A Formal Specification of the Cardano Ledger

(git revision 1.1-486-g301fede)

Jared Corduan
jared.corduan@iohk.io

Polina Vinogradova
polina.vinogradova@iohk.io

Matthias Güdemann
matthias.gudemann@iohk.io

February 20, 2019

Abstract

This documents defines the rules for extending a ledger with transactions. The transactions will affect both UTxO and stake delegation. It is intended to serve as the specification for random generators of transactions which adhere to the rules presented here.

List of Contributors

Nicolás Arqueros, Nicholas Clarke, Duncan Coutts, Ruslan Dudin, Sebastien Guillemot, Vincent Hanquez, Ru Horlick, Michael Hueschen, Philipp Kant, Jean-Christophe Mincke, Damian Nadales, Nicolas Di Prima.

Contents

List of Figures

1 Introduction

This document is a formal specification of the functionality of the ledger on the blockchain. The blockchain layer of the protocol and the interaction between the ledger and the blockchain layer is presented in a separate document, see Formal Methods Team (TODO). The details of the background and the larger context for the design decisions formalized in this document are presented in Kant et al. (2018)

In this work, we present important properties any implementation of the ledger must have. Specifically, we model the following aspects of the functionality of the ledger on the blockchain:

Preservation of value Every coin in the system is accounted for, and the total amount is unchanged by every transaction and epoch change. In particular, every coin is accounted for by one of the following categories:

- Circulation (UTxO)
- Deposit pot
- Fee pot
- Reserves (monetary expansion)
- Rewards (account addresses)
- Reward pot (undistributed)
- Treasury

Witnesses Authentication of parts of the transaction data by means of cryptographic entities (such as signatures and private keys) contained in these transactions.

Delegation Validity of delegation certificates, which delegate block-signing rights.

Stake Staking rights associated to an address.

While the blockchain protocol is a reactive system driven by the arrival of blocks causing updates to the ledger, the formal description is a collection of rules which is a static description of what a *valid ledger* is. The specifics of the semantics we use to define and apply the rules we present in this document are explained in detail in Formal Methods Team (2018). A valid ledger state can only reached by applying a sequence of inference rules, and any valid ledger state is reachable by applying some sequence of these rules.

The structure of the rules we give here is such that their application is deterministic. That is, given a specific initial state and relevant environmental constants, there is no ambiguity about which rule should be applied at any given time (i.e. which state transition is allowed take place). This is an important property which reflects the reality of the implementation — the blockchain evolves in a particular way given some user activity and the passage of time, and its behaviour is never unexpected.

2 Notation

The transition system is explained in Formal Methods Team (2018).

Powerset Given a set X, $\mathbb{P}\,\mathsf{X}$ is the set of all the subsets of X.

Sequences Given a set X, X^* is the set of sequences having elements taken from X. The empty sequence is denoted by ϵ, and given a sequence Λ, $\Lambda;\mathsf{x}$ is the sequence that results from appending $\mathsf{x} \in \mathsf{X}$ to Λ.

Functions $A \to B$ denotes a **total function** from A to B. Given a function f we write $f\ a$ for the application of f to argument a.

Inverse Image Given a function $f : A \to B$ and $b \in B$, we write $f^{-1}\ b$ for the **inverse image** of f at b, which is defined by $\{a \mid fa = b\}$.

Maps and partial functions $A \mapsto B$ denotes a **partial function** from A to B, which can be seen as a map (dictionary) with keys in A and values in B. Given a map $m \in A \mapsto B$, notation $a \mapsto b \in m$ is equivalent to $m\ a = b$.

Map Operations Figure 1 describes some non-standard map operations.

Relations A relation on $A \times B$ is a subset of $A \times B$. Both maps and functions can be thought of as relations. A function $f : A \to B$ is a relation consisting of pairs $(a, f(a))$ such that $a \in A$. A map $m : A \mapsto B$ is a relation consisting of pairs (a, b) such that $a \mapsto b \in m$. Given a relation R on $A \times B$, we define the inverse relation R^{-1} to be all pairs (b, a) such that $(a, b) \in R$. Similarly, given a function $f : A \to B$ we define inverse relation f^{-1} to consist of all pairs $(f(a), a)$. Finally, given two relations $R \subseteq A \times B$ and $S \subseteq B \times C$, we define the compostion $R \circ S$ to be all pairs (a, c) such that $(a, b) \in R$ and $(b, c) \in S$ for some $b \in B$.

In Figure 1, we specify the notation we use in the rest of the document.

$$
\begin{array}{rll}
set \lhd map = \{k \mapsto v \mid k \mapsto v \in map,\ k \in set\} & & \text{domain restriction} \\
set \not\lhd map = \{k \mapsto v \mid k \mapsto v \in map,\ k \notin set\} & & \text{domain exclusion} \\
map \rhd set = \{k \mapsto v \mid k \mapsto v \in map,\ v \in set\} & & \text{range restriction} \\
map \not\rhd set = \{k \mapsto v \mid k \mapsto v \in map,\ v \notin set\} & & \text{range exclusion} \\
A \triangle B = (A \setminus B) \cup (B \setminus A) & & \text{symmetric difference} \\
M \underset{\rightarrow}{\cup} N = (\mathrm{dom}\, N \not\lhd M) \cup N & & \text{union override right} \\
M \underset{\leftarrow}{\cup} N = M \cup (\mathrm{dom}\, M \not\lhd N) & & \text{union override left} \\
M \cup_+ N = (M \triangle N) \cup \{k \mapsto v_1 + v_2 \mid k \mapsto v_1 \in M \wedge k \mapsto v_2 \in N\} & & \text{union override plus (for monoidal values)}
\end{array}
$$

Figure 1: Non-standard map operators

3 Cryptographic primitives

figure 2 introduces the cryptographic abstractions used in this document. we begin by listing the abstract types, which are meant to represent the corresponding concepts in cryptography. Only the functionality explicitly stated in the figures below is assumed within the scope of this paper. That is, their exact implementation remains open to interpretation, and we do not rely on any additional properties derived from the study or implementation of public key cryptography outside this work. The types and rules we give here are needed in order to guarantee certain security properties of the delegation process, which we discuss later.

The cryptographic concepts required for the formal definition of witnessing include public-private key pairs, one-way functions and signatures. The constraint we introduce states that a signature of some data signed with a (private) key is only correct whenever we can verify it using the corresponding public key.

Besides basic cryptographic abstractions, we also make use of some abstract data storage properties in this document in order to build necessary definitions and make judgement calls about them.

Abstract data types in this paper are essentially placeholders with names indicating the data types they are meant to represent in an implementation. Derived types are made up of data structures (i.e. products, lists, finite maps, etc.) built from abstract types. The underlying structure of a data type is implementation-dependent, and furthermore, the way the data is stored on physical storage can vary as well.

Serialization is a physical manifestation of data on a given storage device. In this document, the properties and rules we state involving serialization are assumed to hold true independently of the storage medium and style of data organization chosen for an implementation.

Abstract types

$$\begin{array}{rcll} sk & \in & \mathsf{SKey} & \text{private signing key} \\ vk & \in & \mathsf{VKey} & \text{public verifying key} \\ hk & \in & \mathsf{HashKey} & \text{hash of a key} \\ \sigma & \in & \mathsf{Sig} & \text{signature} \\ d & \in & \mathsf{Data} & \text{data} \end{array}$$

Derived types

$$(sk, vk) \in \mathsf{KeyPair} \quad \text{signing-verifying key pairs}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{hashKey} & \in & \mathsf{VKey} \to \mathsf{HashKey} & \text{hashKey function} \\ \mathsf{verify} & \in & \mathbb{P}\ (\mathsf{VKey} \times \mathsf{Data} \times \mathsf{Sig}) & \text{verification relation} \\ \mathsf{sign} & \in & \mathsf{SKey} \to \mathsf{Data} \to \mathsf{Sig} & \text{signing function} \end{array}$$

Constraints

$$\forall (sk, vk) \in \mathsf{KeyPair},\ d \in \mathsf{Data},\ \sigma \in \mathsf{Sig} \cdot \mathsf{sign}\ sk\ d = \sigma \implies \mathsf{verify}\ vk\ d\ \sigma$$

Notation for serialized and verified data

$$\begin{array}{ll} [\![x]\!] & \text{serialised representation of } x \\ \mathcal{V}_{vk}[\![d]\!]_\sigma = \mathsf{verify}\ vk\ d\ \sigma & \text{shorthand notation for verify} \end{array}$$

Figure 2: Cryptographic definitions

4 Addresses

Addresses are described in section 4.2 of the delegation design document Kant et al. (2018). The types needed for the addresses are defined in Figure 3. There are three types of UTxO addresses:

- Base addresses, $\mathsf{Addr}_{\mathsf{base}}$, containing the hash of a payment key and the hash of a staking key,
- Pointer addresses, $\mathsf{Addr}_{\mathsf{ptr}}$, containing the hash of a payment key and a pointer to a stake key registration certificate,
- Enterprise addresses, $\mathsf{Addr}_{\mathsf{enterprise}}$, containing only the hash of a payment key (and which have no staking rights).

Together, these three address types make up the Addr type, which will be used in transaction outputs in Section 7.

Note that for security, privacy, and usability reasons, the staking (delegating) key pair associated with an address should be different from its payment key pair. Before the stake key is registered and delegated to an existing stake pool, the payment key can be used for transactions, though it will not receive rewards from staking. Once a stake key is registered, the shorter pointer addresses can generated.

Finally, there is an account style address $\mathsf{Addr}_{\mathsf{rwd}}$ which contains the hash of a staking key. These account addresses will only be used for receiving rewards from the proof of stake leader election. Apendix A of Kant et al. (2018) explains this design choice. The mechanism for transferring rewards from these accounts will be explained in Section 7, and follows Zahnentferner (2018).

Abstract types

$$\begin{array}{rcll} \mathit{slot} & \in & \mathsf{Slot} & \text{absolute slot} \\ \mathit{ix} & \in & \mathsf{Ix} & \text{index} \end{array}$$

Derived types

$$\begin{array}{rcll} (s,t,c) \in \mathsf{Ptr} & = & \mathsf{Slot} \times \mathsf{Ix} \times \mathsf{Ix} & \text{certificate pointer} \\ \mathit{addr} \in \mathsf{Addr}_{\mathsf{base}} & = & \mathsf{HashKey}_{\mathit{pay}} \times \mathsf{HashKey}_{\mathit{stake}} & \text{base address} \\ \mathit{addr} \in \mathsf{Addr}_{\mathsf{ptr}} & = & \mathsf{HashKey}_{\mathit{pay}} \times \mathsf{Ptr} & \text{pointer address} \\ \mathit{addr} \in \mathsf{Addr}_{\mathsf{enterprise}} & = & \mathsf{HashKey}_{\mathit{pay}} & \text{enterprise address} \\ \mathit{addr} \in \mathsf{Addr} & = & \mathsf{Addr}_{\mathsf{base}} \uplus \mathsf{Addr}_{\mathsf{ptr}} \uplus \mathsf{Addr}_{\mathsf{enterprise}} & \text{output address} \\ \mathit{acct} \in \mathsf{Addr}_{\mathsf{rwd}} & = & \mathsf{HashKey}_{\mathit{stake}} & \text{reward account} \end{array}$$

Accessor Functions

$$\begin{array}{rcll} \mathsf{paymentHK} & \in & \mathsf{Addr} \to \mathsf{HashKey}_{\mathit{pay}} & \text{hash of payment key from addr} \\ \mathsf{stakeHK_b} & \in & \mathsf{Addr}_{\mathsf{base}} \to \mathsf{HashKey}_{\mathit{stake}} & \text{hash of stake key from base addr} \\ \mathsf{stakeHK_r} & \in & \mathsf{Addr}_{\mathsf{rwd}} \to \mathsf{HashKey}_{\mathit{stake}} & \text{hash of stake key from reward account} \\ \mathsf{addrPtr} & \in & \mathsf{Addr}_{\mathsf{ptr}} \to \mathsf{Ptr} & \text{pointer from pointer addr} \end{array}$$

Constructor Functions

$$\mathsf{addr}_{\mathsf{rwd}} \in \mathsf{HashKey}_{\mathit{stake}} \to \mathsf{Addr}_{\mathsf{rwd}} \quad \text{construct a reward account}$$

Figure 3: Definitions used in Addresses

5 Protocol Parameters

The rules for the ledger depend on several parameters and are contained in the PParams type defined in Figure 4.

The type Coin is defined as an alias for the integers. Negative values will not be allowed in UTxO outputs or reward accounts, and $\mathbb{Z}$ is only chosen over $\mathbb{N}$ for its additive inverses.

The minfee function calculates the minimum fee that must be paid by a transaction. This value depends on the protocol parameters and the size of the transaction.

Two time related types are introduced, Epoch and Duration. A Duration is the difference between two slots, as given by $-_s$.

One global constant is defined, SlotsPerEpoch, representing the number of slots in an epoch. As a global constant, this value can only be changed by updating the software.

Lastly, there are two functions, epoch and firstSlot for converting between epochs and slots.

Abstract types

$$\begin{array}{rcll} \mathit{fparams} & \in & \mathsf{FeeParams} & \text{min fee parameters} \\ \mathit{dur} & \in & \mathsf{Duration} & \text{difference between slots} \\ \mathit{epoch} & \in & \mathsf{Epoch} & \text{epoch} \end{array}$$

Derived types

$$\mathit{coin} \in \mathsf{Coin} = \mathbb{Z} \quad \text{unit of value}$$

Protocol Parameters

$$\mathsf{PParams} = \left(\begin{array}{rcll} \mathit{fparams} & \in & \mathsf{FeeParams} & \text{min fee parameters} \\ \mathit{keyDeposit} & \in & \mathsf{Coin} & \text{stake key deposit} \\ \mathit{keyMinRefund} & \in & [0,\ 1] & \text{stake key min refund} \\ \mathit{keyDecayRate} & \in & [0,\ \infty) & \text{stake key decay rate} \\ \mathit{poolDeposit} & \in & \mathsf{Coin} & \text{stake pool deposit} \\ \mathit{poolMinRefund} & \in & [0,\ 1] & \text{stake pool min refund} \\ \mathit{poolDecayRate} & \in & [0,\ \infty) & \text{stake pool decay rate} \\ \mathit{movingAvgWeight} & \in & [0,\ 1] & \text{moving average weight} \\ \mathit{movingAvgExp} & \in & (0,\ \infty) & \text{moving average exponent} \\ E_{max} & \in & \mathsf{Epoch} & \text{epoch bound on pool retirement} \\ n_{opt} & \in & \mathbb{N}^+ & \text{desired number of pools} \\ a_0 & \in & (0,\ \infty) & \text{pool influence} \\ \tau & \in & [0,\ 1] & \text{treasury expansion} \\ \rho & \in & [0,\ 1] & \text{monetary expansion} \end{array} \right)$$

Accessor Functions

fparams, keyDeposit, keyMinRefund, keyDecayRate, poolDeposit, poolMinRefund, poolDecayRate, movingAvgWeight, movingAvgExp, emax, nopt, influence, tau, rho

Abstract Functions

$$\begin{array}{rcll} \mathsf{minfee} & \in & \mathsf{PParams} \to \mathsf{Tx} \to \mathsf{Coin} & \text{minimum fee calculation} \\ (\ -_s\ ) & \in & \mathsf{Slot} \to \mathsf{Slot} \to \mathsf{Duration} & \text{duration between slots} \end{array}$$

Global Constants

$$\mathsf{SlotsPerEpoch} \in \mathbb{N} \quad \text{slots per epoch}$$

Derived Functions

$$\mathsf{epoch} \in \mathsf{Slot} \to \mathsf{Epoch} \qquad \text{epoch of a slot}$$
$$\mathsf{epoch}\ \mathit{slot} = \mathit{slot}\ \mathsf{div}\ \mathsf{SlotsPerEpoch}$$

$$\mathsf{firstSlot} \in \mathsf{Epoch} \to \mathsf{Slot} \qquad \text{first slot of an epoch}$$
$$\mathsf{firstSlot}\ e = e \cdot \mathsf{SlotsPerEpoch}$$

Figure 4: Definitions used in Protocol Parameters

6 Transactions

Transactions are defined in Figure 5. A transaction body, TxBody, is made up of six pieces:

- A set of transaction inputs. The TxIn derived type identifies an output from a previous transaction. It consists of a transaction id and an index to uniquely identify the output.
- An indexed collection of transaction outputs. The TxOut type is an address paired with a coin value.
- A list of certificates, which will be explained in detail in Section 8.
- A transaction fee. This value will be added to the fee pot and eventually handed out as stake rewards.
- A time to live. A transaction will be deemed invalid if processed after this slot.
- A mapping of reward account withdrawals. The type Wdrl is a finite map that maps a reward address to the coin value to be withdrawn. The coin value must be equal to the full value contained in the account. Explicitly stating these values ensures that error messages can be precise about why a transaction is invalid.

A transaction, Tx, is a transaction body together with:

- A collection of witnesses, represented as a finite map from payment verification keys to signatures.

Additionally, the UTxO type will be used by the ledger state to store all the unspent transaction outputs. It is a finite map from transaction inputs to transaction outputs that are available to be spent.

