls the rule that does not update DPState at all, only the UTxOState. This state update is specified in the UTXOS transition (and applied via the UTXO and UTXOW transitions).

The only parts of the state that are updated are

- the fee pot, which is increased by the amount of the sum of Ada in the collateral UTxO entries, and
- the UTxO, where the collateral entries are removed.

□

Property C.3 (Validating when No NN Scripts)**.** Whenever a valid transaction does not have any phase-2 scripts, its txValTag = True.

Lemma C.4. For all environments e, transactions tx and states s, s' such that

$$e \vdash s \xrightarrow[\text{LEDGER}]{tx} s',$$

If $\text{range}(\text{txscripts } tx) \cap \text{Script}^{\text{ph2}} = \emptyset$ then txValTag = True.

Proof. With the same argument as previously, we only need to discuss the equivalent claim for the UTXOS transition. Under these assumptions, $sLst$ is an empty list. Thus evalScripts $sLst = \mathsf{True}$, and the transition rule had to be Scripts−Yes. □

Property C.4 (Paying fees)**.** In the case that all phase-2 scripts in a transaction validate, the transaction pays at least the minimum transaction fee amount into the fee pot. In the case that some do not validate, it must pay at least the percentage of the minimum fee the collateral is required to cover.

Lemma C.5. For all environments e, transactions tx and states s, s' such that

$$e \vdash s \xrightarrow[\text{LEDGER}]{tx} s'$$

The following holds :

$$\mathsf{field}_{fees}\ (s') \ \geq\ \mathsf{field}_{fees}\ (s) + \begin{cases} \mathsf{minfee}\ tx & \mathsf{txValTag} = \mathsf{True} \\ \mathsf{quot}\ (\mathsf{collateralPercent}\ (\mathsf{field}_{pp}\ (s)))\ *\ (\mathsf{minfee}\ tx)\ 100 & \mathsf{txValTag} = \mathsf{False} \end{cases}.$$

Proof. The fee pot is updated by the Scripts−Yes and the Scripts−No rules, one of which is necessarily called by a valid transaction via the sequence of transitions called in the order LEDGER, UTXOW, UTXO, UTXOS.

The Scripts−Yes rule (i.e. txValTag = True) transfers the amount of Lovelace in the txfee field of the transaction to the fee pot, which is checked to be at least the minfee by feesOK in the UTXO transition. So, we get

$$\mathsf{field}_{fees}\ (s') \ =\ \mathsf{field}_{fees}\ (s) \ +\ \mathsf{txfee}\ tx \ \geq\ \mathsf{field}_{fees}\ (s) \ +\ \mathsf{minfee}\ tx$$

The Scripts−No rule (i.e. txValTag = False) removes the collateral inputs from the UTxO, and adds the balance of these realized inputs to the fee pot.

$$\mathsf{field}_{fees}\ (s') \ =\ field_{fees}\ (s) \ +\ \mathsf{ubalance}\ (\mathsf{collateral}\ txb \lhd utxo)$$

We can conclude that txrdmrs is non-empty whenever txValTag = False from the following observations :

- txValTag = False whenever evalScripts fails.
- evalScripts is a $\wedge$-fold over a list of scripts and their inputs, containing all scripts that need to be run to validate this transaction
- All phase-1 scripts must succeed, as this is checked in phase-1 validation (UTXOW rule check). Therefore, evalScripts encounters a failing script which is phase-2.
- All phase-2 scripts necessarily have an associated redeemer attached to the transaction (UTXOW rule check)

See Property C.8 for more details on what inputs evalScripts has, and what we can say about the outcome of its computation with respect to its inputs.