Finally, txid computes the transaction id of a given transaction. This function must produce a unique id for each unique transaction.

Abstract types

$$txid \in \mathsf{TxId} \quad \text{transaction id}$$

Derived types

$$\begin{array}{rcll}
(txid, ix) & \in \mathsf{TxIn} & = & \mathsf{TxId} \times \mathsf{Ix} & \text{transaction input} \\
(addr, c) & \in \mathsf{TxOut} & = & \mathsf{Addr} \times \mathsf{Coin} & \text{transaction output} \\
utxo & \in \mathsf{UTxO} & = & \mathsf{TxIn} \mapsto \mathsf{TxOut} & \text{unspent tx outputs} \\
wdrl & \in \mathsf{Wdrl} & = & \mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin} & \text{reward withdrawal}
\end{array}$$

Transaction Types

$$\begin{array}{rcll}
txbody & \in \mathsf{TxBody} & = & \mathbb{P}\ \mathsf{TxIn} \times (\mathsf{Ix} \mapsto \mathsf{TxOut}) \times \mathsf{DCert}^* \times \mathsf{Coin} \times \mathsf{Slot} \times \mathsf{Wdrl} \\
tx & \in \mathsf{Tx} & = & \mathsf{TxBody} \times (\mathsf{VKey} \mapsto \mathsf{Sig})
\end{array}$$

Accessor Functions

$$\begin{array}{rcll}
\mathsf{txins} & \in \mathsf{Tx} \to \mathbb{P}\ \mathsf{TxIn} & & \text{transaction inputs} \\
\mathsf{txouts} & \in \mathsf{Tx} \to (\mathsf{Ix} \mapsto \mathsf{TxOut}) & & \text{transaction outputs} \\
\mathsf{txcerts} & \in \mathsf{Tx} \to \mathsf{DCert}^* & & \text{delegation certificates} \\
\mathsf{txfee} & \in \mathsf{Tx} \to \mathsf{Coin} & & \text{transaction fee} \\
\mathsf{txttl} & \in \mathsf{Tx} \to \mathsf{Slot} & & \text{time to live} \\
\mathsf{txwdrls} & \in \mathsf{Tx} \to \mathsf{Wdrl} & & \text{withdrawals} \\
\mathsf{txbody} & \in \mathsf{Tx} \to \mathsf{TxBody} & & \text{transaction body} \\
\mathsf{txwits} & \in \mathsf{Tx} \to (\mathsf{VKey} \mapsto \mathsf{Sig}) & & \text{witnesses}
\end{array}$$

Abstract Functions

$$\mathsf{txid} \in \mathsf{Tx} \to \mathsf{TxId} \quad \text{compute transaction id}$$

Figure 5: Definitions used in the UTxO transition system

7 UTxO

A key constraint that must always be satisfied as a result and precondition of a valid ledger state transition is called the *general accounting property*, or the *preservation of value* condition. Every piece of software that is a part of the implementation of the Cardano cryptocurrency must function in such a way as to not result in a violation of this rule. If this condition is not satisfied, it is an indicator of incorrect accounting, potentially due to malicious disruption or a bug.

The preservation of value is expressed as an equality that uses values in the ledger state and the environment, as well as the values in the body of the signal transaction. We have defined the rules of the delegation protocol in a way that should consistently satisfy the preservation of value. In the future, we hope to give a formally-verified proof that every *valid* ledger state satisfies this property.

In this section, we discuss the relevant accounting that needs to be done as a result of processing a transaction, i.e. the deposits for all certificates, transaction fees, transaction withdrawals, and refunds for individual deregistration, so that we may keep track of whether the preservation of value is satisfied. Stake pool retirement refunds are not triggered by a transaction (but rather, happen at the epoch boundary), and are therefore not considered in our state change rules invoked due to a signal transaction.

Note, that when a transaction is issued by a wallet to be applied to the ledger state (i.e. processed), we define the rules in this section in such a way that it is impossible to apply only some parts of a transaction (e.g. only certain certificates). Every part of the transaction must be valid and it must be live, otherwise it is ignored entirely. It is the wallet's responsibility to inform the user that a transaction failed to be processed.

7.1 UTxO Transitions

Figure 6 defines functions needed for the UTxO transition system.

- The function outs creates unspent outputs generated by a transaction, so that they can be added to the ledger state. For each output in the transaction, outs maps the transaction id and output index to the output.
- The ubalance function calculates sum total of all the coin in a given UTxO.
- The wbalance function calculates sum total of all the withdrawals in a transaction.
- The calculation consumed gives the value consumed by the transaction *tx* in the context of the protocol parameters, the current UTxO on the ledger, and the registered stake keys. This calculation is a sum of all coin in the inputs of *tx*, reward withdrawals, and stake key deposit refunds. Some of the definitions used in this function will be defined in Section 7.2. In particular, keyRefunds is defined in Figure 9 and StakeKeys is defined in Figure 14.
- The calculation produced gives the value produced by the transaction *tx* in the context of the protocol parameters and the registered stake pools. This calculation is a sum of all coin in the outputs of *tx*, the transaction fee, and all needed deposits. Some of the definitions used in this function will be defined in Section 7.2. In particular, deposits is defined in Figure 9 and StakePools is defined in Figure 14.

The preservation of value property holds for a transaction, for a given ledger state, exactly when the results of consumed equal the results of produced. Moreover, when the property holds, value is only moved between transaction outputs, the reward accounts, the fee pot, and the deposit pot.

Note that the consumed function takes the registered stake pools (*stpools*) as a parameter only in order to determine which pool registration certificates are new (and thus require a deposit) and which ones are updates. Registration will be discussed more in Section 8.

$\mathsf{outs} \in \mathsf{Tx} \to \mathsf{UTxO}$ — tx outputs as UTxO

$\mathsf{outs}\ tx = \{(\mathsf{txid}\ tx, ix) \mapsto txout \mid ix \mapsto txout \in \mathsf{txouts}\ tx\}$

$\mathsf{ubalance} \in \mathsf{UTxO} \to \mathsf{Coin}$ — UTxO balance

$$\mathsf{ubalance}\ utxo = \sum_{(\_ \mapsto (\_,\ c)) \in utxo} c$$

$\mathsf{wbalance} \in \mathsf{Wdrl} \to \mathsf{Coin}$ — withdrawal balance

$$\mathsf{wbalance}\ ws = \sum_{\_ \mapsto c \in ws} c$$

$\mathsf{consumed} \in \mathsf{PParams} \to \mathsf{UTxO} \to \mathsf{StakeKeys} \to \mathsf{Wdrl} \to \mathsf{Tx} \to \mathsf{Coin}$ — value consumed

$$\begin{aligned}&\mathsf{consumed}\ pp\ utxo\ stkeys\ rewards\ tx = \\ &\quad \mathsf{ubalance}\ (\mathsf{txins}\ tx \lhd utxo) + \mathsf{wbalance}\ (\mathsf{txwdrls}\ tx) \\ &\qquad + \mathsf{keyRefunds}\ pp\ stkeys\ tx\end{aligned}$$

$\mathsf{produced} \in \mathsf{PParams} \to \mathsf{StakePools} \to \mathsf{Tx} \to \mathsf{Coin}$ — value produced

$$\begin{aligned}&\mathsf{produced}\ pp\ stpools\ tx = \\ &\quad \mathsf{ubalance}\ (\mathsf{outs}\ tx) + \mathsf{txfee}\ tx + \mathsf{deposits}\ pp\ stpools\ (\mathsf{dcerts}\ tx)\end{aligned}$$

Figure 6: Functions used in UTxO rules

The types for the UTxO transition are given in Figure 7. The environment, UTxOEnv, consists of:

- The current slot.
- The protocol parameters.
- The registered stake keys (which will be explained in Section 8, Figure 14).
- The registered stake pools (also explained in Section 8, Figure 14).

The current slot and the registrations are need for the refund calculations described in Section 7.2.

The state needed for the UTxO transition, UTxOState, consists of:

- The current UTxO.
- The deposit pot.
- The fee pot.

The signal for the UTxO transition is a transaction.

UTxO environment

$$\mathsf{UTxOEnv} = \begin{pmatrix} \mathit{slot} & \in & \mathsf{Slot} & \text{current slot} \\ \mathit{pp} & \in & \mathsf{PParams} & \text{protocol parameters} \\ \mathit{stkeys} & \in & \mathsf{StakeKeys} & \text{stake key} \\ \mathit{stpools} & \in & \mathsf{StakePools} & \text{stake pool} \end{pmatrix}$$

UTxO States

$$\mathsf{UTxOState} = \begin{pmatrix} \mathit{utxo} & \in & \mathsf{UTxO} & \mathsf{UTxO} \\ \mathit{deposits} & \in & \mathsf{Coin} & \text{deposits pot} \\ \mathit{fees} & \in & \mathsf{Coin} & \text{fee pot} \end{pmatrix}$$

UTxO transitions

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

Figure 7: UTxO transition-system types

The UTxO transition system is given in Figure 8. Rule 1 specifies the conditions under which a transaction can be applied to a particular UTxOState in environment UTxOEnv:

The transition contains the following predicates:

- The transaction is live (the current slot is less than its time to live).
- The transaction has at least one input. The global uniqueness of transaction inputs prevents replay attacks. By requiring that all transactions spend at least one input, the entire transaction is safe from such attacks. A delegation certificate by itself, for example, does not have this property.
- The fee paid by the transaction has to be greater than or equal to the minimum fee, which is based on the size of the transaction. A user or wallet might choose to create a fee larger than necessary in exchange for a faster processing time.
- Each input spent in the transaction must be in the set of unspent outputs.

- The *preservation of value* property must hold. In other words, the amount of value produced by the transaction must be the same as the amount consumed.
- The coin value of each new output must be non-negative.

If all the predicates are satisfied, the state is updated as follows:

- Update the UTxO:
  - Remove from the UTxO all the $(txin, txout)$ pairs associated with the *txins*'s in the *inputs* list of *tx*.
  - Add all the *outputs* of *tx* to the UTxO, associated with the txid *tx*
- Add all new deposits to the deposit pot and subtract all refunds.
- Add the transaction fee to the fee pot. Additionally, for any refund returned by this transaction, add the amount of the original deposit which has decayed to the fee pot. The amount decayed will depend on the time to live of the transaction and will be explained further in Section 7.2.

The accounting for the reward withdrawals is not done in this transition system. The rewards are tracked with the delegation state and will be removed in the final delegation transition, see 11.

Note here that output entries for both the deposit refunds and the rewards withdrawals must be included in the body of the transaction carrying the deregistration certificates (requesting these refunds) and the reward requests. It is the job of the wallet to calculate the value of these refunds and withdrawals, and generate the correct outputs to include in the outputs list of *tx* such that applying this transaction results in a valid ledger update adding correct amounts of coin to the right addresses.

The approach of including refunds and rewards directly in the $outputs$ gives great flexibility to the management of the coin value obtained from these accounts, i.e. it can be directed to any address. However, it means there is no direct link between the *wdrls* requests (similarly, the key deregistration certificate addresses and refund amounts) and the $outputs$. We verify that the included outputs are correct and authorized through the preservation of value condition and witnessing the transaction. The combination of the preservation of value and witnessing, described in Section 7.3, assures that the ledger state is updated correctly.

The main difference, however, in how rewards and refunds work is that refunds come from a *deposits* pool, which is a single coin value indicating the total decayed amount of all the deposits ever made, while rewards come from individual accounts where a reward is accumulated to a specific address.

Note that the *refunded* and *decayed* values added together give what the full, non-decayed refund for all the key deregistration certificates in *tx* would be, and this total value is always removed from the *deposits* amount on the ledger. The *refunded* amount is returned to the certificate author, and the *decayed* amount is transferred over to *fees* (this allows the ledger to adhere to the preservation of value).

Note also that the reason only the decayed value of requested refunds from *this epoch* is transferred to fees is that at the epoch boundary, the total decayed value for the whole epoch for both the individual and pool deposits is transferred into the fees (independent of refund requests).

$$\frac{\begin{array}{c}\mathsf{txttl}\ tx \geq slot \qquad \mathsf{txins}\ tx \neq \emptyset \qquad \mathsf{minfee}\ pp\ tx \leq \mathsf{txfee}\ tx \qquad \mathsf{txins}\ tx \subseteq \mathrm{dom}\ utxo \\ \mathsf{consumed}\ pp\ utxo\ stkeys\ rewards\ tx = \mathsf{produced}\ pp\ stpools\ tx \\ \\ \forall (\_ \mapsto (\_, c)) \in \mathsf{txouts}\ tx, c \geq 0 \\ \\ \begin{array}{rl} refunded &= \mathsf{keyRefunds}\ pp\ stkeys\ \ tx \\ decayed &= \mathsf{decayedTx}\ pp\ stkeys\ \ tx \\ depositChange &= (\mathsf{deposits}\ pp\ \ stpools\ \mathsf{dcerts}\ tx) - (refunded + decayed) \end{array}\end{array}}{\begin{array}{l} slot \\ pp \\ stkeys \\ stpools \end{array} \vdash \begin{pmatrix} utxo \\ deposits \\ fees \end{pmatrix} \xrightarrow[\text{UTXO}]{tx} \begin{pmatrix} (\mathbf{txins}\ tx \lhd utxo) \cup \mathbf{outs}\ tx \\ deposits + depositChange \\ fees + \mathbf{txfee}\ tx + decayed \end{pmatrix}} \text{UTxO-inductive} \tag{1}$$

Figure 8: UTxO inference rules

7.2 Deposits and Refunds

Deposits are described in appendix B.2 of the delegation design document Kant et al. (2018). These deposit functions were used above in the UTxO transition, 7.1. Deposits are used for stake key registration certificate and pool registration certificates, which will be explained in Section 8. In particular, the function cwitness , which gets the certificate witness from a certificate, will be defined later. Figure 9 defines the deposit and refund functions.

- The function deposits returns the total deposits that have to be made by a transaction. This calculation is based on the protocol parameters. Specifically, for a given transaction, it sums up the values of the stake key deposits and the stake pool deposits. Those certificates which are updates of stake pool parameters of already registered pool keys should not (and are, in fact, not allowed to) make a deposit.
- The function refund calculates the deposit refund with an exponential decay.
- The function keyRefund, calculates the refund for an individual stake key registration deposit, based on the slot when it was created and the slot passed to the function. The creation slot should always exist in the map *stkeys* passed to the function, and this would be a good property to prove about the transition system.
- The function keyRefunds, in turn, uses keyRefund to calculate the total value to be refunded to all individual key deregistration certificate authors in a transaction.

  It is important to note here that instead of the *current* slot number, the time to live of *tx* is passed to the certRefunds function within the summation in keyRefunds. The reason for this is that the refunds for any key deregistration certificates are, in fact, included in the *tx* itself — meaning that the coin value of the refund must be explicitly specified in the outputs of the transaction. So, the value of the included refund must be calculated before this transaction is ever processed, and be the same *no matter when* the *tx is actually processed* in order to allow the system to continue to satisfy the general accounting property.

  It is impossible to predict the exact slot number in which *tx* will be processed, but it will be some time before slot number txttl *tx*. So, this is the slot number value used in both the calculation to generate the refund coin value in the outputs of *tx* and in the general accounting property equation.

  Note also that keyRefunds calculates the total individual refunds for a transaction based on *current* protocol parameters. This means that any deposits made prior to a change will be different from their corresponding (decayed) refunds in the case of key deregistration after a change in protocol parameters. Constants may only change at the epoch boundary, and ensuring there are always sufficient funds for all refunds in the *deposits* pool is part of the protocol constant change transition, described in Section 10.

  The protocol parameters are not expected to change often, and using the current ones for the calculation is a deliberate simplification choice, which does not introduce any inconsistencies into the system rules or properties. In particular, the general accounting property is not violated.

Figure 10 defines the decays functions.

- The function decayedKey calculates how much of a stake key deposit has decayed since the last epoch. Again, this is done using the time to live of the transaction (and not the current slot, as explained above). At the epoch boundaries, decayed portions of deposits are moved to the reward pot, so between epochs we need only account for what has decayed since the last epoch. The value is calculated by subtracting the refund calculation based at the epoch boundary from the refund calculation based at the time to live of the transaction.

- The function decayedTx calculates the total decayed deposits associated with all the refunds in a given transaction. This function was used earlier in the UTxO transition in Figure 8.