The feesOK check enforces that if a transaction has a non-empty txrdmrs field, the balance of the realized collateral inputs (which $\mathsf{field}_{fees}\ (s)$ is increased by as part of processing tx) is

$$\mathsf{ubalance}\ (\mathsf{collateral}\ txb \lhd utxo)\ \geq\ \mathsf{quot}\ (\mathsf{txfee}\ \ txb * (\mathsf{collateralPercent}\ pp))\ 100$$

which, since $\mathsf{txfee}\ \ txb\ \geq\ \mathsf{minfee}\ txb$, gives

$$\geq\ \mathsf{quot}\ (\mathsf{minfee}\ txb * (\mathsf{collateralPercent}\ pp))\ 100))$$

where txbody $tx = txb$. This shows that the total collateral that gets moved into the fee pot from the UTxO by the Scripts−No rule is at least the minimum transaction fee scaled by the collateral percent parameter.

$$\mathsf{field}_{fees}\ (s') \geq \mathsf{field}_{fees}\ (s) + \mathsf{quot}\ (\mathsf{minfee}\ txb * (\mathsf{collateralPercent}\ pp))\ 100))$$

□

An immediate corollary to this property is that when the collateralPercent is set to 100 or more, the transaction always pays at least the minimum fee. This, in turn, implies that it pays at least the fee that it has stated in the txfee field.

Corollary C.6. For all environments e, transactions tx and states s, s' such that

$$e \vdash s \xrightarrow[\text{LEDGER}]{tx} s' \ \wedge\ \mathsf{collateralPercent}\ (\mathsf{field}_{pp}\ (s)) \geq 100$$

The following holds :

$$\mathsf{field}_{fees}\ (s') \geq \mathsf{field}_{fees}\ (s) + \mathsf{minfee}\ tx$$

Proof. In both two cases in the lemma above, the field_{fees} field in the ledger state is increased by at least minfee tx :

- when txValTag = True, the lemma states already that
$$\mathsf{field}_{fees}\ (s') \geq \mathsf{field}_{fees}\ (s) + \mathsf{minfee}\ tx$$
- when txValTag = False, we have $\mathsf{field}_{fees}\ (s') \geq \mathsf{field}_{fees}\ (s) + \mathsf{quot}\ (\mathsf{minfee}\ txb * (\mathsf{collateralPercent}\ pp))\ 100$
$\geq \mathsf{field}_{fees}\ (s) + \mathsf{quot}\ (\mathsf{minfee}\ tx * 100)\ 100$
$= \mathsf{field}_{fees}\ (s) + \mathsf{minfee}\ tx$

□

Property C.5 (Correct tag)**.** The txValTag tx tag of a phase-1 valid transaction must match the result of the evalScripts function for that transaction.

Lemma C.7. For all environments e, transactions tx and states s, s' such that

$$e \vdash s \xrightarrow[\text{UTXOS}]{tx} s',$$

The following holds :

$$\mathsf{txValTag}\ tx = \mathsf{evalScripts}\ tx\ sLst$$

where

$$sLst := \mathsf{collectTwoPhaseScriptInputs\ EI\ SysSt\ field}_{pp}(s)\ tx\ \mathsf{Utxo}(s)$$

Proof. Inspecting the Scripts−Yes and the Scripts−No rules of the UTXOS transition, we see that both the txValTag and the result of evalScripts are True in the former, and False in the latter. □

Property C.6 (Replay protection)**.** A transaction always removes at least one UTxO entry from the ledger, which provides replay protection.

Lemma C.8. For all environments e, transactions tx and states s, s' such that

$$e \vdash s \xrightarrow[\text{LEDGER}]{tx} s',$$

The following holds : $\mathsf{Utxo}\ (s) \not\subseteq \mathsf{Utxo}\ (s')$.

Proof. The UTxO is updated by the Scripts−Yes and the Scripts−No rules, on of which is necessarily called by a valid transaction. Both of these rules remove UTxOs corresponding to a set of inputs.

In both cases, there is a check that the removed inputs set is non-empty. The Scripts−Yes rule of the UTXOS transition removes the UTxOs associated with the input set txinputs tx from the ledger. The txinputs tx set must be non-empty because there is a check in the UTXO rule (which calls UTXOS) that ensure this is true.

For the Scripts−No rule of the UTXOS transition removes the UTxOs associated with the input set collateral tx from the ledger. The collateral tx set must be non-empty because the feesOK function (called by the same rule that calls UTXO, UTXOS) ensures that in the case that the tx contains phase-2 scripts, collateral tx must be non-empty.