Recall that the stake pool retirement refunds are issued not when a certificate scheduling the retirement is processed, but at the epoch boundary for which the retirement is scheduled. The decayed value over the full previous epoch is also accounted for at the boundary change. For details of this accounting, see Section Section 10.

$$\mathsf{deposits} \in \mathsf{PParams} \to \mathsf{StakePools} \to \mathsf{DCert}^* \to \mathsf{Coin} \qquad \text{total deposits for transaction}$$

$$\mathsf{deposits}\ pp\ stpools\ certs = \sum_{c \in certs \cap \mathsf{DCert}_{\mathsf{regkey}}} (\mathsf{keyDeposit}\ pp) + \sum_{\substack{c \in certs \cap \mathsf{DCert}_{\mathsf{regpool}} \\ (\mathsf{cwitness}\ c) \notin stpools}} (\mathsf{poolDeposit}\ pp)$$

$$\mathsf{refund} \in \mathsf{Coin} \to [0,\ 1] \to (0,\ \infty) \to \mathsf{Duration} \to \mathsf{Coin} \qquad \text{refund calculation}$$

$$\mathsf{refund}\ d_{\mathsf{val}}\ d_{\mathsf{min}}\ \lambda\ \delta = \left\lfloor d_{\mathsf{val}} \cdot \left(d_{\mathsf{min}} + (1 - d_{\mathsf{min}}) \cdot e^{-\lambda \cdot \delta}\right)\right\rfloor$$

$$\mathsf{keyRefund} \in \mathsf{Coin} \to [0,\ 1] \to (0,\ \infty) \to \mathsf{StakeKeys} \to \mathsf{Slot} \to \mathsf{DCert}_{\mathsf{deregkey}} \to \mathsf{Coin} \qquad \text{key refund for a certificate}$$

$$\mathsf{keyRefund}\ d_{\mathsf{val}}\ d_{\mathsf{min}}\ \lambda\ stkeys\ slot\ c = \begin{cases} 0 & \text{if } \mathsf{cwitness}\ c \notin \mathsf{dom}\ stkeys \\ \mathsf{refund}\ d_{\mathsf{val}}\ d_{\mathsf{min}}\ \lambda\ \delta & \text{otherwise} \end{cases}$$
$$\textbf{where} \quad \delta = slot\ -_s\ (stkeys\ (\mathsf{cwitness}\ c))$$

$$\mathsf{keyRefunds} \in \mathsf{PParams} \to \mathsf{StakeKeys} \to \mathsf{Tx} \to \mathsf{Coin} \qquad \text{key refunds for a transaction}$$

$$\mathsf{keyRefunds}\ pp\ stkeys\ tx = \sum_{\substack{c \in \mathsf{dcerts}\ tx \\ c \in \mathsf{DCert}_{\mathsf{deregkey}}}} \mathsf{keyRefund}\ d_{\mathsf{val}}\ d_{\mathsf{min}}\ \lambda\ stkeys\ (\mathsf{txttl}\ tx)\ c$$
$$\textbf{where}$$
$$d_{\mathsf{val}} = \mathsf{keyDeposit}\ pp$$
$$d_{\mathsf{min}} = \mathsf{keyMinRefund}\ pp$$
$$\lambda = \mathsf{keyDecayRate}\ pp$$

Figure 9: Functions used in Deposits - Refunds

$$\mathsf{decayedKey} \in \mathsf{PParams} \to \mathsf{StakeKeys} \to \mathsf{Slot} \to \mathsf{DCert}_{\mathsf{deregkey}} \to \mathsf{Coin} \qquad \text{decayed since epoch}$$

$$\mathsf{decayedKey}\ pp\ stkeys\ cslot\ c =$$

$$\begin{cases} 0 & \text{if cwitness } c \notin \text{dom } stkeys \\ epochRefund - currentRefund & \text{otherwise} \end{cases}$$

$$\begin{aligned} \textbf{where} \qquad created &= stkeys\ (\mathsf{cwitness}\ c) \\ start &= \mathsf{max}\ (\mathsf{firstSlot\ epoch}\ cslot)\ created \\ epochRefund &= \mathsf{keyRefund}\ d_{\mathsf{val}}\ d_{\min}\ \lambda\ stkeys\ start\ c \\ currentRefund &= \mathsf{keyRefund}\ d_{\mathsf{val}}\ d_{\min}\ \lambda\ stkeys\ cslot\ c \\ d_{\mathsf{val}} &= \mathsf{keyDeposit}\ pp \\ d_{\min} &= \mathsf{keyMinRefund}\ pp \\ \lambda &= \mathsf{keyDecayRate}\ pp \end{aligned}$$

$$\mathsf{decayedTx} \in \mathsf{PParams} \to \mathsf{StakeKeys} \to \mathsf{Tx} \to \mathsf{Coin} \qquad \text{decayed deposit portions}$$

$$\mathsf{decayedTx}\ pp\ stkeys\ tx =$$

$$\sum_{\substack{c \in \mathsf{dcerts}\ tx \\ c \in \mathsf{DCert}_{\mathsf{deregkey}}}} \mathsf{decayedKey}\ pp\ stkeys\ (\mathsf{txttl}\ tx)\ c$$

Figure 10: Functions used in Deposits - Decay

7.3 Witnesses

The purpose of witnessing is make sure that the intended action is authorized by the holder of the signing key, providing replay protection as a consequence. Replay prevention is an inherent property of UTxO type accounting since transaction IDs are unique, and we require all transaction to consume at least one input.

A transaction is witnessed by a signature and a verification key corresponding to this signature. The witnesses, together with the transaction body, form a full transaction. Every witness in a transaction signs the transaction body. Moreover, the witnesses are represented as finite maps from verification keys to signatures, so that any key that is required to sign a transaction only provides a single witness. This means that, for example, transaction which includes a delegation certificate and a reward withdrawal corresponding to the same stake key still only includes one signature.

Figure 11 defines the function which gathers all the (hashes of) verification keys needed to witness a given transaction. This consists of:

- payment keys for outputs being spent
- stake keys for reward withdrawals
- stake keys for delegation certificates (all five types)
- stake keys for the pool owners in a pool registration certificate

$$\mathsf{witnessNeeded} \in \mathsf{UTxO} \to \mathsf{Tx} \to \mathbb{P}\ \mathsf{HashKey} \qquad \text{hashkeys for needed witnesses}$$

$$\begin{aligned}
&\mathsf{witnessNeeded}\ utxo\ tx = \\
&\quad \{\mathsf{paymentHK}\ a \mid i \mapsto (a, _) \in utxo,\ i \in \mathsf{txins}\ tx\} \cup \\
&\quad \{\mathsf{stakeHK_r}\ a \mid a \mapsto _ \in \mathsf{txwdrls}\ tx\} \cup \\
&\quad \{\mathsf{cwitness}\ c \mid c \in \mathsf{txcerts}\ tx\} \cup \\
&\quad \bigcup_{\substack{c \in \mathsf{txcerts}\ tx \\ c \in \mathsf{DCert_{regpool}}}} \mathsf{poolOwners}\ c
\end{aligned}$$

Figure 11: Functions used in witness rule

The UTxOW transition system adds witnessing to the previous UTxO transition system. Figure 12 defines the type for this transition.

Figure 13 defines UTxOW transition. It has two predicates:

- Every signature in the transaction is a valid signature of the transaction body.
- The set of (hashes of) verification keys given by the transaction is exactly the set of needed (hashes of) verification keys.

If the predicates are satisfied, the state is transitioned by the UTxO transition rule.

UTxO with witness transitions

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

Figure 12: UTxO with witness transition-system types

$$\text{UTxO-wit}\ \frac{\begin{array}{c}(utxo, \_, \_) = utxoSt \\ \\ \forall vk \mapsto \sigma \in \mathsf{txwits}\ tx, \mathcal{V}_{vk}[\![\mathsf{txbody}\ tx]\!]_{\sigma} \\ \mathsf{witsNeeded}\ utxo\ tx = \{\mathsf{hashKey}\ vk \mid vk \in \mathrm{dom}\,(\mathsf{txwits}\ tx)\} \\ utxoEnv \ \vdash utxoSt \xrightarrow[\text{UTXO}]{tx} utxoSt'\end{array}}{utxoEnv \ \vdash utxoSt \xrightarrow[\text{UTXOW}]{tx} \boldsymbol{utxoSt'}} \tag{2}$$

Figure 13: UTxO with witnesses inference rules

8 Delegation

We briefly describe the motivation and context for delegation. The full context is contained in Kant et al. (2018).

Stake is said to be *active* in the blockchain protocol when it is eligible for participation in the leader election. In order for stake to become active, the associated verification stake key must be registered and its staking rights must be delegated to an active stake pool. Individuals who wish to participate in the protocol can register themselves as a stake pool.

Stake keys are registered (deregistered) through the use of registration (deregistration) certificates. Registered stake keys are delegated through the use of delegation certificates. Finally, stake pools are registered (retired) through the use of registration (retirement) certificates.

Stake pool retirement is handled a bit differently than stake key deregistration. Stake keys are considered inactive as soon as a deregistration certificate is applied to the ledger state. Stake pool retirement certificates, however, specify the epoch in which it will retire.

Delegation requires the following to be tracked by the ledger state: the registered stake keys, the delegation map from registered stake keys to stake pools, pointers associated with stake keys, the registered stake pools, and upcoming stake pool retirements. Additionally, the blockchain protocol rewards eligible stake, and so we must also include a mapping from active stake keys to rewards.

8.1 Delegation Definitions

In Figure 14 we give the delegation primitives. Here we introduce the following primitive datatypes used in delegation:

- $\mathsf{DCert}_{\mathsf{regkey}}$: a stake key registration certificate.
- $\mathsf{DCert}_{\mathsf{deregkey}}$: a stake key de-registration certificate.
- $\mathsf{DCert}_{\mathsf{delegate}}$: a stake key delegation certificate.
- $\mathsf{DCert}_{\mathsf{regpool}}$: a stake pool registration certificate.
- $\mathsf{DCert}_{\mathsf{retirepool}}$: a stake key retirement certificate.
- DCert: any one of of the five certificate types above.

The following derived types are introduced:

- StakeKeys represents registered stake keys, and is represented by a finite map from hashkeys to slot when it was registered.
- StakePools represents registered stake pools, and has the same type as StakeKeys.
- PoolParam represents the parameters found in a stake pool registration certificate that must be tracked:
  - the pool owners.
  - the pool cost.
  - the pool margin.
  - the pool pledge.
  - the pool reward account.

The idea of pool owners is explained in Section 4.4.4 of Kant et al. (2018). The pool cost and margin indicate how much more of the rewards pool leaders get than the members. The pool pledge is explained in Section 5.1 of Kant et al. (2018). The pool reward account is where all pool rewards go.

Accessor functions for certificates and pool parameters are also defined, but only the cwitness accessor function needs explanation. It does the following:

- For a $\mathsf{DCert}_{\mathsf{regkey}}$ certificate, cwitness returns the hashkey of the key being registered.
- For a $\mathsf{DCert}_{\mathsf{deregkey}}$ certificate, cwitness returns the hashkey of the key being de-registered.
- For a $\mathsf{DCert}_{\mathsf{delegate}}$ certificate, cwitness returns the hashkey of the key that is delegating (and not the key to which the stake in being delegated to).
- For a $\mathsf{DCert}_{\mathsf{regpool}}$ certificate, cwitness returns the hashkey of the key of the pool operator.
- For a $\mathsf{DCert}_{\mathsf{retirepool}}$ certificate, cwitness returns the hashkey of the key of the pool operator.

Abstract types

Delegation Certificate types

$$\begin{aligned}\mathsf{DCert} = \mathsf{DCert}_{\mathsf{regkey}} \uplus \mathsf{DCert}_{\mathsf{deregkey}} \uplus \mathsf{DCert}_{\mathsf{delegate}} \\ \uplus \mathsf{DCert}_{\mathsf{regpool}} \uplus \mathsf{DCert}_{\mathsf{retirepool}}\end{aligned}$$

Derived types

$\mathsf{StakeKeys}$	$=$	$\mathsf{HashKey} \mapsto \mathsf{Slot}$	registered stake keys
$\mathsf{StakePools}$	$=$	$\mathsf{HashKey} \mapsto \mathsf{Slot}$	registered stake pools
$\mathsf{PoolParam}$	$=$	$\mathbb{P}\ \mathsf{HashKey} \times \mathsf{Coin} \times [0,\ 1] \times \mathsf{Coin} \times \mathsf{Addr}_{\mathsf{rwd}}$	stake pool parameters

Certificate Accessor functions

$\mathsf{cwitness}$	$\in$	$\mathsf{DCert} \to \mathsf{HashKey}$	certificate witness
dpool	$\in$	$\mathsf{DCert}_{\mathsf{delegate}} \to \mathsf{HashKey}$	pool being delegated to
$\mathsf{poolParam}$	$\in$	$\mathsf{DCert}_{\mathsf{regpool}} \to \mathsf{PoolParam}$	stake pool
retire	$\in$	$\mathsf{DCert}_{\mathsf{retirepool}} \to \mathsf{Epoch}$	epoch of pool retirement

Pool Parameter Accessor functions

$\mathsf{poolOwners}$	$\in$	$\mathsf{PoolParam} \to \mathbb{P}\ \mathsf{HashKey}$	stake pool owners
$\mathsf{poolCost}$	$\in$	$\mathsf{PoolParam} \to \mathsf{Coin}$	stake pool cost
$\mathsf{poolMargin}$	$\in$	$\mathsf{PoolParam} \to [0,\ 1]$	stake pool margin
$\mathsf{poolPledge}$	$\in$	$\mathsf{PoolParam} \to \mathsf{Coin}$	stake pool pledge
$\mathsf{poolRAcnt}$	$\in$	$\mathsf{PoolParam} \to \mathsf{Addr}_{\mathsf{rwd}}$	stake pool reward account

Figure 14: Delegation Definitions

8.2 Delegation Transitions

In Figure 15 we give the delegation and stake pool state transition types. We define two separate parts of the ledger state.

- DState keeps track of the delegation state, consisting of:
  - *stkeys* tracks the registered stake keys. It consists of a finite mapping from hashkeys to the slot of the registration.
  - *rewards* stores the rewards accumulated by stake keys. These are represented by a finite map from reward addresses to the accumulated rewards.
  - *delegations* stores the delegation relation, mapping stake keys to the pool to which is delegates.
  - *ptrs* maps stake keys to the position of the registration certificate in the blockchain. This is needed to lookup the stake hashkey of a pointer address.
- PState keeps track of the stake pool information:
  - *stpools* tracks the registered stake pools. It consists of a finite mapping from hashkeys to the slot of the registration.
  - *poolParams* tracks the parameters associated with each stake pool, such as their costs and margin.
  - *retiring* tracks stake pool retirements, using a map from hashkeys to the epoch in which it will retire.
  - *avgs* stores the latest value of the pool's performance moving average. This value quantifies the desirability of delegating to a given pool, based on past performance.

The environment for the state transition for DState contains the current slot number and the index for the current certificate pointer. The environment for the state transition for PState contains the current slot number and the protocol parameters.

Delegation States

$$\mathsf{DState} = \left(\begin{array}{rcll} \mathit{stkeys} & \in & \mathsf{StakeKeys} & \text{registered stake keys} \\ \mathit{rewards} & \in & \mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin} & \text{rewards} \\ \mathit{delegations} & \in & \mathsf{HashKey}_{\mathit{stake}} \mapsto \mathsf{HashKey}_{\mathit{pool}} & \text{delegations} \\ \mathit{ptrs} & \in & \mathsf{Ptr} \mapsto \mathsf{HashKey} & \text{pointer to hashkey} \end{array} \right)$$

$$\mathsf{PState} = \left(\begin{array}{rcll} \mathit{stpools} & \in & \mathsf{StakePools} & \text{registered pools to creation time} \\ \mathit{poolParams} & \in & \mathsf{HashKey}_{\mathit{pool}} \mapsto \mathsf{PoolParam} & \text{registered pools to pool parameters} \\ \mathit{retiring} & \in & \mathsf{HashKey}_{\mathit{pool}} \mapsto \mathsf{Epoch} & \text{retiring stake pools} \\ \mathit{avgs} & \in & \mathsf{HashKey}_{\mathit{pool}} \mapsto \mathbb{R}^{\geq 0} & \text{performance moving average} \end{array} \right)$$

Delegation Environment

$$\mathsf{DEnv} = \left(\begin{array}{rcll} \mathit{slot} & \in & \mathsf{Slot} & \text{slot} \\ \mathit{ptr} & \in & \mathsf{Ptr} & \text{certificate pointer} \end{array} \right)$$

Pool Environment

$$\mathsf{PEnv} = \left(\begin{array}{rcll} \mathit{slot} & \in & \mathsf{Slot} & \text{slot} \\ \mathit{pp} & \in & \mathsf{PParams} & \text{protocol parameters} \end{array} \right)$$

Delegation Transitions

$$\_ \vdash \_ \xrightarrow[\text{DELEG}]{} \_ \in \mathbb{P}\ (\mathsf{DEnv} \times \mathsf{DState} \times \mathsf{DCert} \times \mathsf{DState})$$

$$\_ \vdash \_ \xrightarrow[\text{POOL}]{} \_ \in \mathbb{P}\ (\mathsf{PEnv} \times \mathsf{PState} \times \mathsf{DCert} \times \mathsf{PState})$$

Figure 15: Delegation Transitions

8.3 Delegation Rules

The rules for registering and delegating stake keys are given in Figure 16. Note that section 5.2 of Kant et al. (2018) describes how a wallet would help a user choose a stake pool, though these concerns are independent of the ledger rules.

- Stake key registration is handled by Equation (3), since it contains the precondition that the certificate has type $\mathsf{DCert}_{\mathsf{regkey}}$. All the equations in DELEG and POOL follow this same pattern of matching on certificate type.

  There is also a precondition on registration that the hashkey associated with the certificate witness of the certificate is not already found in the current list of stake keys.