Note that by property C.3, phase-2 scripts must always be present if txValTag tx = False (that is, whenever Scripts−No rule is used). □

Property C.7 (UTxO-changing transitions)**.** The UTXOS transition fully specifies the change in the ledger UTxO for each transaction.

Lemma C.9. For all environments e, transitions TRNS, transactions tx and states s, s' such that

$$e \vdash s \xrightarrow[\text{TRNS}]{tx} s'$$

The following holds :

$$\mathsf{Utxo}(s) \neq \mathsf{Utxo}(s') \;\Rightarrow\; e' \vdash u \xrightarrow[\text{UTXOS}]{tx} u'$$

where $\mathsf{Utxo}(s) = \mathsf{Utxo}(u) \;\wedge\; \mathsf{Utxo}(s') = \mathsf{Utxo}(u')$ for some environment e', and states u, u'.

Proof. By inspecting each transition in this specification, as well as those inherited from the Shelley one, we see that a non-trivial update from the UTxO of s to that of s' must be done by UTXOS. Every other transition that changes the UTxO and has a signal of type Tx only applies the UTXOS. □

Property C.8 (Deterministic script evaluation)**.** The deterministic script evaluation property, also stated as "script interpreter arguments are fixed", is a property of the Alonzo ledger that guarantees that the outcome of phase-2 script validation in a (phase-1 valid) transaction depends only on the contents of that transaction, and not on when, by who, or in what block that transaction was submitted.

For this property, we make some assumptions about things that are outside the scope of this specification :

- The Plutus script interpreter is a pure function that receives only the arguments provided by the ledger when it is invoked by calling the runPLCScript function. We assume that this is also true in an implementation. In particular, the interpreter does not obtain any system randomness, etc.
- We do not need to check here that the hashes that are the keys of the txscripts and txdats fields match the computed hashes of the scripts and datum objects they index. This is because these hashes must be computed as part of the deserialization of a transaction (see the CDDL specification), instead of being transmitted as part of the transaction and then checked. We assume these hashes are correct.
- We assume that the consensus function epochInfoSlotToUTCTime for converting a slot interval into a system time interval is deterministic.

- We assume that the two global constants, EpochInfo and SystemStart, which it also takes as parameters, cannot change within the span of any interval for which epochInfoSlotToUTCTime is able to convert both endpoints to system time.

The assumptions above are implicit in the functional style of definitions in this specification, but they are worth pointing out for implementation. With these assumptions in mind, we can say that script evaluation is deterministic.

We split this result into a lemma and its corollary :

- First, we demonstrate that all the scripts and their arguments that are collected for phase-2 validation are the same for two phase-1 valid transactions with the same body, independent of the (necessarily valid) ledger state to which they are being applied
- Second, we derive the corollary that under the same validity assumptions, phase-2 validation results in the same outcome for both transactions

Lemma C.10. For all environments e, e', transactions tx, tx' and states s, s', u, u' such that e and s are subsets of fields of some valid chain state c, and e' and s' are subsets of fields of some valid chain state c',

$$e \vdash s \xrightarrow[\text{LEDGER}]{tx} s', e' \vdash u \xrightarrow[\text{LEDGER}]{tx'} u', \mathsf{txbody}\ tx = \mathsf{txbody}\ tx'$$

The following holds :

$$\mathsf{toSet}\ (\mathsf{collectTwoPhaseScriptInputs}\ \mathsf{EI}\ \mathsf{SysSt}\ \mathsf{field}_{pp}(s)\ tx\ \mathsf{Utxo}(s))$$
$$=$$
$$\mathsf{toSet}\ (\mathsf{collectTwoPhaseScriptInputs}\ \mathsf{EI}\ \mathsf{SysSt}\ \mathsf{field}_{pp}(u)\ tx'\ \mathsf{Utxo}(u))$$

Proof. The collectTwoPhaseScriptInputs function (see 7) constructs a list where each entry contains a Plutus script and the arguments passed to the interpreter for its evaluation.