  Registration causes the following state transformation:

  - The key is added to the set of registered stake keys.
  - A reward account is created for this key, with a starting balance of zero.
  - The certificate pointer is mapped to the new stake key.

- Stake key deregistration is handled by Equation (4). There is a precondition that the key has been registered, and that the reward balance is zero. Deregistration causes the following state transformation:

  - The key is removed from the collection of registered keys.
  - The reward account is removed.
  - The key is removed from the delegation relation.
  - The certificate pointer is removed.

- Stake key delegation is handled by Equation (5). There is a precondition that the key has been registered. Delegation causes the following state transformation:

  - The delegation relation is updated so that stake stake key is delegated to the given stake pool. The use of union override here allows us to use the same rule to perform both an initial delegation and an update to an existing delegation.

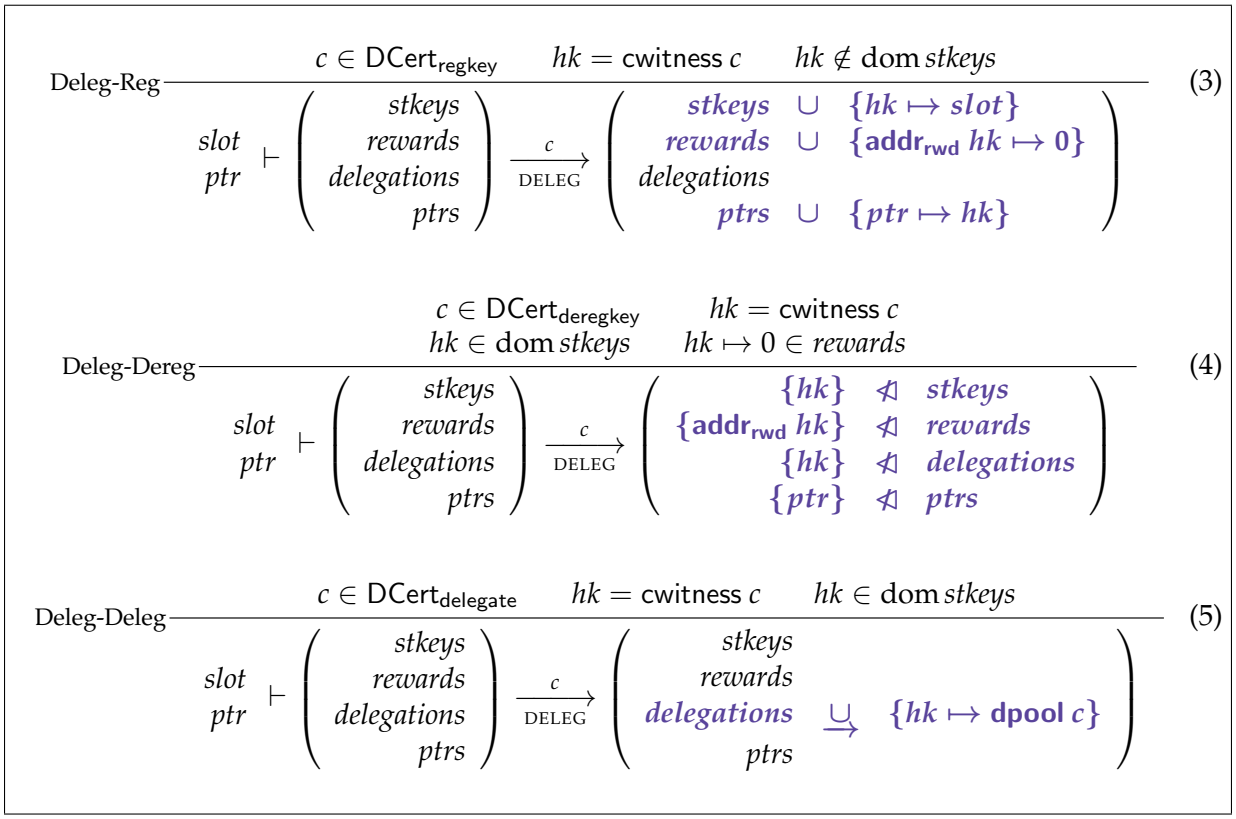


Figure 16: Delegation Inference Rules

8.4 Stake Pool Rules

The rules for updating the part of the ledger state defining the current stake pools are given in Figure 17. The calculation of stake distribution is described in Section 10.3.

In the pool rules, the stake pool is identified with the hashkey of the pool operator. For each rule, again, we first check that a given certificate c is of the correct type.

- Stake pool registration is handled by Equation (6). It is required that the pool not be currently registered. Registration causes the following state transformation:
  - The key is added to the set of registered stake pools.
  - The pool's parameters are stored.
- Stake pool parameter updates are handled by Equation (6). This rule, which also matches on the certificate type DCertRegPool, is distinguished from Equation (6) by the requirement that the pool be registered. This rule also ends stake pool retirements. Reregistration causes the following state transformation:
  - The pool's parameters are updated.
  - The pool is removed from the collection of retiring pools.
  - Note that *stpools* is **not** updated. The registration creation slot does does not change.
- Stake pool retirements are handled by Equation (8). Given a slot number *slot*, the application of this rule requires that the planned retirement epoch e stated in the certificate is in the future, i.e. after *cepoch*, the epoch of the current slot number in this context, as well as that it is less than $\mathsf{E}_{\mathsf{max}}$ epochs after the current one. It is also required that the pool be registered. Note that imposing the $\mathsf{E}_{\mathsf{max}}$ constraint on the system is not strictly necessary. However, forcing stake pools to announce their retirement a shorter time in advance will curb the growth of the *retiring* list in the ledger state.

  The pools scheduled for retirement must be removed from the *retiring* state variable at the end of the epoch they are scheduled to retire in. This non-signaled transition (triggered, instead, directly by a change of current slot number in the environment), along with all other transitions that take place at the epoch boundary, are described in Section 10.

  Reregistration causes the following state transformation:
  - The pool is marked to retire on the given epoch. If it was previously retiring, the retirement epoch is now updated.

$$\text{Pool-Reg}\ \frac{c \in \mathsf{DCert}_{\mathsf{regpool}} \qquad hk = \mathsf{cwitness}\ c \qquad hk \notin \mathrm{dom}\ stpools}{slot,\ pp \vdash \begin{pmatrix} stpools \\ poolParams \\ retiring \\ avgs \end{pmatrix} \xrightarrow[\text{POOL}]{c} \begin{pmatrix} stpools & \cup & \{hk \mapsto slot\} \\ poolParams & \cup & \{hk \mapsto \mathbf{poolParam}\ c\} \\ retiring & & \\ avgs & & \end{pmatrix}} \tag{6}$$

$$\text{Pool-reReg}\ \frac{c \in \mathsf{DCert}_{\mathsf{regpool}} \qquad hk = \mathsf{cwitness}\ c \qquad hk \in \mathrm{dom}\ stpools}{slot,\ pp \vdash \begin{pmatrix} stpools \\ poolParams \\ retiring \\ avgs \end{pmatrix} \xrightarrow[\text{POOL}]{c} \begin{pmatrix} stpools & & \\ poolParams & \underrightarrow{\cup} & \{hk \mapsto \mathbf{poolParam}\ c\} \\ \{hk\} & \lhd & retiring \\ avgs & & \end{pmatrix}} \tag{7}$$

$$\text{Pool-Retire}\ \frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{retirepool}} \qquad hk = \mathsf{cwitness}\ c \qquad hk \in \mathrm{dom}\ stpools \\ e = \mathsf{retire}\ c \qquad cepoch = \mathsf{epoch}\ slot \qquad cepoch < e < cepoch + (\mathsf{emax}\ pp) \end{array}}{slot,\ pp \vdash \begin{pmatrix} stpools \\ poolParams \\ retiring \\ avgs \end{pmatrix} \xrightarrow[\text{POOL}]{c} \begin{pmatrix} stpools & & \\ poolParams & & \\ retiring & \underrightarrow{\cup} & \{hk \mapsto e\} \\ avgs & & \end{pmatrix}} \tag{8}$$

Figure 17: Pool Inference Rule

8.5 Delegation and Pool Combined Rules

We now combine the delegation and pool transition systems. Figure 18 gives the state, environment, and transition type for the combined transition.

Delegation and Pool Combined Environment

$$\mathsf{DPEnv} = \begin{pmatrix} \mathit{slot} & \in & \mathsf{Slot} & \text{slot} \\ \mathit{ptr} & \in & \mathsf{Ptr} & \text{certificate pointer} \\ \mathit{pp} & \in & \mathsf{PParams} & \text{protocol parameters} \end{pmatrix}$$

Delegation and Pool Combined State

$$\mathsf{DPState} = \begin{pmatrix} \mathit{dstate} & \in & \mathsf{DState} & \text{delegation state} \\ \mathit{pstate} & \in & \mathsf{PState} & \text{pool state} \end{pmatrix}$$

Delegation and Pool Combined Transition

$$\_ \vdash \_ \xrightarrow[\text{DELPL}]{} \_ \in \mathbb{P}\left(\mathsf{DPEnv} \times \mathsf{DPState} \times \mathsf{DCert} \times \mathsf{DPState}\right)$$

Figure 18: Delegation and Pool Combined Transition Type

Figure 19, gives the rules for the combined transition. Note that for any given certificate, at most one of the two rules (Equation (9) and Equation (10)) will be successful, since the pool certificates are disjoint from the delegation certificates.

Delegation and Pool Combined Rules

$$\text{Delpl-Del}\ \frac{\begin{matrix} slot \\ ptr \end{matrix} \vdash dstate \xrightarrow[\text{DELEG}]{c} dstate'}{\begin{pmatrix} slot \\ ptr \\ pp \end{pmatrix} \vdash \begin{pmatrix} dstate \\ pstate \end{pmatrix} \xrightarrow[\text{DELPL}]{c} \begin{pmatrix} \mathbf{dstate'} \\ pstate \end{pmatrix}} \tag{9}$$

$$\text{Delpl-Pool}\ \frac{\begin{matrix} slot \\ pp \end{matrix} \vdash pstate \xrightarrow[\text{POOL}]{c} pstate'}{\begin{pmatrix} slot \\ ptr \\ pp \end{pmatrix} \vdash \begin{pmatrix} dstate \\ pstate \end{pmatrix} \xrightarrow[\text{DELPL}]{c} \begin{pmatrix} dstate \\ \mathbf{pstate'} \end{pmatrix}} \tag{10}$$

Figure 19: Delegation and Pool Combined Transition Rules

We now describe a transition system that processes the list of certificates inside a transaction. It is defined recursively from the transition system in Figure 19 above.

Figure 20 defines the types for the delegation certificate sequence transition.

Certificate Sequence Environment

$$\mathsf{DPSEnv} = \begin{pmatrix} slot & \in & \mathsf{Slot} & \text{slot} \\ txIx & \in & \mathsf{Ix} & \text{transaction index} \\ pp & \in & \mathsf{PParams} & \text{protocol parameters} \end{pmatrix}$$

$$\_ \vdash \_ \xrightarrow[\text{DELEGS}]{} \_ \in \mathbb{P}\ (\mathsf{DPSEnv} \times \mathsf{DPState} \times \mathsf{DCert}^* \times \mathsf{DPState})$$

Figure 20: Delegation sequence transition type

Figure 21 defines the transition system recursively. This definition guarantees that a certificate list (and therefore, the transaction carrying it) cannot be processed unless every certificate in it is valid. For example, if a transaction is carrying a certificate that schedules a pool retirement in a past epoch, the whole transaction will be invalid.

- The base case, when the list is empty, is captured by Equation (11). In the base case, we address one final accounting detail not yet covered by the UTxO transition, namely setting the reward account balance to zero for any account that made a withdrawal. There is therefore a precondition that all withdrawals are correct, where correct means that there is a reward account for each stake key, and that the balance matches that of the reward being withdrawn. The base case triggers the following state transformation:
  - Reward accounts are set to zero for each corresponding withdrawal.
- The inductive case, when the list is non-empty, is captured by Equation (12). It constructs a certificate pointer given the current slot and transaction index, calls DELPL on the next certificate in the list, and inductively calls DELEGS on the rest of the list. The inductive case triggers the following state transformation:

– The delegation and pool states are (inductively) updated by the results of DELEGS, which is then updated according to DELPL.

$$\text{Seq-delg-base}\ \frac{\begin{array}{c} wdrls = \mathsf{txwdrls}\ tx \qquad wdrls \subseteq rewards \\ rewards' = rewards \underset{\Rightarrow}{\cup} \{(w, 0) \mid w \in \mathrm{dom}\ wdrls\} \end{array}}{\begin{pmatrix} slot \\ txIx \\ pp \end{pmatrix} \vdash \begin{pmatrix} stkeys \\ rewards \\ delegations \\ ptrs \\ stpools \\ poolParams \\ retiring \\ avgs \end{pmatrix} \xrightarrow[\text{DELEGS}]{\epsilon} \begin{pmatrix} stkeys \\ \boldsymbol{rewards'} \\ delegations \\ ptrs \\ stpools \\ poolParams \\ retiring \\ avgs \end{pmatrix}} \tag{11}$$

$$\text{Seq-delg-ind}\ \frac{\begin{array}{c} c \in \mathsf{DCert}_{\mathsf{delegate}} \Rightarrow \mathsf{dpool}\ c \in \mathrm{dom}\ stpools \\ ptr = (slot,\ txIx,\ \mathsf{len}\ \Gamma - 1) \\ \begin{pmatrix} slot \\ txIx \\ pp \end{pmatrix} \vdash dpstate \xrightarrow[\text{DELEGS}]{\Gamma} dpstate' \\ \begin{pmatrix} slot \\ ptr \\ pp \end{pmatrix} \vdash dpstate' \xrightarrow[\text{DELPL}]{c} dpstate'' \end{array}}{\begin{pmatrix} slot \\ txIx \\ pp \end{pmatrix} \vdash dpstate \xrightarrow[\text{DELEGS}]{\Gamma;c} \boldsymbol{dpstate''}} \tag{12}$$

Figure 21: Delegation sequence rules

9 Ledger State Transition

The entire state transformation of the ledger state caused by a valid transaction can now be given as the combination of the UTxO transition and the delegation transitions.

Figure 22 defines the types for this transition. The environment for this rule is consists of:

- The current slot.
- The transaction index within the current block.
- The protocol parameters.

The ledger state consists of:

- The UTxO state.
- The delegation and pool states.

Ledger environment

$$\mathsf{LEnv} = \left(\begin{array}{rcll} \mathit{slot} & \in & \mathsf{Slot} & \text{current slot} \\ \mathit{txIx} & \in & \mathsf{Ix} & \text{transaction index} \\ \mathit{pp} & \in & \mathsf{PParams} & \text{protocol parameters} \end{array} \right)$$

Ledger state

$$\mathsf{LState} = \left(\begin{array}{rcll} \mathit{utxoSt} & \in & \mathsf{UTxOState} & \text{UTxO state} \\ \mathit{dpstate} & \in & \mathsf{DPState} & \text{delegation and pool state} \end{array} \right)$$

Ledger transitions

$$\_ \vdash \_ \xrightarrow[\text{LEDGER}]{} \_ \subseteq \mathbb{P}\ (\mathsf{LEnv} \times \mathsf{LState} \times \mathsf{Tx} \times \mathsf{LState})$$

Figure 22: Ledger transition-system types

Figure 22 defines the ledger state transition. It has a single rule, which first calls the UTXOW transition, and then calls the DELEGS transition.

$$
\text{ledger}\ \frac{\begin{array}{c}(dstate, pstate) = dpstate \\ (stkeys, \_, \_, \_) = dstate \\ (\_, \_, stpools, \_) = pstate \\ \\ \left(\begin{array}{r} slot \\ pp \\ stkeys \\ stpools \end{array}\right) \vdash utxoSt \xrightarrow[\text{UTXOW}]{tx} utxoSt' \\ \\ \left(\begin{array}{l} slot \\ txIx \\ pp \end{array}\right) \vdash dpstate \xrightarrow[\text{DELEGS}]{tx} dpstate'\end{array}}{\left(\begin{array}{l} slot \\ txIx \\ pp \end{array}\right) \vdash \left(\begin{array}{l} utxoSt \\ dpstate \end{array}\right) \xrightarrow[\text{LEDGER}]{tx} \left(\begin{array}{l} \boldsymbol{utxoSt'} \\ \boldsymbol{dpstate'} \end{array}\right)} \tag{13}
$$

Figure 23: Ledger inference rule

10 Rewards and the Epoch Boundary

This chapter introduces two main transition systems. Neither transition is triggered by a transaction, and in fact have no signal.

The first one, defined in Section 10.7, involves calculations that occur at the epoch boundary. This includes taking stake distribution snapshots (Sections 10.2 and 10.4), retiring stake pools (Section 10.5), and performing protocol updates (Section 10.6). The second transition, defined in Sections 10.8 and 10.9, distributes the leader election rewards.

10.1 Overview of the Reward Calculation

The rewards for a given epoch e_i involve the two epochs surrounding it. In particular, the stake distribution will come from the previous epoch, and the rewards will be calculated in the following epoch. More concretely:

(A) A stake distribution snapshot is taken at the begining of epoch e_{i-1}.

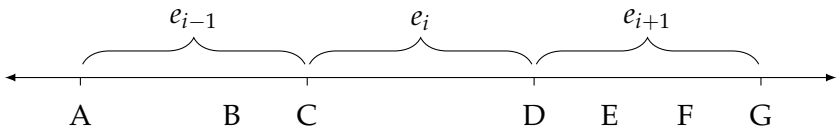
(B) The randomness for leader election is fixed during epoch e_{i-1}

(C) Epoch e_i begins.

(D) Epoch e_i ends. A snapshot is taken of the stake pool performance during epoch e_i. A snapshot is also taken of the fee pot and the decayed deposit values.

(E) The snapshots from (D) are stable and the reward calculation can begin.

(F) Rewards are given out.



We must therefore store the last three stake distributions. The mnemonic "mark, set, go" will be used to keep track of the snapshots, where the label "mark" refers to the most recent snapshot, and "go" refers to the snapshot that is ready to be used in the reward calculation. In the above diagram, the snapshot taken at (A) is labeled "mark" during epoch e_{i-1}, "set" during epoch e_i, and "go" during epoch e_{i+1}. At (G) the snapshot taken at (A) is no longer needed and will be discarded.

The two main transition systems in this section are:

- The transition system named EPOCH, which is defined in Section 10.7, covering what happens at the epoch boundary, such as at (A), (C), (D), and (G).
- The transition named REWARD, which is defined in Section 10.9, covering the reward calculation which happens between (E) and (F).

10.2 Helper Functions

Figure 24 defines four helper functions needed throughout the rest of the section.

- The function obligation calculates the the minimal amount of coin needed to pay out all deposit refunds, as of the current slot.

- The function poolRefunds is used to calculate the total refunds that must be distributed for stake pools scheduled to retire. Note that this calculation takes a slot number corresponding to the epoch boundary slot when the calculation is performed. The returned map maps pool operator hashkeys to the refunds, which will ultimately be returned to the registered reward account.
- The function poolStake filters the stake distribution to one stake pool.
- The function updateAvgs calculates the new performance moving averages.

Total possible refunds

$$\begin{aligned}
&\mathsf{obligation} \in \mathsf{PParams} \to \mathsf{StakeKeys} \to \mathsf{StakePools} \to \mathsf{Slot} \to \mathsf{Coin}\\
&\mathsf{obligation}\ pp\ stkeys\ stpools\ cslot =\\
&\quad \sum_{(_ \mapsto s) \in stkeys} \mathsf{refund}\ d_{\mathsf{val}}\ d_{\min}\ \lambda_d\ (cslot\ -_s\ s) + \sum_{(_ \mapsto s) \in stpools} \mathsf{refund}\ p_{\mathsf{val}}\ p_{\min}\ \lambda_p\ (cslot\ -_s\ s)\\
&\quad \textbf{where} \quad d_{\mathsf{val}},\ d_{\min},\ \lambda_d = \mathsf{keyDeposit}\ pp,\ \mathsf{keyMinRefund}\ pp,\ \mathsf{keyDecayRate}\ pp\\
&\qquad\qquad\ p_{\mathsf{val}},\ p_{\min},\ \lambda_p = \mathsf{poolDeposit}\ pp,\ \mathsf{poolMinRefund}\ pp,\ \mathsf{poolDecayRate}\ pp
\end{aligned}$$

Pool refunds

$$\begin{aligned}
&\mathsf{poolRefunds} \in \mathsf{PParams} \to (\mathsf{HashKey}_{pool} \mapsto \mathsf{Epoch}) \to \mathsf{Slot} \to (\mathsf{HashKey}_{pool} \mapsto \mathsf{Coin})\\
&\mathsf{poolRefunds}\ pp\ retiring\ cslot = \{hk \mapsto \mathsf{refund}\ p_{\mathsf{val}}\ p_{\min}\ \lambda\ (cslot\ -_s\ (\mathsf{slot}\ e)) \mid hk \mapsto e \in retiring\}\\
&\quad \textbf{where}\ \ p_{\mathsf{val}},\ p_{\min},\ \lambda_p = \mathsf{poolDeposit}\ pp,\ \mathsf{poolMinRefund}\ pp,\ \mathsf{poolDecayRate}\ pp
\end{aligned}$$

Filter Stake to one Pool

$$\begin{aligned}
&\mathsf{poolStake} \in \mathsf{HashKey}_{pool} \to (\mathsf{HashKey}_{stake} \mapsto \mathsf{HashKey}_{pool}) \to \mathsf{Stake} \to \mathsf{Stake}\\
&\mathsf{poolStake}\ hk\ delegs\ stake = \mathrm{dom}\ (delegs \lhd \{hk\}) \rhd stake
\end{aligned}$$

Update Moving Averages

$$\begin{aligned}
&\mathsf{updateAvgs} \in \mathsf{PParams} \to \mathsf{Avgs} \to \mathsf{BlocksMade} \to (\mathsf{HashKey}_{stake} \mapsto \mathsf{HashKey}_{pool}) \to \mathsf{Stake} \to \mathsf{Avgs}\\
&\mathsf{updateAvgs}\ pp\ avgs\ blocks\ delegs\ stake =\\
&\quad \{hk \mapsto \mathsf{movingAvg}\ pp\ hk\ n\ (\mathsf{expected}\ hk)\ avgs \mid hk \mapsto n \in blocks\}\\
&\quad \textbf{where}\\
&\qquad tot = \sum_{_ \mapsto c \in stake} c\\
&\qquad \mathsf{expected}\ hk = \left(\sum_{_ \mapsto c \in (\mathsf{poolStake}\ hk\ delegs\ stake)} c \right) \cdot \mathsf{SlotsPerEpoch} / tot
\end{aligned}$$

Figure 24: Helper Functions used in Rewards and Epoch Boundary

10.3 Stake Distribution Calculation

This section defines the stake distribution calculations. Figure 25 introduces three new derived types:

- BlocksMade represents the number of blocks each stake pool produced during an epoch.
- Stake represents the amount of stake (in Coin) controlled by each stake pool.
- Avgs represents the performance moving averages of the stake pools.

Derived types

$$
\begin{array}{rcll}
\mathit{blocks} \in \mathsf{BlocksMade} & = & \mathsf{HashKey}_{\mathit{pool}} \mapsto \mathbb{N} & \text{blocks made by stake pools} \\
\mathit{stake} \in \mathsf{Stake} & = & \mathsf{HashKey}_{\mathit{stake}} \mapsto \mathsf{Coin} & \text{stake} \\
\mathit{avgs} \in \mathsf{Avgs} & = & \mathsf{HashKey}_{\mathit{pool}} \mapsto \mathbb{R}^{\geq 0} & \text{performance moving averages}
\end{array}
$$

Figure 25: Epoch definitions

The stake distribution calculation is given in Figure 26.

- $\mathsf{aggregate}_+$ takes a relation on $A \times B$, where B is any monoid, and returns a map from each $a \in A$ to the sum of all $b \in B$ such that $(a, b) \in B$.
- stakeDistr uses the $\mathsf{aggregate}_+$ function and several relations to compute the stake distribution, mapping each hashkey to the total coin under its control. Keys that are not both registered and delegated are filtered out. The relation passed to $\mathsf{aggregate}_+$ is made up of:
  - $\mathsf{stakeHK_b}^{-1}$, relating hashkeys to (base) addresses
  - $(\mathsf{addrPtr} \circ \mathit{ptr})^{-1}$, relating hashkeys to (pointer) addresses
  - range utxo, relating addresses to coins
  - $\mathsf{stakeHK_r}^{-1} \circ \mathit{rewards}$, relating (reward) addresses to coins

  The notation for relations is explained in Section 2.

Aggregation (for a monoid B)

$$\mathsf{aggregate}_+ \in \mathbb{P}\ (A \times B) \to (A \mapsto B)$$

$$\mathsf{aggregate}_+\ R = \left\{ a \mapsto \sum_{(a,b)\in R} b \ \mid\ a \in \operatorname{dom} R \right\}$$

Stake Distribution (using functions and maps as relations)

$\mathsf{stakeDistr} \in \mathsf{UTxO} \to \mathsf{DState} \to \mathsf{PState} \to \mathsf{Stake}$

$\mathsf{stakeDistr}\ utxo\ dstate\ pstate = (\operatorname{dom} activeDelegs) \lhd (\mathsf{aggregate}_+\ stakeRelation)$

where

$(stkeys,\ rewards,\ delegations,\ ptrs) = dstate$

$(stpools,\ \_,\ \_,\ \_) = pstate$

$stakeRelation = \left(\left(\mathsf{stakeHK_b}^{-1} \cup (\mathsf{addrPtr} \circ ptr)^{-1}\right) \circ (\operatorname{range} utxo)\right) \cup \left(\mathsf{stakeHK_r}^{-1} \circ rewards\right)$

$activeDelegs = (\operatorname{dom} stkeys) \lhd delegations \rhd (\operatorname{dom} stpools)$

Figure 26: Stake Distribution Function

10.4 Snapshot Transition

The state transition types for stake distribution snapshots are given in Figure 27. The type Snapshots contains the information needing to be saved on the epoch boundary:

- $pstake_{mark}$, $pstake_{set}$, and $pstake_{go}$ are the three stake distribution snapshots (paired with the corresponding delegation map), as explained in Section 10.1.
- *poolsSS* stores the pool parameters from the epoch boundary.
- *blocksSS* stores the performance of the completed epoch.
- *feeSS* stores the fees and decayed deposit amounts at the epoch boundary.

Snapshot environment

$$\mathsf{SnapshotEnv} = \left(\begin{array}{rcll} e_{new} & \in & \mathsf{Epoch} & \text{the upcoming epoch} \\ pp & \in & \mathsf{PParams} & \text{protocol parameters} \\ dstate & \in & \mathsf{DState} & \text{delegation state} \\ pstate & \in & \mathsf{PState} & \text{pool state} \\ blocks & \in & \mathsf{BlocksMade} & \text{blocks made} \end{array} \right)$$

Snapshots

$$\mathsf{Snapshots} = \left(\begin{array}{rcll} pstake_{mark} & \in & \mathsf{Stake} \times (\mathsf{HashKey}_{stake} \mapsto \mathsf{HashKey}_{pool}) & \text{newest stake} \\ pstake_{set} & \in & \mathsf{Stake} \times (\mathsf{HashKey}_{stake} \mapsto \mathsf{HashKey}_{pool}) & \text{middle stake} \\ pstake_{go} & \in & \mathsf{Stake} \times (\mathsf{HashKey}_{stake} \mapsto \mathsf{HashKey}_{pool}) & \text{oldest stake} \\ poolsSS & \in & \mathsf{HashKey} \mapsto \mathsf{PoolParam} & \text{pool parameters} \\ blocksSS & \in & \mathsf{BlocksMade} & \text{blocks made} \\ feeSS & \in & \mathsf{Coin} & \text{fee snapshot} \end{array} \right)$$

Snapshot States

$$\mathsf{SnapshotState} = \left(\begin{array}{rcll} ss & \in & \mathsf{Snapshots} & \text{snapshots} \\ utxoSt & \in & \mathsf{UTxOState} & \text{utxo state} \end{array} \right)$$

Snapshot transitions

$$\_ \vdash \_ \xrightarrow[\text{SNAP}]{} \_ \subseteq \mathbb{P}\left(\mathsf{SnapshotEnv} \times \mathsf{SnapshotState} \times \mathsf{SnapshotState}\right)$$

Figure 27: Snapshot transition-system types

The snapshot transition rule is given in Figure 28. This transition has no preconditions and results in the following state change:

- The oldest snapshot is replaced with the penultimate one.
- The penultimate snapshot is replaced with the newest one.
- The newest snapshot is replaced with one just calculated.
- The pool parameters are stored.
- The pool performance is stored.
- The fees and decayed deposits are stored in *feeSS*. Note that this value will not change between epochs, unlike the *fees* and *deposits* values in the UTxO state.

- In the UTxO state, the decayed deposit amounts are moved from the deposit pot to the fee pool. Note that in the reward transition (Section 10.9), the value *feeSS* will be removed from the fee pot in the UTxO state. The decay is calculated based on *the first slot in the upcoming epoch*.

$$\text{Snapshot}\ \frac{\begin{array}{r}(utxo,\ deposits,\ fees) = utxoSt \\ (stkeys,\ \_,\ delegations,\ \_) = dstate \\ (stpools,\ poolParams,\ \_,\ \_) = pstate \\ stake = \mathsf{stakeDistr}\ utxo\ dstate\ pstate \\ slot = \mathsf{firstSlot}\ e_{new} \\ oblg = \mathsf{obligation}\ pp\ stkeys\ stpools\ slot \\ decayed = deposits - oblg\end{array}}{\begin{array}{l}e_{new} \\ pp \\ dstate \\ pstate \\ blocks\end{array} \vdash \begin{pmatrix} pstake_{mark} \\ pstake_{set} \\ pstake_{go} \\ poolsSS \\ blocksSS \\ feeSS \\ \\ utxo \\ deposits \\ fees \end{pmatrix} \xrightarrow[\text{SNAP}]{} \begin{pmatrix} \boldsymbol{(stake,\ delegations)} \\ \boldsymbol{pstake_{mark}} \\ \boldsymbol{pstake_{set}} \\ \boldsymbol{poolParams} \\ \boldsymbol{blocks} \\ \boldsymbol{fees + decayed} \\ \\ utxo \\ \boldsymbol{oblg} \\ \boldsymbol{fees + decayed} \end{pmatrix}} \tag{14}$$

Figure 28: Snapshot Inference Rule

10.5 Pool Reaping Transition

Figure 29 defines the types for the pool reap transition, which is responsible for removing pools slated for retirement in the given epoch.

Pool Reap environment

$$\mathsf{PlReapEnv} = \left(\begin{array}{rcll} e_{new} & \in & \mathsf{Epoch} & \text{the upcoming epoch} \\ pp & \in & \mathsf{PParams} & \text{protocol parameters} \end{array} \right)$$

Pool Reap State

$$\mathsf{PlReapState} = \left(\begin{array}{rcll} acnt & \in & \mathsf{Acnt} & \text{accounting} \\ dstate & \in & \mathsf{DState} & \text{delegation state} \\ pstate & \in & \mathsf{PState} & \text{pool state} \end{array} \right)$$

Pool Reap transitions

$$\_ \vdash \_ \xrightarrow[\text{POOLREAP}]{} \_ \in \mathbb{P}\left(\mathsf{PlReapEnv} \times \mathsf{PlReapState} \times \mathsf{PlReapState}\right)$$

Figure 29: Pool Reap Transition

The pool-reap transition rule is given in Figure 30. This transition has no preconditions and results in the following state change:

- For each retiring pool, the refund for the pool registration deposit is added to the pool's registered reward account, provided the reward account is still registered.
- The sum of all the refunds attached to unregistered reward accounts are added to the treasury.
- Any delegation to a retiring pool is removed.
- Each retiring pool is removed from all four maps in the pool state.

$$retired = retiring^{-1}\ e_{new}$$
$$pr = \mathsf{poolRefunds}\ pp\ retiring\ (\mathsf{firstSlot}\ e_{new})$$
$$rewardAcnts = \{hk \mapsto \mathsf{poolRAcnt}\ pool \mid hk \mapsto pool \in retired \lhd poolParams\}$$
$$refunds = \left\{ a \mapsto c \;\middle|\; \begin{array}{rcl} hk \mapsto c & \in & pr, \\ hk \mapsto a & \in & rewardAcnts, \\ a & \in & \mathrm{dom}\ rewards \end{array} \right\}$$
$$unclaimed = \sum_{\substack{hk \mapsto c \in pr \\ hk \mapsto a \in rewardAcnts, \\ a \notin \mathrm{dom}\ rewards}} c$$

$$\text{Pool-Reap}\ \frac{}{e_{new} \\ pp \vdash \begin{pmatrix} treasury \\ reserves \\ rewardPot \\ \\ stkeys \\ rewards \\ delegations \\ ptrs \\ \\ stpools \\ poolParams \\ retiring \\ avgs \end{pmatrix} \xrightarrow[\text{POOLREAP}]{} \begin{pmatrix} \mathbf{treasury} & + & \mathbf{unclaimed} \\ reserves & & \\ rewardPot & & \\ & & \\ stkeys & & \\ \mathbf{rewards} & \cup_+ & \mathbf{refunds} \\ \mathbf{delegations} & \rhd & \mathbf{retired} \\ ptrs & & \\ & & \\ \mathbf{retired} & \lhd & \mathbf{stpools} \\ \mathbf{retired} & \lhd & \mathbf{poolParams} \\ \mathbf{retired} & \lhd & \mathbf{retiring} \\ \mathbf{retired} & \lhd & \mathbf{avgs} \end{pmatrix}} \quad (15)$$

Figure 30: Pool Reap Inference Rule

10.6 Protocol Parameter Update Transition

Finally, reaching the epoch boundary may trigger a change in the protocol parameters. The protocol parameters environment consists of the upcoming epoch number, the new protocol parameters, and delegation and pool states. The state change is a change of the UTxOState, the Acnt states, and the current PParams. The type of this state transition is given in Figure 31.