To show that collectTwoPhaseScriptInputs returns a list containing the same elements for both tx and tx', we observe that the set of elements from which this list is produced via toList is generated using the following functions, as well as set comprehension and list operations. We demonstrate that the functions used in this definition produce the same output for tx and tx', as all the data they inspect is fixed by the transaction body :

- scriptsNeeded : The PolicyID, $\mathsf{Addr}_{\mathsf{rwd}}$, and DCert data output by this function as the second term of the hash-purpose pair is obtained directly from the mint, txwdrls, txcerts fields of the transaction, which are all fixed by the body of the transaction. These types of script purposes all include the hash of the validating script.

  The only other data this function inspects is the UTxO entries associated with the txinputs (passed via the UTxO argument) to get the realized inputs. We know that the UTxO is a field in a valid chain state for both the phase-1 valid transactions tx and the tx' ($\mathsf{Utxo}(s)$ and $\mathsf{Utxo}(u)$, respectively). This means that the TxId in each input present in either UTxO is a hash of the body of the transaction containing the TxOut part of the UTxO entry indexed by that input. The order of the outputs is also fixed by the body, which fixes the Ix of the entry.

  A different value in output part of the UTxO entry (or a different order of outputs) would necessarily imply that the hash of the body containing that output must be different. Therefore, for all Plutus script-locked realized inputs of either transaction, the script hash in the payment credential of the address (and, by the same argument, the datum hash) are

fixed by the inputs in body of the transaction, despite not being explicitly contained in it. We then get that

$$\mathsf{scriptsNeeded}\ \mathsf{Utxo}(s)\ (\mathsf{txbody}\ tx) = \mathsf{scriptsNeeded}\ \mathsf{Utxo}(u)\ (\mathsf{txbody}\ tx')$$

- getDatum : In the case the script purpose is of the input type, the datum this function returns is one that is associated with the corresponding realized input. More precisely, it is the datum whose hash is specified in the realized input, and is looked up by hash in the txdats transaction field. If there is no datum hash in the realized input, or the script purpose is of a non-input type, the empty list is returned.

  The UTXOW rule checks that the transaction is carrying all datums corresponding to its realized inputs. Since the inputs (and realized inputs) are the same for tx and tx' (fixed by the body), this guarantees that of the datum hashes in the realized inputs (and therefore, their preimages) are the same as well.

- txscripts : in the UTXOW rule, there is a check that all the script preimages of the script hashes returned by the scriptsNeeded function must be present in this field.

- indexedRdmrs : like scriptsNeeded, this function examines four fields fixed by the transaction body (mint, txwdrls, txcerts, and txinputs). It also looks at data in the txrdmrs field, which is fixed by the transaction body via the scriptIntegrityHash hash. This is done as follows: the UTXOW rule verifies that this hash-containing field matches the computed hash of the preimage constructed from several fields of the transaction, including txrdmrs (this calculation is done by the hashScriptIntegrity function).

- language : this is directly conveyed by the type of a script.

- costmdls : The hash calculated by the hashScriptIntegrity function and compared to the hash value in the scriptIntegrityHash field must include in the preimage the current cost models of all script languages of scripts carried by the transaction. Recall that if a cost model changed between when a transaction was submitted and the time at which it was processed, the field and the calculated hash values will not match.

- valContext and txInfo : The valContext function is defined in A and is pure. All fields of the TxInfo structure, with the exceptions listed below, are straightforward translations of the corresponding transaction body fields (see A) that are given no additional arguments, and therefore completely determined by tx and tx'. The fields not directly appearing in the body are :

  – txInfoInputs : this field contains the realized inputs of the transaction which are fixed by the transaction and the unique UTxO entries to which the inputs correspond. As explained above, accessing information in realized inputs for script evaluation does not break determinism.
  – txInfoValidRange : this field contains the transaction validity interval as system time (converted from the slot numbers, which are used to specify the interval in the transaction body). This conversion is done by a function defined in the consensus layer, and takes two global constants in addition to the slot interval itself. Since the slot interval conversion function epochInfoSlotToUTCTime necessarily succeeds if both tx and tx' pass phase-1 validation, the additional global constant arguments must be the same (by assumption). The determinism of this conversion is one of the assumptions of this property, and thus gives the same output for both transactions.
  – txInfoData : this field is populated with the datums (and their hashes) in the transaction field txdats, which are fixed by the body via the scriptIntegrityHash field.