New Proto Param environment

$$\mathsf{NewPParamEnv} = \left(\begin{array}{rcll} e_{new} & \in & \mathsf{Epoch} & \text{upcoming epoch} \\ pp_{new} & \in & \mathsf{PParams} & \text{new protocol parameters} \\ dstate & \in & \mathsf{DState} & \text{delegation state} \\ pstate & \in & \mathsf{PState} & \text{pool state} \end{array} \right)$$

New Proto Param States

$$\mathsf{NewPParamState} = \left(\begin{array}{rcll} utxoSt & \in & \mathsf{UTxOState} & \text{utxo state} \\ acnt & \in & \mathsf{Acnt} & \text{accounting} \\ pp & \in & \mathsf{PParams} & \text{current protocol parameters} \end{array} \right)$$

New Proto Param transitions

$$\_ \vdash \_ \xrightarrow[\text{NEWPP}]{} \_ \subseteq \mathbb{P}\ (\mathsf{NewPParamEnv} \times \mathsf{NewPParamState} \times \mathsf{NewPParamState})$$

Figure 31: New Proto Param transition-system types

Figure 32 defines the new protocol parameter transition. The transition has two rules, depending on whether or not the new protocol parameters would incur a debt of the system that could not be covered by the reserves. The transition has two rules, each with one precondition. The preconditions are negations of each other, so that exactly one will always be met. This transition results in the following state change:

- If the new protocol parameters mean that **fewer** funds are required in the deposit pot to cover all possible refunds, then Rule 16 meets the precondition. The excess is moved to the reserves and the protocol parameters are updated.
- If the new protocol parameters mean that **more** funds are required in the deposit pot to cover all possible refunds, and the difference is **less** than the reserve pot, then Rule 16 meets the precondition. Funds are moved from the reserve pot to cover the difference and the protocol parameters are updated.
- If the new protocol parameters mean that **more** funds are required in the deposit pot to cover all possible refunds, and the difference is **more** than the reserve pot, then Rule 17 meets the precondition and no state changes.

Note that here, unlike most of the inference rules in this document, the $utxoSt'$ and the $acnt'$ do not come from valid UTxO or accounts transitions in the antecedent. We simply define the consequent transition using these directly (instead of listing all the fields in both states in the consequent transition). It is done this way here for ease of reading.

$$
\begin{array}{rcl}
slot & = & \mathsf{firstSlot}\ e_{new} \\
oblg_{cur} & = & \mathsf{obligation}\ pp\ stkeys\ stpools\ slot \\
oblg_{new} & = & \mathsf{obligation}\ pp_{new}\ stkeys\ stpools\ slot \\
diff & = & oblg_{cur} - oblg_{new}
\end{array}
$$

$$
reserves + diff \geq 0
$$

$$
utxoSt' = \begin{pmatrix} utxo \\ \boldsymbol{oblg_{new}} \\ fees \end{pmatrix} \qquad acnt' = \begin{pmatrix} treasury \\ \boldsymbol{reserves + diff} \\ rewardPot \\ rewards \end{pmatrix}
$$

$$
\text{New-Proto-Param-Accepted} \frac{}{\begin{matrix} e_{new} \\ pp_{new} \\ dstate \\ pstate \end{matrix} \vdash \begin{pmatrix} utxoSt \\ acnt \\ pp \end{pmatrix} \xrightarrow[\text{NEWPP}]{} \begin{pmatrix} \boldsymbol{utxoSt'} \\ \boldsymbol{acnt'} \\ \boldsymbol{pp_{new}} \end{pmatrix}} \quad (16)
$$

$$
\begin{array}{rcl}
slot & = & \mathsf{firstSlot}\ e_{new} \\
oblg_{cur} & = & \mathsf{obligation}\ pp\ stkeys\ stpools\ slot \\
oblg_{new} & = & \mathsf{obligation}\ pp_{new}\ stkeys\ stpools\ slot \\
diff & = & oblg_{cur} - oblg_{new}
\end{array}
$$

$$
\text{New-Proto-Param-Denied} \frac{reserves + diff < 0}{\begin{matrix} e_{new} \\ pp_{new} \\ dstate \\ pstate \end{matrix} \vdash \begin{pmatrix} utxoSt \\ acnt \\ pp \end{pmatrix} \xrightarrow[\text{NEWPP}]{} \begin{pmatrix} utxoSt \\ acnt \\ pp \end{pmatrix}} \quad (17)
$$

Figure 32: New Proto Param Inference Rule

10.7 Complete Epoch Boundary Transition

Finally, it is possible to define the complete epoch boundary transition type, which is defined in Figure 33. In the environment of this transition, we have the slot number, potentially new protocol parameters, and the blocks made this epoch. The state is made up of the the UTxO state, the accounting state, the delegation state, the pool state, the current protocol parameters, and the snapshots.

Epoch environment

$$\mathsf{EpochEnv} = \left(\begin{array}{rcll} e_{new} & \in & \mathsf{Epoch} & \text{upcoming epoch} \\ pp_{new} & \in & \mathsf{PParams} & \text{new protocol parameters} \\ blocks & \in & \mathsf{BlocksMade} & \text{blocks made in the epoch} \end{array} \right)$$

Epoch States

$$\mathsf{EpochState} = \left(\begin{array}{rcll} utxoSt & \in & \mathsf{UTxOState} & \text{utxo state} \\ acnt & \in & \mathsf{Acnt} & \text{accounting} \\ dstate & \in & \mathsf{DState} & \text{delegation state} \\ pstate & \in & \mathsf{PState} & \text{pool state} \\ pp & \in & \mathsf{PParams} & \text{current protocol parameters} \\ ss & \in & \mathsf{Snapshots} & \text{snapshots} \end{array} \right)$$

Epoch transitions

$$\_ \vdash \_ \xrightarrow[\text{EPOCH}]{} \_ \subseteq \mathbb{P}\left(\mathsf{EpochEnv} \times \mathsf{EpochState} \times \mathsf{EpochState}\right)$$

Figure 33: Epoch transition-system types

The epoch transition rule calls SNAP, POOLREAP, and NEWPP in sequence.

$$\vdash \begin{pmatrix} ss \\ utxoSt \end{pmatrix} \xrightarrow[\text{SNAP}]{} \begin{pmatrix} ss' \\ utxoSt' \end{pmatrix} \quad \text{where } e_{new},\ pp,\ dstate,\ pstate,\ blocks$$

$$e_{new},\ pp \vdash \begin{pmatrix} acnt \\ dstate \\ pstate \end{pmatrix} \xrightarrow[\text{POOLREAP}]{} \begin{pmatrix} acnt' \\ dstate' \\ pstate' \end{pmatrix}$$

$$\text{Epoch}\ \frac{e_{new},\ pp_{new},\ dstate',\ pstate'' \vdash \begin{pmatrix} utxoSt' \\ acnt' \\ pp \end{pmatrix} \xrightarrow[\text{NEWPP}]{} \begin{pmatrix} utxoSt'' \\ acnt'' \\ pp' \end{pmatrix}}{e_{new},\ pp_{new},\ blocks \vdash \begin{pmatrix} utxoSt \\ acnt \\ dstate \\ pstate \\ pp \\ ss \end{pmatrix} \xrightarrow[\text{EPOCH}]{} \begin{pmatrix} \boldsymbol{utxoSt''} \\ \boldsymbol{acnt''} \\ \boldsymbol{dstate'} \\ \boldsymbol{pstate'} \\ \boldsymbol{pp'} \\ \boldsymbol{ss'} \end{pmatrix}} \tag{18}$$

Figure 34: Epoch Inference Rule

10.8 Rewards Distribution Calculation

This section defines the reward calculation for the proof of stake leader election. Figure 35 defines the pool reward as described in section 6.5.1 of Kant et al. (2018).

- The function maxPool gives the maximum reward a stake pool can receive in an epoch. This is a fraction of the total available rewards for the epoch. The result depends on the pool's relative stake, the pool's pledge, and the following protocol parameters:
  - a_0, the leader-stake influence
  - n_{opt}, the optimal number of saturated stake pools
- The function movingAvg calculates the new moving average for a given stake pool based on its performance and the protocol parameter:
  - α, the moving average weight
- The function poolReward gives the total rewards available to be distributed to the members of the given pool. It depends on one additional protocol parameter:
  - γ, the moving average exponent

Maximal Reward Function, called $f(s,\sigma)$ in section 6.5.1 of Kant et al. (2018)

$$\mathsf{maxPool} \in \mathsf{PParams} \to \mathsf{Coin} \to [0,\ 1] \to [0,\ 1] \to \mathsf{Coin}$$

$$\mathsf{maxPool}\ pp\ R\ \sigma\ p_r = \left\lfloor \frac{R}{1+a_0} \cdot \left(\sigma' + p' \cdot a_0 \cdot \frac{\sigma' - p' \frac{z_0 - \sigma'}{z_0}}{z_0} \right) \right\rfloor$$

where

$$a_0 = \mathsf{influence}\ pp$$
$$n_{opt} = \mathsf{nopt}\ pp$$
$$z_0 = 1/n_{opt}$$
$$\sigma' = \min(\sigma,\ z_0)$$
$$p' = \min(p_r,\ z_0)$$

Exponential moving average, called $\langle x \rangle_e$ in 6.5.1 of Kant et al. (2018)

$$\mathsf{movingAvg} \in \mathsf{PParams} \to \mathsf{HashKey} \to \mathbb{N} \to \mathbb{R}^{\geq 0} \to \mathsf{Avgs} \to \mathbb{R}$$

$$\mathsf{movingAvg}\ pp\ hk\ n\ \overline{N}\ avgs = \begin{cases} \frac{n}{\max(\overline{N},1)} & hk \notin \mathrm{dom}\ avgs \\ \alpha \cdot \frac{n}{\max(\overline{N},1)} + (1-\alpha) \cdot prev & hk \mapsto prev \in avgs \end{cases}$$

where

$$\alpha = \mathsf{movingAvgWeight}\ pp$$

Actual Reward Function, called $\hat{f}_j$ in section 6.5.1 of Kant et al. (2018)

$$\mathsf{poolReward} \in \mathsf{PParams} \to \mathsf{HashKey} \to \mathbb{N} \to \mathbb{R}^{\geq 0} \to \mathsf{Avgs} \to \mathsf{Coin} \to \mathsf{Coin}$$

$$\mathsf{poolReward}\ pp\ hk\ n\ \overline{N}\ avgs\ maxP = \lfloor avg^{\gamma} \cdot maxP \rfloor$$

where

$$\gamma = \mathsf{movingAvgExp}\ pp$$
$$avg = \mathsf{movingAvg}\ pp\ hk\ n\ \overline{N}\ avgs$$

Figure 35: Functions used in the Reward Calculation

Figure 36 gives the calculation for splitting the pool rewards with its members, as described 6.5.2 of Kant et al. (2018). The portion of rewards allocated to the pool operator and owners is different than that of the members.

- The $\mathsf{r}_{\mathsf{leader}}$ function calculates the leader reward, based on the pool cost, margin, and proportion of the pool's total stake. Note that this reward will go to the reward account specified in the pool registration certificate.
- The $\mathsf{r}_{\mathsf{member}}$ function calculates the member reward, proportionally to their stake after the cost and margin are removed.

Pool leader reward, from section 6.5.2 of Kant et al. (2018)

$$\mathsf{r}_{\mathsf{leader}} \in \mathsf{Coin} \to \mathsf{PoolParam} \to [0,\ 1] \to (0,\ 1] \to \mathsf{Coin}$$

$$\mathsf{r}_{\mathsf{leader}}\ \hat{f}\ pool\ s\ \sigma = \begin{cases} \hat{f} & \hat{f} \leq c \\ c + \left\lfloor (\hat{f} - c) \cdot \left(m + (1 - m) \cdot \frac{s}{\sigma}\right) \right\rfloor & \text{otherwise.} \end{cases}$$

where

$c = \mathsf{poolCost}\ pool$

$m = \mathsf{poolMargin}\ pool$

Pool member reward, from section 6.5.2 of Kant et al. (2018)

$$\mathsf{r}_{\mathsf{member}} \in \mathsf{Coin} \to \mathsf{PoolParam} \to [0,\ 1] \to (0,\ 1] \to \mathsf{Coin}$$

$$\mathsf{r}_{\mathsf{member}}\ \hat{f}\ pool\ t\ \sigma = \begin{cases} 0 & \hat{f} \leq c \\ \left\lfloor (\hat{f} - c) \cdot (1 - m) \cdot \frac{t}{\sigma} \right\rfloor & \text{otherwise.} \end{cases}$$

where

$c = \mathsf{poolCost}\ pool$

$m = \mathsf{poolMargin}\ pool$

Figure 36: Functions used in the Reward Splitting

Finally, the full reward calculation is presented in Figure 37. The calculation is done pool-by-pool.

- The rewardOnePool function calculates the rewards given out to each member of a given pool. The pool leader is identified by the stake key of the pool operator. The function returns both the rewards and the total amount of unrealized potential rewards (ie the difference beween the max reward R and what was actually paid out). The unrealized amount will go to the treasury. Note that rewards attached to unregistered reward accounts will end up in the unrealized amount. Involved in the calculation is:
  - *pstake*, the total amount of stake controlled by the stake pool.
  - *ostake*, the total amount of stake controlled by the stake pool operator and owners
  - σ, the total proportion of stake controlled by the stake pool.
  - $\overline{N}$, the expected number of blocks the pool should have produced.
  - *pledge*, the pool's pledge in lovelace.

- p_r, the pool's pledge, as a proportion of active stake.
- *maxP*, maximum rewards the pool can claim if the pledge is met, and zero otherwise.
- *poolR*, the pool's actual reward, based on its performance.
- *mRewards*, the member's rewards as a mapping of reward accouts to coin.
- *lReward*, the leader's reward as coin.
- *potentialRewards*, the combination of *mRewards* and *lRewards*.
- *rewards*, the restriction of *potentialRewards* to the active reward accounts.
- *unrealized*, difference between R (max rewards) and the sum of all individual rewards paid out.

- The reward function applies rewardOnePool to each registered stake pool, calculating both the full reward mapping and the total unrealized rewards value.

Calculation to reward a single stake pool

$$\begin{aligned}
&\mathsf{rewardOnePool} \in \mathsf{PParams} \to \mathsf{Coin} \to \mathbb{N} \to \mathsf{HashKey} \to \mathsf{PoolParam} \\
&\quad \to \mathsf{Stake} \to \mathsf{Avgs} \to \mathsf{Coin} \to \mathbb{P}\ \mathsf{Addr}_{\mathsf{rwd}} \to (\mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin}) \times \mathsf{Coin} \\
&\mathsf{rewardOnePool}\ pp\ R\ n\ poolHK\ pool\ stake\ avgs\ tot\ addrs_{rew} = (rewards,\ unrealized) \\
&\quad \textbf{where} \\
&\quad pstake = \sum_{_ \mapsto c \in stake} c \\
&\quad ostake = \sum_{\substack{hk_i \to c \in stake \\ hk \in (\mathsf{poolOwners}\ pool)}} c \\
&\quad \sigma = pstake / tot \\
&\quad \overline{N} = \sigma * \mathsf{SlotsPerEpoch} \\
&\quad pledge = \mathsf{poolPledge}\ pool \\
&\quad p_r = pledge / tot \\
&\quad maxP = \begin{cases} \mathsf{maxPool}\ pp\ R\ \sigma\ p_r & pledge \leq ostake \\ 0 & \text{otherwise.} \end{cases} \\
&\quad poolR = \mathsf{poolReward}\ pp\ hk\ n\ \overline{N}\ avgs\ maxP \\
&\quad mRewards = \left\{ \mathsf{addr}_{\mathsf{rwd}}\ \ hk \mapsto \mathsf{r}_{\mathsf{member}}\ poolR\ pool\ \frac{c}{tot}\ \sigma \;\middle|\; hk \mapsto c \in stake,\ \ hk \neq poolHK \right\} \\
&\quad lReward = \mathsf{r}_{\mathsf{leader}}\ poolR\ pool\ \frac{ostake}{tot}\ \sigma \\
&\quad potentialRewards = mReward \cup \{(\mathsf{poolRAcnt}\ pool) \mapsto lReward\} \\
&\quad rewards = addrs_{rew} \lhd potentialRewards \\
&\quad unrealized = R - \left(\sum_{_ \mapsto c \in rewards} c \right)
\end{aligned}$$

Calculation to reward all stake pools

$$\begin{aligned}
&\mathsf{reward} \in \mathsf{PParams} \to \mathsf{BlocksMade} \to \mathsf{Coin} \to \mathbb{P}\ \mathsf{Addr}_{\mathsf{rwd}} \to (\mathsf{HashKey} \mapsto \mathsf{PoolParam}) \\
&\quad \to \mathsf{Avgs} \to \mathsf{Stake} \to (\mathsf{HashKey}_{stake} \mapsto \mathsf{HashKey}_{pool}) \to (\mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin}) \times \mathsf{Coin} \\
&\mathsf{reward}\ pp\ blocks\ R\ addrs_{rew}\ poolParams\ avgs\ stake\ delegs = (rewards,\ unrealized) \\
&\quad \textbf{where} \\
&\quad tot = \sum_{_ \mapsto c \in stake} c \\
&\quad pdata = \left\{ hk \mapsto (p,\ n,\ \mathsf{poolStake}\ hk\ delegs\ stake) \;\middle|\; \begin{array}{l} hk \mapsto p \ \in\ poolParams \\ hk \mapsto n \ \in\ blocks \end{array} \right\} \\
&\quad results = \{hk \mapsto \mathsf{rewardOnePool}\ pp\ R\ n\ hk\ p\ s\ avgs\ tot\ addrs_{rew} \mid hk \mapsto (p,n,s) \in pdata\} \\
&\quad unrealized = \sum_{_ \mapsto (_,\ u) \in results} u \\
&\quad rewards = \bigcup_{_ \mapsto (r,\ _) \in results} r
\end{aligned}$$

Figure 37: The Reward Calculation

10.9 Reward Transition

Figure 38 gives the definitions for the reward transition. The figure lists the accounting fields, denoted by Acnt, which consists of:

- The value *treasury* tracks the amount of coin currently stored in the treasury. Initially there will be no way to remove these funds.
- The value *reserves* tracks the amount of coin currently stored in the reserves. This pot is used to pay rewards.
- The value *rewardPot* to tracks the rewards from the previous epoch that were not payed out.