– txInfold : this field contains the hash of the transaction body, which is clearly the same for transactions with the same body.

Therefore, each field of the TxInfo structure is the same for two transactions with the same body, ie.

$$\mathsf{txInfo}\ \mathsf{PlutusV1}\ \mathsf{Utxo}(s)\ tx = \mathsf{txInfo}\ \mathsf{PlutusV1}\ \mathsf{Utxo}(u)\ tx'$$

□

We can now make the general statement about evaluation of all scripts done in phase-2 validation : for any phase-1 valid transactions with the same body, the outcome of phase-2 script evaluation is the same. We make the same assumptions as in Lemma C.10.

Corollary C.11. For all environments e, e', transactions tx, tx' and states s, s', u, u' such that

$$e \vdash s \xrightarrow[\text{LEDGER}]{tx} s', e' \vdash u \xrightarrow[\text{LEDGER}]{tx'} u', \mathsf{txbody}\ tx = \mathsf{txbody}\ tx'$$

The following holds :

$$\mathsf{evalScripts}\ tx\ (\mathsf{collectTwoPhaseScriptInputs}\ \mathsf{EI}\ \mathsf{SysSt}\ \mathsf{field}_{pp}(s)\ tx\ \mathsf{Utxo}(s))$$
$$=$$
$$\mathsf{evalScripts}\ tx'\ (\mathsf{collectTwoPhaseScriptInputs}\ \mathsf{EI}\ \mathsf{SysSt}\ \mathsf{field}_{pp}(u)\ tx'\ \mathsf{Utxo}(u))$$

Proof. Let us consider the use of arguments of evalScripts (see Figure 7). The first argument (of type Tx) is only inspected in the case that the script sc (the first element in the pair at the head of the list of script-arguments pairs is a phase-1 script. Since all phase-1 scripts are checked in phase one of validation (see 13) by calling validateScript on all scripts attached to the transaction. For this to apply to sc, we must also show that sc is a script attached to the transaction (see the second argument explanation). Note also that the tx argument passed to evalScripts at the use site (in the UTXOS transition) is the same, unmodified tx as is the signal for the LEDGER transition. We verify this by inspecting the sequence of transitions through which it is propagated (UTXOS, UTXO, UTXOW, and LEDGER), and their signals.

This allows us to conclude that at every step of the iteration over the script-arguments pairs list, the first argument to evalScripts,

- has no impact on the outcome of script evaluation in the case the script being validated at this step is phase-2, as it is completely ignored, and,
- because collectTwoPhaseScriptInputs filters out all phase-1 scripts, is, in fact, ignored always.

The second argument to evalScripts, ie. the list of scripts and their arguments, has already been shown to contain the same tuples for both transactions in the lemma above. The order of the list does not affect the validation outcome, since the interpreter is run on each tuple of a script and its arguments independently of all other tuples in the list. The function evalScripts is a $\wedge$-fold over the list. Thus, we may ignore the order of the elements in the generated list as it does not affect the evaluation outcome.

From this we may conclude that the outcome of both phase-1 and phase-2 script evaluations at each step of evalScripts must be the same for tx and tx'. Therefore, the $\wedge$-fold of them done by evalScripts also produces the same outcome for both transactions.

□

1. *Commutativity of Translation.* Translate, then apply to Alonzo ledger is the same as apply to MA ledger, then translate the ledger.
2. *Zero ExUnits.* If a script is run (there's a redeemer/exunits) , it will fail with 0 units
3. *Cost Increase.* if a transaction is valid, it will remain valid if you increase the execution units
4. *Cost Lower Bound.* if a transaction contains at least one valid script, it must have at least one execution unit
5. *Tx backwards Compatibility.* Any transaction that was accepted in a previous version of the ledger rules has exactly the same cost and effect, except that the transaction output is extended.
6. *Run all scripts.* All scripts attached to a transaction are run
7. ... Anything else?