The figure also defines the reward environment, RewardEnv, and the reward state, RewardState, which combines the accounting fields described above with the UTxO State, the delegation state, and the pool state. The reward state transition type, like all the transition types in this section, has no signal.

Accounting Fields

$$\mathsf{Acnt} = \left(\begin{array}{rcll} \mathit{treasury} & \in & \mathsf{Coin} & \text{treasury pot} \\ \mathit{reserves} & \in & \mathsf{Coin} & \text{reserve pot} \\ \mathit{rewardPot} & \in & \mathsf{Coin} & \text{reward pot} \end{array}\right)$$

Rewards environment

$$\mathsf{RewardEnv} = \left(\begin{array}{rcll} \mathit{pp} & \in & \mathsf{PParams} & \text{protocol parameters} \\ \mathit{blocks} & \in & \mathsf{BlocksMade} & \text{blocks made in the epoch} \\ \mathit{ss} & \in & \mathsf{SnapshotState} & \text{snapshots} \end{array}\right)$$

Rewards States

$$\mathsf{RewardState} = \left(\begin{array}{rcll} \mathit{acnt} & \in & \mathsf{Acnt} & \text{accounting} \\ \mathit{dstate} & \in & \mathsf{DState} & \text{delegation state} \\ \mathit{pstate} & \in & \mathsf{PState} & \text{pool state} \\ \mathit{utxoSt} & \in & \mathsf{UTxOState} & \text{utxo state} \end{array}\right)$$

Rewards transitions

$$\_ \vdash \_ \xrightarrow[\text{REWARD}]{} \_ \subseteq \mathbb{P}\left(\mathsf{RewardEnv} \times \mathsf{RewardState} \times \mathsf{RewardState}\right)$$

Figure 38: Rewards transition-system types

Figure 39 captures the potential movement of funds in the entire system, taking every transition system in this document into account. Value is moved between accounting pots, but the total amount of value in the system remains constant. In particular, the red subgraph represents the inputs and outputs to the "total pot" used during the reward calculation in Figure 40. The blue arrows represent the movement of funds that pass through the "total pot".

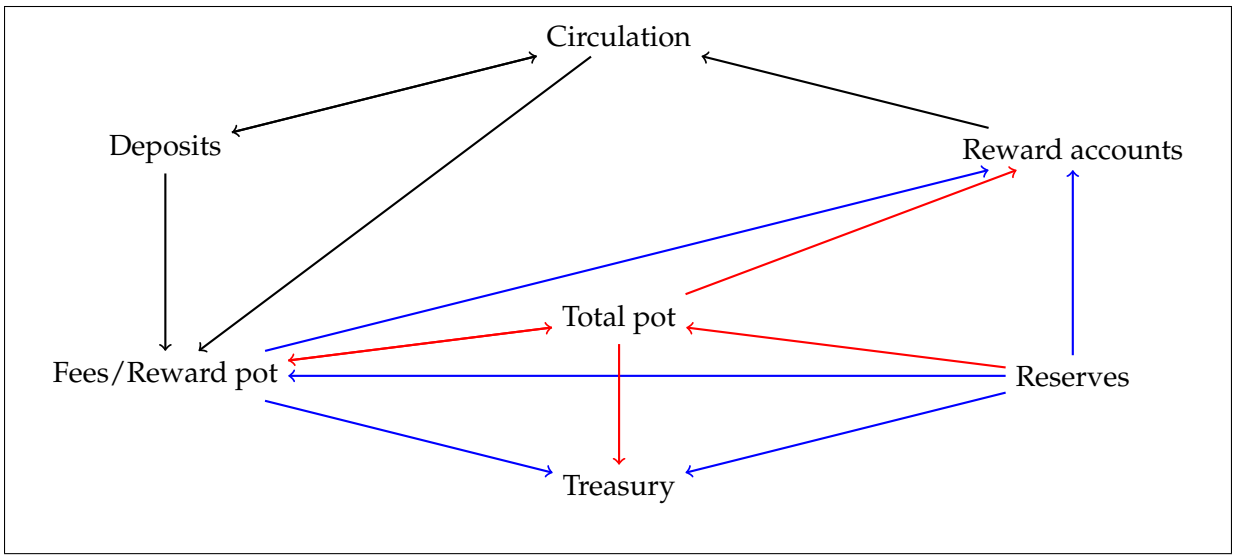


Figure 39: Preservation of Value

Figure 40 defines the reward transition rule. The reward transition has no preconditions.

- First we calculate *totalPot*, the total amount of coin avail for rewards this epoch, , as described in section 6.4 of Kant et al. (2018). It consists of four pots:
  - The fee pot, containing the transaction fees from the epoch.
  - The amount of coin in the deposit pot that is no longer needed, due to decay.
  - The reward pot, which is the left-over rewards from the previous epoch.
  - Some amount of monetary expansion from the reserves, as determined by the ρ protocol parameter.

  Note that the fee pot and the decayed amount are taken from the snapshot taken at the epoch boundary. (See Figure28).
- Some proportion of the total pot is moved to the treasury, as determined by the τ protocol parameter. The remaining pot is called the R, just as in section 6.5 of Kant et al. (2018).
- The rewards are calculated, using the oldest stake distribution snapshot (the one labeled "go"). As given by maxPool, each pool can receive a maximal amount, determined by its performance. The difference between the maximal amount and the actual amount received is moved to the treasury.
- The reward pot is now set to R less the total amount of rewards actually paid out and the unrealized rewards given to the treasury.
- The moving averages are updated. Note the averages were already computed on the previous epoch boundary, as a part of the stake distribution calculation, and could therefore be cached in order to prevent calculating them twice.
- The fee pot is reduced by *feeSS*.

Note that fees are not explicitly removed from any account: the fees come from transactions paying them, and are accounted for whenever transactions are processed, and the deposit decay value comes from returning smaller refunds for deposits than were paid upon depositing.

$$\begin{pmatrix} \_ \\ \_ \\ pstate_{go} \\ poolsSS \\ blocksSS \\ feeSS \end{pmatrix} = ss$$

$$(stake,\ delegs) = pstate_{go}$$

$$expansion = \lfloor (\mathsf{rho}\ pp) \cdot reserves \rfloor$$

$$totalPot = feeSS + rewardPot + expansion$$

$$newTreasury = \lfloor (\mathsf{tau}\ pp) \cdot totalPot \rfloor$$

$$R = totalPot - newTreasury$$

$$rewards',\ unrealized = \mathsf{reward}\ pp\ blocksSS\ R\ (\mathrm{dom}\ rewards)\ poolsSS\ avgs\ stake\ delegs$$

$$newTreasury' = newTreasury + unrealized$$

$$paidRewards = \left(\sum_{\_ \mapsto c \in rewards'} c \right) + unrealized$$

$$avgs' = \mathsf{updateAvgs}\ pp\ avgs\ blocks\ delegs\ stake$$

Rewards

$$pp,\ blocks,\ ss \vdash \begin{pmatrix} treasury \\ reserves \\ rewardPot \\ \\ stkeys \\ rewards \\ delegations \\ ptrs \\ \\ stpools \\ poolParams \\ retiring \\ avgs \\ \\ utxo \\ deposits \\ fees \end{pmatrix} \xrightarrow[\text{REWARD}]{} \begin{pmatrix} \boldsymbol{treasury} & + & \boldsymbol{newTreasury'} \\ \boldsymbol{reserves} & - & \boldsymbol{expansion} \\ \boldsymbol{R} & - & \boldsymbol{paidRewards} \\ \\ stkeys & & \\ \boldsymbol{rewards} & \cup_+ & \boldsymbol{rewards'} \\ delegations & & \\ ptrs & & \\ \\ stpools & & \\ poolParams & & \\ retiring & & \\ \boldsymbol{avgs'} & & \\ \\ utxo & & \\ deposits & & \\ \boldsymbol{fees} & - & \boldsymbol{feeSS} \end{pmatrix} \tag{19}$$

Figure 40: Rewards inference rules

11 Properties

In this section we discuss the properties which we want the ledger to have. One goal is to include these properties in the executable specification for doing property-based testing or formal verification.

11.1 Validity of a Ledger State

Many properties only make sense when applied to a valid ledger state. In informal terms, a valid ledger state l can only be reached when starting from an initial state l_0 (genesis state) and only executing state transition rules as specified in Section 7 for UTxO or Section 8 for delegation.

$$
\begin{aligned}
Genesis_{Id} &\in && \mathsf{TxId} \\
Genesis_{Out} &\in && \mathsf{TxOut} \\
Genesis_{UTxO} &\coloneqq && \{Genesis_{Id}, \emptyset\} \mapsto Genesis_{Out} \\
\mathsf{ledgerState} &\in && \begin{pmatrix} \mathsf{UTxO} \\ \mathsf{DPState} \end{pmatrix} \\
\\
\mathsf{getUTxO} &\in && \mathsf{ledgerState} \to \mathsf{UTxO} \\
\mathsf{getUTxO} &\coloneqq && (utxo, \_) \to utxo
\end{aligned}
$$

Figure 41: Definitions and Functions for Valid Ledger State

In Figure 41 $Genesis_{Id}$ marks the transaction identifier of the initial coin distribution, where $Genesis_{Out}$ represents the initial UTxO. It should be noted that no corresponding inputs exists, i.e., the transaction inputs are the empty set for the initial transaction. An element of ledgerState is a tuple of UTxO and delegation witness state (DPState).

Definition 11.1 (Valid Ledger State).

$$
\begin{aligned}
\forall l_0, \ldots, l_n \in \mathsf{ledgerState}, l_0 = & \begin{pmatrix} \{Genesis_{UTxO}\} \\ \begin{pmatrix} \emptyset \\ \emptyset \end{pmatrix} \end{pmatrix} \\
& \implies \forall 0 < i \leq n, (\exists tx_i \in \mathsf{Tx}, l_{i-1} \xrightarrow[\text{LEDGER}]{tx_i} l_i) \implies \mathsf{validLedgerState}\ l_n
\end{aligned}
$$

Definition 11.1 defines a valid ledger state reachable from the genesis state via valid UTxO, stake delegation or stake pool transactions. This gives a constructive rule how to reach a valid ledger state.

11.2 Ledger Properties

The following properties state the desired features of updating a valid ledger state.

Property 11.1 (Preserve Balance Modulo Fee).

$$
\begin{aligned}
\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l & \\
& \implies \forall tx \in \mathsf{Tx}, l \xrightarrow[\text{UTXOW}]{tx} l' \\
& \implies \mathsf{destroyed}\ pcutxostKeysrewardstx = \mathsf{created}\ pcstPoolstx
\end{aligned}
$$

Property 11.1 states that for each valid ledger l, if a transaction tx is added to the ledger via the state transition rule $utxow$ to the new ledger state l', the balance of the UTxOs in l equals the balance of the UTxOs in l' in the sense that the amount of created value in l' equals the amount of destroyed value in l. This means that the total amount of value is left unchanged by a transaction.

Property 11.2 (Preserve Balance Restricted to TxIns in Balance of TxOuts).

$$\begin{aligned}&\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l \\ &\implies \forall tx \in \mathsf{Tx}, l \xrightarrow[\text{UTXOW}]{tx} l' \implies \mathsf{ubalance}(\mathsf{txins}\ tx \lhd \mathsf{getUTxO}\ l) = \mathsf{ubalance}(\mathsf{outs}\ tx) + \mathsf{txfee}\ tx\end{aligned}$$

Property 11.2 states the more detailed relation of the balances change. For ledgers l, l' and a transaction tx as above, the balance of the UTxOs of l restricted to those whose domain is in the set of transaction inputs of tx equals the balance of the transaction outputs of tx minus the transaction fees.

Property 11.3 (Preserve Outputs of Transaction).

$$\begin{aligned}&\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l \\ &\implies \forall tx \in \mathsf{Tx}, l \xrightarrow[\text{UTXOW}]{tx} l' \implies \forall out \in \mathsf{outs}\ tx, out \in \mathsf{getUTxO}\ l'\end{aligned}$$

Property 11.3 states that for every ledger states l, l' and transaction tx as above, all output UTxOs of tx are in the UTxO set of l', i.e., they are now available as unspent transaction output.

Property 11.4 (Eliminate Inputs of Transaction).

$$\begin{aligned}&\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l \\ &\implies \forall tx \in \mathsf{Tx}, l \xrightarrow[\text{UTXOW}]{tx} l' \implies \forall in \in \mathsf{txins}\ tx, in \notin \mathsf{dom}(\mathsf{getUTxO}\ l')\end{aligned}$$

Property 11.4 states that for every ledger states l, l' and transaction tx as above, all transaction inputs in of tx are not in the domain of the UTxO set of l', i.e., these are no longer available to spend.

Property 11.5 (Completeness and Collision-Freeness of new Transaction Ids).

$$\begin{aligned}&\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l \\ &\implies \forall tx \in \mathsf{Tx}, l \xrightarrow[\text{UTXOW}]{tx} l' \implies \forall utxo' \in \mathsf{outs}\ tx, utxo' \in \mathsf{getUTxO}\ l' \wedge \\ &(utxo' = ((txId', \_) \mapsto \_) \implies \forall utxo \in \mathsf{getUTxO}\ l, utxo = ((txId, \_) \mapsto \_) \implies txId' \neq txId\end{aligned}$$

Property 11.5 states that for ledger states l, l' and a transaction tx as above, the UTxOs of l' contain all newly created UTxOs and the referred transaction id of each new UTxO is not used in the UTxO set of l.

Property 11.6 (Absence of Double-Spend).

$$\forall l_0, \ldots, l_n \in \mathsf{ledgerState}, l_0 = \begin{pmatrix} \{Genesis_{UTxO}\} \\ \begin{pmatrix} \emptyset \\ \emptyset \end{pmatrix} \end{pmatrix} \wedge \mathsf{validLedgerState}\ l_n$$
$$\implies \forall 0 < i \leq n, tx_i \in \mathsf{Tx}, l_{i-1} \xrightarrow[\text{LEDGER}]{tx_i} l_i \wedge \mathsf{validLedgerState}\ l_i$$
$$\implies \forall j < i, \mathsf{txins}\ tx_j \cap \mathsf{txins}\ tx_i = \emptyset$$

Property 11.6 states that for each valid ledger state l_n reachable from the genesis state, each transaction t_i does not share any input with any previous transaction t_j. This means that each output of a transition is spent at most once.

11.3 Ledger State Properties for Delegation Transitions

$$\begin{aligned}
\mathsf{getStKeys} &\in && \mathsf{ledgerState} \to \mathbb{P}\ \mathsf{HashKey} \\
\mathsf{getStKeys} &:= && (_, (stKeys, _, _), _) \to stkeys \\
\\
\mathsf{getRewards} &\in && \mathsf{ledgerState} \to \mathsf{Addr}_{\mathsf{rwd}} \mapsto \mathsf{Coin} \\
\mathsf{getRewards} &:= && (_, (_, rewards, _), _) \to rewards \\
\\
\mathsf{getDelegations} &\in && \mathsf{ledgerState} \to \mathsf{HashKey} \mapsto \mathsf{HashKey} \\
\mathsf{getDelegations} &:= && (_, (_, _, delegations), _) \to delegations \\
\\
\mathsf{getStPools} &\in && \mathsf{ledgerState} \to \mathsf{HashKey} \mapsto \mathsf{DCert}_{\mathsf{regpool}} \\
\mathsf{getStPools} &:= && (_, _, (stpools, _)) \to stpools \\
\\
\mathsf{getRetiring} &\in && \mathsf{ledgerState} \to \mathsf{HashKey} \mapsto \mathsf{Epoch} \\
\mathsf{getRetiring} &:= && (_, _, (_, retiring)) \to retiring
\end{aligned}$$

Figure 42: Definitions and Functions for Stake Delegation in Ledger States

Property 11.7 (Registered Staking Key with Zero Rewards).

$$\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l$$
$$\implies \forall c \in \mathsf{DCert}_{\mathsf{regkey}}, l \xrightarrow[\text{DELEGW}]{c} l' \implies \mathsf{author}\ c = hk$$
$$\implies hk \in \mathsf{getStKeys}\ l' \wedge (\mathsf{getRewards}\ varrewards)[hk] = 0$$

Property 11.7 states that for each valid ledger state l, if a delegation transaction of type $\mathsf{DCert}_{\mathsf{regkey}}$ is executed, then in the resulting ledger state l', the set of staking keys of l' includes the key hk associated with the key registration certificate and the associated reward is set to 0 in l'.

Property 11.8 (Deregistered Staking Key).

$$\begin{aligned}\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l \\ \implies \forall c \in \mathsf{DCert}_{\mathsf{deregkey}}, l \xrightarrow[\text{DELEGW}]{c} l' \implies \mathsf{author}\ c = hk \\ \implies hk \notin \mathsf{getStKeys}\ l' \wedge (\mathsf{dom}(\mathsf{getRewards}\ l') \cup \mathsf{dom}(\mathsf{getDelegations}\ l')) \cap \{hk\} = \emptyset\end{aligned}$$

Property 11.8 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert}_{\mathsf{deregkey}}$, the staking key hk associated with the deregistration certificate is not in the set of staking keys of l' and is not in the domain of neither the rewards nor the delegation map of l'.

Property 11.9 (Delegated Stake).

$$\begin{aligned}\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l \\ \implies \forall c \in \mathsf{DCert}_{\mathsf{delegate}}, l \xrightarrow[\text{DELEGW}]{c} l' \implies \mathsf{author}\ c = hk \\ \implies hk \in \mathsf{getStKeys}\ l \wedge (\mathsf{getDelegations}\ l')[hk] = \mathsf{pool}\ c\end{aligned}$$

Property 11.9 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert}_{\mathsf{delegate}}$, the staking key hk associated with the deregistration certificate is in the set of staking keys of l and delegates to the staking pool associated with the delegation certificate in l'.

11.4 Ledger State Properties for Staking Pool Transitions

Property 11.10 (Registered Staking Pool).

$$\begin{aligned}\forall l, l' \in \mathsf{ledgerState} : \mathsf{validLedgerstate}\ l \\ \implies \forall c \in \mathsf{DCert}_{\mathsf{regpool}}, l \xrightarrow[\text{POOL}]{c} l' \implies \mathsf{author}\ c = hk \\ \implies (\mathsf{getStPools}\ l')[hk] = c \wedge hk \notin \mathsf{getRetiring}\ l'\end{aligned}$$

Property 11.10 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert}_{\mathsf{regpool}}$, the key hk is associated with the author of the pool registration certificate in *stpools* of l' and that hk is not in the set of retiring stake pools in l'.

Property 11.11 (Start Staking Pool Retirement).

$$\begin{aligned}\forall l, l' \in \mathsf{ledgerState}, cepoch \in \mathsf{Epoch} : \mathsf{validLedgerstate}\ l \\ \implies \forall c \in \mathsf{DCert}_{\mathsf{retirepool}}, l \xrightarrow[\text{POOL}]{c} l' \\ \implies e = \mathsf{retire}\ c \wedge cepoch < e < cepoch + \mathsf{E}_{\mathsf{max}} \wedge \mathsf{author}\ c = hk \\ \implies (\mathsf{getRetiring}\ l')[hk] = e \wedge hk \in \mathsf{dom}(\mathsf{getStPools}\ l)\end{aligned}$$

Property 11.11 states that for l, l' as above but with a delegation transition of type $\mathsf{DCert}_{\mathsf{retirepool}}$, the key hk is associated with the author of the pool registration certificate in *stpools* of l' and that hk is not in the set of retiring stake pools in l'.

Property 11.12 (Stake Pool Reaping).

$$\begin{aligned}\forall l, l' \in \mathsf{ledgerState}, cepoch \in \mathsf{Epoch} : \mathsf{validLedgerstate}\ l \\ \implies l \xrightarrow[\text{POOLREAP}]{} l' \implies \forall retire = retiring^{-1} cepoch, retired \neq \emptyset \\ \wedge\ retire \subseteq \mathsf{dom}(\mathsf{getStPool}\ l) \wedge retire \cap \mathsf{dom}(\mathsf{getStPool}\ l') = \emptyset \\ \wedge\ retire \subseteq \mathsf{dom}(\mathsf{getRetiring}\ l) \wedge retire \cap \mathsf{dom}(\mathsf{getRetiring}\ l') = \emptyset\end{aligned}$$

Property 11.12 states that for l, l' as above but with a delegation transition of type poolreap, there exist registered stake pools in l which are associated to stake pool registration certificates and which are to be retired at the current epoch *cepoch*. In l' all those stake pools are removed from the maps *stpools* and *retiring*.

11.5 Properties of Numerical Calculations

The numerical calculations for refunds and rewards calculation in (see Section 10) are also required to have certain properties. In particular we need to make sure that the functions that use non-integral arithmetic have properties which guarantee consistency of the system. Here, we state those properties and formulate them in a way that makes them possible to use properties-based testing for validation in the executable spec.

Property 11.13 (Minimal Refund). The function refund takes a value, a minimal percentage, a decay parameter and a duration. It must guarantee that the refunded amount is within the minimal refund (off-by-one for rounding / floor) and the original value.

$$\begin{aligned}\forall d_{val} \in \mathbb{N}, d_{min} \in [0,1], \lambda \in (0,\infty), \delta \in \mathbb{N} \\ \implies \max(0, d_{val} \cdot d_{min} - 1) \leq \left\lfloor d_{val} \cdot (d_{min} + (1 - d_{min}) \cdot e^{-\lambda \cdot \delta}) \right\rfloor \leq d_{val}\end{aligned}$$

Property 11.14 (Exponential Moving Average). The function movingAvg calculates the exponential moving average, dividing the number of blocks created by the pool by the expected number of slots the pool is elected leader (or 1 if the expected number is below 1). It guarantees that the result is (i) non-negative and (ii) if a previous moving average has already been calculated, the new moving average lies between the minimum and maximum of the old and new calculated value. With $current := \frac{n}{\max(\overline{N},1)}$ this is trivial for (i), for (ii) it is

$$\begin{aligned}\forall \alpha \in [0,1], n \in \mathbb{N}, \overline{N} \in \mathbb{R}^{\geq 0}, prev \in \mathbb{R}^{\geq 0} \\ \implies 0 \leq \min(prev, current) \leq \alpha \cdot current + (1 - \alpha) \cdot prev \leq \max(prev, current)\end{aligned}$$

Property 11.15 (Maximal Pool Reward). The maximal pool reward is the expected maximal reward paid to a stake pool. The sum of all these rewards cannot exceed the total available reward, let *Pool* be the set of active stake pools:

$$\forall R \in Coin : \sum_{p \in Pools} \left\lfloor \frac{R}{1 + p_{a_0}} \cdot \left(p_{\sigma'} + p_{p' \cdot a_0} \cdot \frac{p_{\sigma'} - p_{p'} \cdot \frac{p_{z_0} - p_{\sigma'}}{p_{z_0}}}{p_{z_0}} \right) \right\rfloor \leq R$$

Property 11.16 (Actual Reward). The actual reward for a stake pool in an epoch is calculated by the function poolReward. The actual reward per stake pool is non-negative and bounded by the

maximal reward for the stake pool, with avg being the calculated moving average of the stake pool and $maxP$ being the maximal reward for the stake pool, we get:

$$\forall \gamma \in [0,1] \implies 0 \leq \lfloor avg^{\gamma} \cdot maxP \rfloor \leq maxP$$

TODO

The property (11.16) requires that $avg \in [0,1]$, else the actual reward can exceed the maximal reward. This is not true, we need to take into account the rewards for all stake pools.

The two functions $\mathsf{r}_{\mathsf{leader}}$ and $\mathsf{r}_{\mathsf{member}}$ are closely related as they do split the reward between the pool leader and the members.

Property 11.17 (Reward Splitting). The reward splitting is done via $\mathsf{r}_{\mathsf{leader}}$ and $\mathsf{r}_{\mathsf{member}}$, i.e., a split between the pool leader and the pool members using the pool cost c and the pool margin m. Therefore the property relates the total reward $\hat{f}$ to the split rewards in the following way:

$$\forall m \in [0,1], c \in Coin \implies c + \left\lfloor (\hat{f} - c) \cdot (m + (1-m)) \cdot \frac{s}{\sigma} \right\rfloor + \sum_{j} \left\lfloor (\hat{f} - c) \cdot (1-m) \cdot \frac{t_j}{\sigma} \right\rfloor \leq \hat{f}$$

Property 11.18 (Accounting transition). The ACCNT transition rule has the following properties in order to ensure non-negative values:

- $obl \leq deposits$
- $\forall \rho \in [0,1] : expansion = \lfloor \rho \cdot reserves \rfloor \leq reserves$
- $\forall \tau \in [0,1] : newTreasury = \lfloor \tau \cdot totalPool \rfloor \leq totalPool$
- $totalPool - newTreasury - paidRewards \geq 0$

where $paidRewards$ is the sum of rewards paid to the stake pool members.

12 Non-Integral Calculations

In the ledger there are several cases where non-integral calculations are required. This does concern the delegation transitions, not value transactions.

12.1 Types of Non-Integral Calculations

The specification employs non-integral calculations for different mathematical operations. Table 1 shows the function and transition rules that use non-integral calculations and which type.

name	page	multiplication	division	exponential function	exponentiation
refund	18	✓		✓	
maxPool	48	✓	✓		
movingAvg	48	✓	✓		
poolReward	48	✓		✓	✓
$\mathsf{r}_{\mathsf{leader}}$	49	✓	✓		
$\mathsf{r}_{\mathsf{member}}$	49	✓	✓		
rewardOnePool	51	✓	✓		
updateAvgs	36	✓	✓		
REWARD	55	✓			

Table 1: Functions with Non-Integral Calculation

The transcendental exponential function is used in reward and refund calculation to model the decay of the deposit values. The pool reward uses exponentiation to calculate a pool's ranking.

The domain for the exponential function are the non-negative reals, more precisely the distribution parameter $\lambda \in (0, \infty)$ multiplied by a discrete non-negative duration δ.

The domain of the base of the exponentiation in poolReward are the non-negative reals resulting from the calculation in movingAvg, the exponent γ is a constant take from the protocol parameters.

12.2 Implementation of Non-Integer Calculations

The large part consists of multiplication and division which can easily be done using fractional arithmetic to the desired precision. The precision necessary is bounded by the ability to represent a single lovelace in all calculations.

12.2.1 Function Simplification

The transcendental function e^x can be approximated using different approaches, depending on the desired accuracy. In general, one uses the exponential laws $e^x = 1/e^{-x}$ and $e^x = \left(e^{\frac{x}{n}}\right)^n, n \in \mathbb{N}$ to reduce the approximation to the unit interval and apply fast integral exponentiation afterwards.

Exponentiation is implemented using the law $a^b = e^{\ln(a^b)} = e^{b\ln(a)}$. This therefore requires being able to calculate e^x and $\ln(x)$. The approximation of the natural logarithm can be approximated using different approaches, again, depending on the desired accuracy. Most approximations work for $\ln(x), x \in [1, c)$ with some $c > 0$. One then uses the law $\log_b(x) = \log_b(\frac{x}{b^n} b^n)$ where $n \in \mathbb{N}$ is chosen in such a way that $\frac{x}{b^n} \in [1, c)$. Using this, one can separate the calculation of the integral and decimal part as follows:

$$\log_b(\frac{x}{b^n}b^n) = \log_b(b^n) + \log_b(\frac{x}{b^n}) = n + \log(\frac{x}{b^n})$$

12.2.2 Properties of Function Approximation

There are several properties that approximations of the transcendental functions are expected to have. In the following let $\ln'(x)$ be the approximation of $\ln(x)$, $\exp'(x)$ be the approximation of e^x and $x \star y$ the approximation of x^y.

Property 12.1 (**monotonicity**)**.** Both $\exp'$ and $\ln'$ must be monotone on their respective domains.

In order to guarantee correctness of the approximations, we also require that the mathematical laws are fulfilled. For some small $\epsilon > 0$, define $x \approx y \Leftrightarrow |x - y| < \epsilon$.

Property 12.2 (**Mathematical Laws**)**.** The following mathematical laws state the requirements for the approximations of the $\ln'$ and $\exp'$ function:

- $\ln'(x \cdot y) \approx \ln'(x) + \ln'(y)$
- $\ln'(x^y) \approx y \cdot \ln'(x)$
- $\ln'(\exp'(x)) \approx \exp'(\ln'(x)) \approx x$
- $x, y \in [0, 1] \implies x \star y \in [0, 1]$
- $x, y, z \in [0, 1], x > 0 \implies (z \star \frac{1}{x}) \star y \approx (z \star y) \star \frac{1}{x}$
- $\exp'(x + y) = \exp'(x) + \exp'(y)$

A Proofs

For the proofs we use the automated theorem prover MetiTarski Akbarpour and Paulson (2010) which is specialized for proofs over real arithmetic, including elementary functions.

Proof. The property (11.13) (p. 60) for the minimal refund can be proven automatically via

```
fof(minimal_refund, conjecture,
! [Dmin, Lambda, Delta, Dval] :
((Dmin : (=0,1=) & Lambda > 0 & Delta > 0 & Dval > 0
=>
Dval*Dmin >= 0 &
(Dval * (Dmin + (1 - Dmin) * exp(-Lambda * Delta))) : (=Dval * Dmin, Dval=)))).

fof(floor_lower_upper, conjecture,
! [X] :
(X >= 0 => X - 1 <= floor(X) & floor(X) <= X)).
```

`minimal_refund` shows that the resulting value is within the interval $[d_{val} \cdot d_{min}, d_{val}]$ and that $d_{val} \cdot d_{min}$ is non-negative, while `floor_lower_upper` shows that the floor of a value x has an upper bound x and lower bound $x - 1$.

□

Proof. The property (11.14) (p. 60) for the bounded moving average can be proven automatically via

```
fof(simple_moving_average, conjecture,
! [Alpha, Prev, N, Nexpected] :
(Alpha : (=0, 1=) & N >= 0 & Nexpected >= 0 & Prev >= 0
=>
0 <= N/max(Nexpected, 1) &
N/max(Nexpected, 1) <= N &
Alpha*N/max(Nexpected, 1) + (1 - Alpha) * Prev <= max(Prev, N/max(Nexpected, 1)) &
min(Prev, N/max(Nexpected, 1)) <= Alpha*N/max(Nexpected, 1) + (1 - Alpha) * Prev)).
```

□

Proof. For the property (11.17) (p. 61) for reward splitting between we actually show a stronger one, by removing the floor function. Using the fractional values we get an upper bound for the real value, and showing that this upper bound is bounded by $\hat{f}$ we show that the real value is also bounded by $\hat{f}$. To eliminate the sum, we use the identity $\frac{s+\sum_j t_j}{\sigma} = 1$, see the definition of σ in Kant et al. (2018). Using this, we show for $\hat{f} > c$

$$
\begin{aligned}
0 \leq c + (\hat{f} - c) \cdot (m + (1 - m)) \cdot \frac{s}{\sigma} + \sum_j (\hat{f} - c) \cdot (1 - m) \cdot \frac{t_j}{\sigma} \leq \hat{f} \\
\Leftrightarrow 0 \leq c + (\hat{f} - c) \cdot m \cdot \frac{s}{\sigma} + (\hat{f} - c) \cdot (1 - m) \cdot \frac{s + \sum_j t_j}{\sigma} \leq \hat{f} \\
\Leftrightarrow 0 \leq c + (\hat{f} - c) \cdot m \cdot \frac{s}{\sigma} + (\hat{f} - c) \cdot (1 - m) \leq \hat{f}
\end{aligned}
$$

This can be proven automatically using

```
fof(reward_splitting, conjecture,
! [C, F, M, S, Sigma] :
(
M : (=0, 1=) & C >= 0 & F > C & Sigma : (0, 1=) & S : (=0, Sigma=)
=>
C + (F - C) * M * S / Sigma + (F - C) * (1 - M) <= F &
0 <= C + (F - C) * M * S / Sigma + (F - C) * (1 - M))).
```

□

References

Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reasoning*, 44(3):175–205, 2010. doi: 10.1007/s10817-009-9149-2. URL https://doi.org/10.1007/s10817-009-9149-2.

IOHK Formal Methods Team. Small step semantics for cardano, 2018. URL https://github.com/input-output-hk/cardano-chain/blob/master/specs/semantics/latex/small-step-semantics.tex.

IOHK Formal Methods Team. ?? - shelley consensus. TODO.

Philipp Kant, Lars Brünjes, and Duncan Coutts. Design specification for delegation and incentives in cardano, 2018. URL https://github.com/input-output-hk/fm-ledger-rules/tree/master/docs/delegation_design_spec.

Joachim Zahnentferner. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. *Cryptology ePrint Archive, Report 2018/262*, 2018. URL https://eprint.iacr.org/2018/262.